J. BUTTERFIELD

MASCHINENORIENTIERTE C64-PROGRAMMIERUNG

BUTTERFIELD'S LEHRBUCH

DER MASCHINEN- UND ASSEMBLERSPRACHE FÜR ALLE COMMODORE COMPUTER

64

HANSER

PHI

Jim Butterfield

Maschinenorientierte C 64-Programmierung

Jim Butterfield

Maschinenorientierte C64-Programmierung

Butterfield's Lehrbuch der Maschinen- und Assemblersprache für alle Commodore-Computer

Übersetzt von Dr. Karl F. Weibezahn

Eine Coedition der Verlage Carl Hanser und Prentice-Hall International

Titel der Originalausgabe:
Machine Language for the Commodore 64 and Other Commodore Computers

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Butterfield, Jim:
[Maschinenorientierte C-vierundsechzig-Programmierung]
Maschinenorientierte C64-Programmierung : Butterfield's Lehrbuch d. Maschinen- u. Assemblersprache für alle Commodore-Computer / Jim Butterfield
Übers. von Karl F. Weibezahn. – München ; Wien : Hanser ; London : Prentice-Hall Internat., 1985. –
Einheitssacht.: Machine language for the Commodore 64 and other Commodore computers ⟨dt.⟩
ISBN 3-446-14378-5 (Hanser)
ISBN 0-89303-652-8 (Prentice-Hall)

Eine Coedition der Verlage:
Carl Hanser Verlag München Wien
Prentice-Hall International Inc., London

Gesamtherstellung: Appl, Wemding
Umschlaggestaltung: Christa Quilici und Günther Gerstmayer

Printed in Germany

Vorwort

Das vorliegende Buch ist hauptsächlich als Lehrbuch gedacht. Darüberhinaus enthält es jedoch eine Vielzahl von Informationen, die der Leser für seine Weiterbildung nutzen kann.

Man benötigt keinerlei Vorkenntnisse in der Maschinensprache-Programmierung. Dagegen erweist es sich als nützlich, wenn dem Leser die Programmierung in einer anderen Sprache nicht fremd ist, sodaß er Begriffe wie zum Beispiel 'Schleifen' und 'Entscheidungen' versteht.

Anfänger haben vielleicht den Eindruck, daß in diesem Buch das Material in zu schneller Folge präsentiert wird. Versuchen Sie dennoch, Schritt zu halten; wenn es notwendig erscheint, wechseln Sie zu den Beispielen über und kehren dann zurück, um eine schwierige Passage abermals zu lesen.

Lesern mit ein wenig Erfahrung in Maschinensprache erscheinen dagegen einige Passagen des Buches vielleicht zu einfach; zum Beispiel könnten Sie mit der Hexadezimalnotation schon vertraut sein und brauchen diesen Absatz nicht zu lesen. Sollte das der Fall sein, so überspringen Sie ihn. Dagegen sollten Sie an jedem Programmvorhaben teilnehmen. Haben Sie einen Punkt nicht verstanden, so wird er Ihnen sicherlich während der Übungen klar werden. Programmieren erlernt man auf praktischem Wege. Ein Anfänger muß unbedingt einfache Dinge über seine Maschine lernen, um ein Gefühl der Sicherheit zu bekommen. Solche einfachen aber notwendigen Elemente könnte man auf folgende Weise aufzählen:

- Maschinensprache. Das ist der Gegenstand dieses Buches, aber man kann ihn ohne die beiden folgenden Begriffe nicht verstehen.
- Maschinenarchitektur. Jede Theorie über Maschinensprache wird dem Leser solange unverständlich erscheinen, bis er über folgendes Bescheid weiß: wo kann ein Programm im Speicher abgelegt werden, wie erreicht man eine Bildschirmdarstellung oder wie erfolgt die Eingabe über die Tastatur.
- Maschinensprachliche Hilfsmittel. Um den Computer irgendeine maschinensprachliche Anweisung ausführen zu lassen, ist es unabdingbar, einen Maschinensprachemonitor zu benutzen, um Inhalte des Computerspeichers zu lesen oder zu verändern. Die Benutzung eines einfachen Assemblers und die Fehlersuche (Debugging) werden Ihnen, wenn Sie sie erst einmal kennengelernt haben, ebenfalls leicht fallen. Vorher jedoch ist es schwierig, Ihre Maschine dazu zu bringen, überhaupt etwas zu tun.

Die Grundlagen der Toncodierung sind wichtig. Diese werden auch ausführlich behandelt, stellen aber nur eine Nebenthematik dieses Buches dar. Das eigentliche Ziel ist folgendes: es ist einfach, Aufgaben auf die richtige, und schwieriger, sie auf die falsche Weise zu lösen. Durch die frühe Einführung guter Programmierbeispiele wird der Leser davon abgehalten nach einem schwierigeren Weg in der Programmierung Ausschau zu halten.

Besonders soll noch darauf hingewiesen werden, daß dieses Buch sich hauptsächlich mit Maschinensprache beschäftigt und nicht mit Assembler. Assembler sind fantastisch, aber für einen Anfänger bereits zu fortgeschritten. Ich möchte den Leser zunächst einmal dazu bringen, daß er eine Vorstellung davon bekommt, wie die Bytes eines Programms innerhalb des Computerspeichers verteilt sind. Erst wenn dieses Konzept klar verstanden ist, können er oder sie die größere Möglichkeit und Flexibilität, die ein Assembler bietet, nutzen.

Anmerkung für den Leser

Dieses Buch führt Anfänger in die Grundlagen der Maschinensprache ein: es beschreibt, was Maschinensprache ist, wie diese funktioniert und wie man damit programmiert.

Es basiert auf einem intensiven Zwei-Tage Kurs in Maschinensprache, der in den letzten fünf Jahren häufig abgehalten wurde.

Der Leser dieses Buches sollte einen Computer zur Hand haben: am besten lernt man durch Nachvollziehen und nicht durch Lesen. Nachdem er das Lehrmaterial dieses Buches beendet hat, wird der Leser eine solide Vorstellung der Grundlagen der Maschinensprache erworben haben. Es gibt sicher noch mehr zu lernen, aber zu diesem Zeitpunkt sollte man in der Lage sein, anderes Material aus Büchern und Magazinen an den jeweils eigenen Computer anzupassen.

Zur Zeit der Publikation unterliegt der Commodore 264 noch einer Entwurfsänderung. Der Name wurde zu „PLUS/4“ geändert. Außerdem wurde eine verwandte Maschine, der Commodore 16 angekündigt. Detaillierte Entwurfsinformationen sind nicht verfügbar. Die Information, die in diesem Buch für den Commodore 264 gegeben wird, sollte im allgemeinen jedoch genau sein.

Einleitung

Warum soll man die Maschinensprache erlernen? Es gibt dafür drei Gründe. Erstens, wegen ihrer Geschwindigkeit; Maschinenprogramme sind schnell. Zweitens, wegen der Vielseitigkeit; alle anderen Sprachen sind in irgendeiner Weise begrenzt, Maschinenspracheprogramme dagegen nicht. Drittens, für das Verständnis; da der Computer in Wirklichkeit nur in Maschinensprache arbeitet, stellt diese den Schlüssel zum eigentlichen Verständnis der Maschinenoperationen dar.

Das ist schwierig? In Wirklichkeit nicht. Es ist knifflig, aber nicht schwierig. Einzelne Maschinensprache-Instruktionen führen nicht viel aus. Aus diesem Grund brauchen wir viele von ihnen, um eine Aufgabe durchführen zu können. Aber jede einzelne Instruktion ist einfach und ein jeder kann sie verstehen, wenn er oder sie die Geduld dazu hat. Einige Programmierer, die ihre Karriere mit Maschinensprache begannen, fanden Sprachen der höheren Ebene, wie zum Beispiel BASIC, ziemlich schwierig. Maschinensprache-Instruktionen sind für sie einfach und präzise, während im Vergleich dazu BASIC Anweisungen vage und schwach definiert erscheinen.

Wohin soll Sie dieses Buch führen? Am Ende sollen Sie verstehen, was Maschinensprache ist, und Sie werden die Prinzipien, die man bei ihrer Programmierung benutzt, kennen. Sie werden kein Experte werden, aber Sie werden einen guten Start haben und nicht länger durch diese scheinbar mysteriöse Sprache frustriert werden.

Werden Sie ihre erlernten Fertigkeiten auch auf andere Maschinen anwenden können? Sicherlich. Wenn Sie erst einmal die Prinzipien der Programmierung verstanden haben, werden Sie auch in der Lage sein, die Programme zu adaptieren. Wenn Sie zu einem Nicht-Commodore Computer wechseln, der einen 6502-Chip benutzt (wie zum Beispiel der Apple oder Atari), werden Sie die Architektur dieser Maschinen und etwas über deren Maschinensprachemonitor lernen müssen. Diese werden zwar ein wenig verschieden sein, aber dieselben Prinzipien werden auf alle von ihnen anwendbar bleiben.

Selbst, wenn Sie zu einem Computer wechseln, der keinen Chip aus der 6502-Familie benutzt, werden Sie in der Lage sein, sich diesem anzupassen. Wenn Sie erst einmal durch die Instruktionen und durch die Bits einer Commodore-Maschine hindurchgestiegen sind, werden Sie die Prinzipien aller Binärcomputer erlernt haben. Sie werden zwar notwendigerweise die neuen Mikroprozessor-Instruktionen lernen müssen, aber es wird Ihnen bei diesem zweiten Mal wesentlich leichter fallen.

Müssen Sie ein Experte in BASIC sein, bevor Sie sich mit der Maschinensprache beschäftigen? Nicht unbedingt. Dieses Buch geht davon aus, daß Sie einige Grundbegriffe der Programmierung kennen: Schleifen, Verzweigungen, Unterroutinen und Entscheidungen. Aber, Sie müssen kein fortgeschrittener Programmierer sein, um die Maschinensprache zu erlernen.

Inhalt

1. Grundbegriffe

Dieses Kapitel behandelt:

- Die innere Arbeitsweise von Mikrocomputern
- Computernotationen: Binär und Hexadezimal
- Der innere Aufbau der 650x Prozessoren
- Erste Benutzung des Maschinensprachemonitors
- Die Speicheranordnung eines Computers
- Erste Maschinensprachebefehle
- Schreiben und Lauf eines einfachen Programms

Die innere Arbeitsweise von Mikrocomputern

Alle Computer enthalten eine große Zahl elektrischer Schaltkreise. In jedem Binärcomputer können diese Schaltkreise sich in zwei Zuständen befinden: „ein" oder „aus".

Techniker werden Ihnen sagen, daß „ein" gewöhnlich bedeutet, daß der betreffende Schaltkreis die volle Spannung durchschaltet und daß „aus" bedeutet, der Schaltkreis schaltet keine Spannung durch. Bei einem Digitalcomputer besteht für irgendeine Form von Zwischenwerten kein Bedarf. Jeder Schaltkreis ist entweder voll an- oder voll ausgeschaltet.

Das Wort „binär" bedeutet „auf 2 basierend" und alles, was innerhalb eines Computers passiert, basiert auf diesen zwei Möglichkeiten eines Schaltkreises: an oder aus. Wir können jede dieser beiden Zustände auf folgende unterschiedliche Weise identifizieren:

AN oder AUS
WAHR oder FALSCH
JA oder NEIN
1 oder 0

Die letzte Beschreibung, 1 oder 0, ist sehr brauchbar. Sie ist kurz und numerisch. Angenommen, wir hätten innerhalb eines Computers eine Gruppe von acht Schaltkreisen, einige von ihnen wären „an", andere „aus", so könnten wir ihre Zustände mit folgendem Ausdruck beschreiben:

```
11000111
```

Das würde bedeuten, daß die beiden linken Leitungen an wären, die nächsten drei aus und die letzten drei an. Der Wert 11000111 sieht wie eine Zahl aus; richtig, es handelt sich um eine Binärzahl, die aus den Ziffern 0 oder 1 besteht. Diese Zahl sollte nicht mit dem äquivalenten

Dezimalwert von etwas über 11 Millionen verwechselt werden. Die einzelnen Ziffern würden zwar gleich aussehen, aber in dezimaler Schreibweise könnte jede Stelle einen Wert zwischen 0 und 9 annehmen. Um nun eine Verwechslung mit Dezimalzahlen zu vermeiden, geht den Binärzahlen oft ein Prozentzeichen voraus, sodaß die Zahl so aussehen würde: %11000111.

Jede Stelle einer Binärzahl heißt ein Bit, die Abkürzung für „binary digit". Die oben gezeigte Zahl besteht aus acht Bits; eine Gruppe von acht Bit ist ein Byte. Bits werden oft von rechts nach links numeriert, beginnend mit Null. Das äußerst rechte Bit der obigen Zahl heißt demnach „Bit 0" und das äußerst links liegende Bit heißt „Bit 7". Das erscheint zwar ungewöhnlich, aber für diese Art der Numerierung gibt es eine mathematische Begründung.

Der Bus

Gewöhnlich wird eine Gruppe von Schaltkreisen gemeinsam genutzt. Die Leitungen verlaufen von einem Mikrochip zum anderen und dann zum nächsten. Dort, wo eine Gruppe von Leitungsverbindungen gemeinsam genutzt wird, indem diese unterschiedliche Punkte miteinander verbindet, wird diese Gruppe ein Bus genannt (manchmal auch „buss" geschrieben).

Der PET, CBM und VIC-20 benutzen einen 6502 Mikroprozessor. Der Commodore 64 benutzt einen 6510, der Commodore der B-Serie einen 6509 und der Commodore PLUS/4 einen 7501. Alle diese Chips sind ähnlich. Außerdem gibt es in dieser Prozessor-Familie einen Chip 6504. Jeder arbeitet nach dem gleichen Prinzip und wir können sie alle dem Familiennamen 650x zuordnen.

Wenden wir uns dem Beispiel eines Bus zu, der irgendeinen 650x benutzt. Ein 650x Chip hat nur wenig Speicherplatz eingebaut. Um eine Programmanweisung zu erhalten oder um eine Berechnung auszuführen, muß der 650x zuerst vom Speicher außerhalb eine Information abrufen, Daten also, die in einem anderen Chip gespeichert sind.

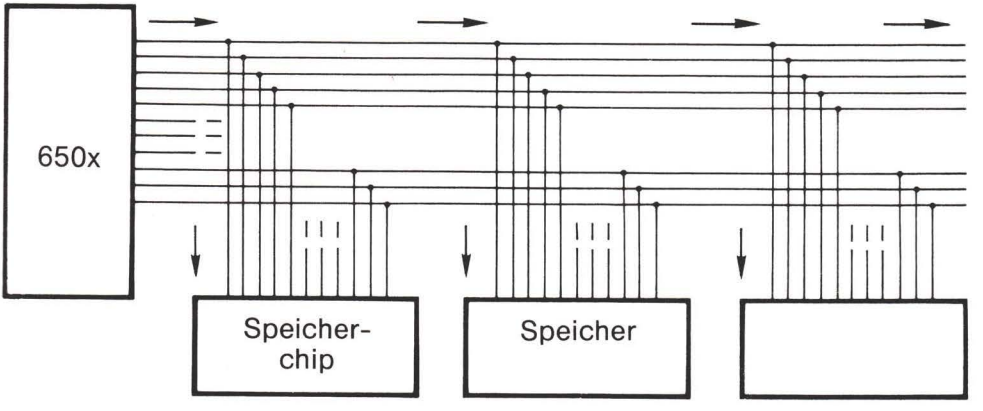


Abbildung 1.1: Der Adressbus verbindet einen 650x und 3 Chips.

Der 650x sendet dazu einen Anruf an alle Speicherchips und fragt nach Information, indem er Spannungen über eine Gruppe von sechzehn Drahtverbindungen, die Adressbus genannt wird, aussendet. Jede dieser sechzehn Verbindungen kann entweder eine Spannung tragen oder keine Spannung tragen. Diese Kombination von Signalen wird eine Adresse genannt.

Jeder Speicherbaustein ist an diesen Adressbus angeschlossen. Jeder dieser Bausteine liest diese Adresse, die aus der Spannungskombination besteht, die vom Prozessor ausgesandt wurde. Einer und nur ein einziger Baustein antwortet: „Das bin ich!" Mit anderen Worten, die spezifische Adresse wählt diesen Baustein aus. Das ist die Vorbereitung zur Kommunikation mit dem 650x. Alle anderen Bausteine hingegen antworten: „Das bin ich nicht!". Diese nehmen dann an dem folgenden Datentransfer nicht teil.

Der Datenbus

Nachdem der 650x Mikroprozessor eine Adresse über den Adressbus gesandt hat und diese Adresse von einem Speicherchip erkannt worden ist, können Daten zwischen dem Speicher und dem 650x fließen. Diese Daten bestehen aus acht Bits (sie fließen über acht Leitungen). Sie könnten folgendermaßen aussehen:

```
01011011
```

Diese Daten können in zwei Richtungen fließen. Das heißt, der 650x könnte einmal aus einem Speicherbaustein lesen. In diesem Fall schickt ein Speicherbaustein, der ausgewählt wurde, Informationen über den Datenbus und diese werden vom Mikroprozessor gelesen. Im zweiten Fall könnte der 650x Daten zum Speicherchip schreiben. Dann sendet der 650x Informationen über den Datenbus und der ausgewählte Speicherchip erhält die Daten und speichert diese.

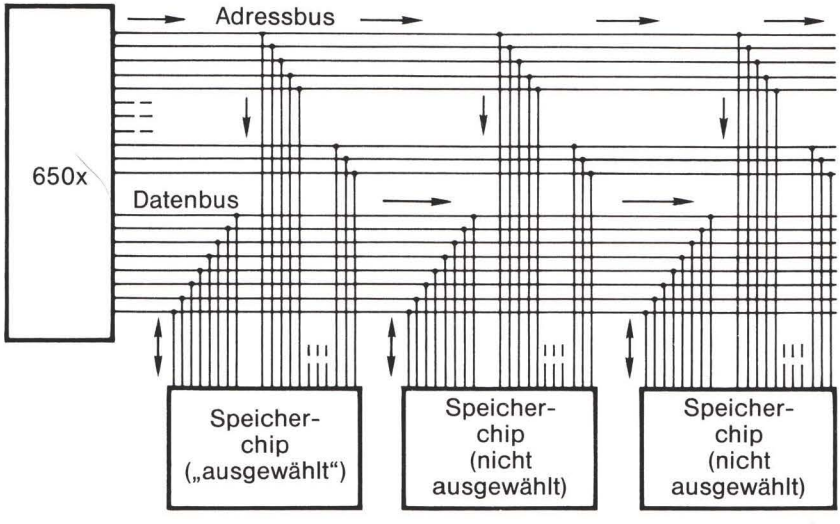


Abbildung 1.2: Zweiweg-Datenbus

Alle anderen Bausteine sind natürlich weiterhin mit dem Datenbus verbunden, aber sie ignorieren die Information, da sie nicht ausgewählt wurden.
Neben dem Adressbus gibt es noch einige weitere Leitungsverbindungen (manchmal Kontrollbus genannt). Diese überwachen den richtigen Zeitablauf und die Richtung, in der die Daten fließen sollen: Lesen oder Schreiben.

Zahlenumfang

Der Adressbus besteht aus sechzehn Bits, jedes davon kann an oder aus sein. Die Zahl der daraus resultierenden Kombinationen beträgt 65536 (zwei hoch sechzehn). Uns stehen also 65536 verschiedene Spannungskombinationen zur Verfügung oder 65536 verschiedene Adressen.

Der Datenbus hat nur acht Bits, die 256 verschiedene Spannungskombinationen erlauben. Jede Speicherstelle kann demnach nur 256 verschiedene Werte speichern.

Oft ist es üblich, eine Adresse in ihrer Dezimalschreibweise darzustellen. Das gilt besonders für die Anweisungen PEEK oder POKE in der Sprache BASIC. Wir können dadurch in dezimal umwandeln, daß wir jedem Bit eine „Gewichtung“ geben. Bit 0 (ganz rechts) bekommt das Gewicht 1, jedes weiter links folgende Bit hat eine Gewichtung, die dem doppelten des vorhergehenden entspricht, sodaß Bit 15 (ganz links) eine Gewichtung von 32768 erhält. Auf diese Weise hat eine Binäradresse wie

0001001010101100

einen Wert von $4096+512+128+32+8+4$ oder 4780. Die Anweisung POKE nach dezimal 4780 würde die obige Binäradresse benutzen, um den richtigen Speicherteil anzusprechen.

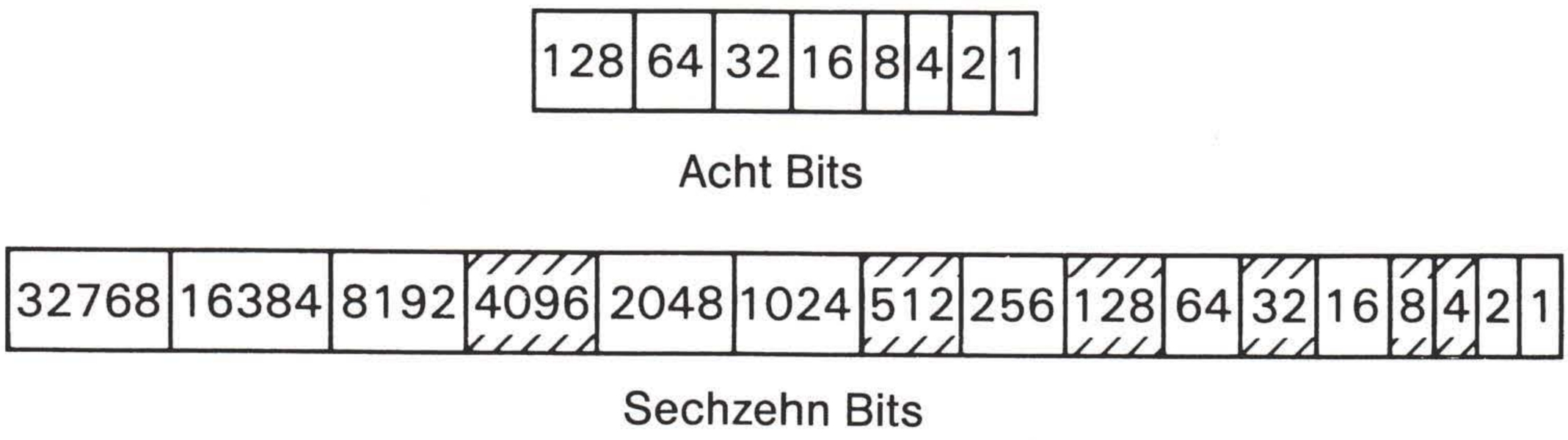


Abbildung 1.3

Eine direkte Umwandlung zwischen dezimal und binär wird selten benötigt. Solche Umwandlungen werden gewöhnlich unter Nutzung eines intermediären Zahlensystems vorgenommen, dem hexadezimalen System.

Hexadezimale Notation

Das binäre System ist zwar für den Computer ausgezeichnet geeignet, aber unhandlich für die meisten Programmierer. Wenn ein Programmierer einen anderen fragen würde: „Welche Adresse soll ich für eine bestimmte Operation benutzen?", dann würde eine Antwort wie: „Adresse %0001001010101100" zwar richtig jedoch sehr unpraktisch sein. Sie enthält nämlich zuviele Stellen.

Der Hexadezimalcode wird gewöhnlich vom Menschen benutzt, um Binärzahlen angemessen darzustellen. Der Computer hingegen benutzt Binär- nicht Hexadezimalzahlen. Programmierer benutzen hexadezimal, weil binär mühsam ist.

Um eine Binärzahl hexadezimal darzustellen, ordnet man die Bits am besten in Gruppen zu vier an. Tun wir das mit obiger Binärzahl, so erhalten wir

```
0001 0010 1010 1100
```

Nun wird jede Gruppe, bestehend aus vier Bit, durch eine Zahl repräsentiert, die wir der folgenden Tabelle entnehmen:

```
0000-0  0100-4  1000-8  1100-C
0001-1  0101-5  1001-9  1101-D
0010-2  0110-6  1010-A  1110-E
0011-3  0111-7  1011-B  1111-F
```

Auf diese Weise würde die Zahl durch den Hexadezimalwert 12AC repräsentiert. Ein Dollarzeichen wird oft vorangestellt, um die Hexadezimalzahl als solche erkennbar zu machen: $12AC.

Dieselbe Art von Gewichtung wird auf jedes Bit der Vierergruppe, wie eben beschrieben, angewandt. Mit anderen Worten, das äußerst rechte Bit (Bit 0) hat eine Gewichtung von 1. Das nächst links folgende eine von 2, dann das nächste eine von 4 und das ganz links liegende (Bit 3) eine von 8. Wenn die Summe der gewichteten Bits die Zahl 9 überschreitet, wird als Zahlenrepräsentant ein Buchstabe benutzt: A für zehn, B für elf, C ist zwölf und F steht für fünfzehn.

Acht-Bit Zahlen werden damit durch zwei Hexadezimalzahlen dargestellt:

%01011011 kann man schreiben als $5B.

Hexadezimal nach Dezimal

Wie wir gesehen haben, kann man Hexadezimal- und Binärzahlen leicht ineinander umwandeln. Obwohl wir gewöhnlich Werte hexadezimal schreiben werden, sind wir manchmal gezwungen, ihren wahren Binärzustand zu untersuchen, um eine bestimmte Bitinformation zu erhalten.

Es ist nicht schwer, hexadezimal in dezimal zu übersetzen. Sie werden sich aus dem Schulunterricht erinnern, daß die Zahl 24 bedeutet, „zwei Zehner und vier Einer". Auf gleiche Weise bedeutet hexadezimal 24, „zwei Sechzehner und vier Einer", oder ein Dezimalwert von 36. Nebenbei, es ist besser, Hex-zahlen als „zwei vier" nicht als „vierundzwanzig" auszusprechen, um Verwechslungen mit Dezimalzahlen zu vermeiden.

Die formale Prozedur, auch Algorithmus genannt, um von Hex nach Dezimal umzuwandeln, verläuft folgendermaßen.

Schritt 1: Man nehme die äußerst linke Stelle; wenn es ein Buchstabe A bis F ist, wandle man ihn in den entsprechenden Zahlenwert um (A entspricht 10, B entspricht 11 usw.).
Schritt 2: Wenn keine Stellen mehr übrig sind, ist man fertig; man hat die Zahl. STOP
Schritt 3: man multipliziere den erhaltenen Wert mit sechzehn. Addiere die nächste Stelle zu diesem Ergebnis unter eventueller Umwandlung der Buchstaben. Man gehe zurück zu Schritt 2.

Wir wollen nun die Hexadezimalzahl $12AC umwandeln, indem wir die obigen Schritte ausführen.

Schritt 1: Die äußerst linke Stelle ist eine 1.
Schritt 2: Es gibt noch mehr Stellen, also müssen wir fortfahren.
Schritt 3: 1 mal 16 ist 16, plus 2 ergibt 18.
Schritt 2: Es gibt weitere Stellen.
Schritt 3: 18 mal 16 ist 288, plus 10 (für A) ist 298.
Schritt 2: Es gibt mehr Stellen.
Schritt 3: 298 mal 16 ist 4768, plus 12 (für C) ist 4780.
Schritt 2: keine weiteren Stellen: 4780 ist der Dezimalwert.

Das läßt sich leicht von Hand oder mit einem Taschenrechner ausführen.

Dezimal nach Hexadezimal

Die einfachste Methode, um von Dezimal nach Hexadezimal umzuwandeln, besteht in einer wiederholten Teilung durch 16; nach jeder Teilung stellt der Rest die nächste Hexadezimalstelle dar, wobei wir von rechts nach links vorgehen. Diese Methode ist allerdings für einfache Taschenrechner nicht sehr geeignet, da sie keinen Rest anzeigen. Dafür bietet die folgende Tabelle Hilfestellung:

```
,0000-0   ,2500-4   ,5000-8   ,7500-C
,0625-1   ,3125-5   ,5625-9   ,8125-D
,1250-2   ,3750.6   ,6250-A   ,8750-E
,1875-3   ,4375-7   ,6875-B   ,9375-F
```

Wenn wir nach dieser Methode 4780 umwandeln, teilen wir durch 16 und erhalten 298,75. Der Rest (,75) ergibt C als letzte Stelle; wir teilen nun 298 durch 16 und erhalten 18,625. Der Rest entspricht A. Damit heißen die letzten beiden Stellen AC. Jetzt teilen wir 18 durch 16 und erhalten 1,125 - die letzten drei Stellen lauten 2AC. Wir brauchen die 1 nicht mehr durch 16 zu teilen, obwohl das auch funktionieren würde; wir stellen sie der Zahl einfach voran und erhalten als Endergebnis $12AC.

Es gibt noch andere Methoden zur Umwandlung von Dezimal nach Hexadezimal. Diese kann man in einem Buch über Zahlensysteme nachlesen. Man kann auch einen speziellen Taschenrechner kaufen, der einem diese Arbeit abnimmt. Einige Programmierer haben soviel Erfahrung, daß sie die Umwandlung im Kopf vornehmen können. Ich nenne sie „Hex-Knakker".

Man sollte sich jedoch nicht zu sehr auf bestimmte Zahlennotationen festlegen. Speicheradressen können immer auch als Binärzahl dargestellt sein und müssen dann in dezimal oder

hexadezimal umgewandelt werden. Diese müssen nicht immer eine numerische Bedeutung haben: die Speicherzelle kann nämlich entweder ein ASCII-codiertes Zeichen, eine Instruktion oder irgendetwas anderes enthalten.

Speicherelemente

Man unterscheidet im allgemeinen drei verschiedene Bausteintypen, die an den Speicherbus (Adress-, Daten- und Kontrollbus) angeschlossen sind:

- RAM (Random access memory): Das ist der Lese- und Schreibspeicher, in dem wir geschriebene Programme zusammen mit Daten, die von dem Programm benutzt werden, speichern. Im RAM können wir Informationen speichern und jederzeit wieder abrufen.
- ROM (Read only memory): Hierin sind bestimmte feste Routinen für den Computer abgespeichert. Wir können Informationen nicht in das ROM speichern; der Inhalt des ROM wird bei seiner Herstellung festgelegt. Wir werden bestimmte Programmeinheiten (Unterroutinen), die im ROM gespeichert sind, für spezielle Aufgaben wie beispielsweise Eingabe und Ausgabe nutzen.
- IA (Interface adaptor): Hierbei handelt es sich nicht um Speicherbausteine im üblichen Sinn. Diese Chips sind jedoch bestimmten Adressen des Speicherbus zugeordnet, weshalb wir sie "speicherorientierte" Einheiten nennen („memory-mapped"). Information kann zu oder von diesen Einheiten fließen, aber diese Information wird im allgemeinen nicht im konventionellen Sinn gespeichert. IA-chips enthalten Funktionen wie: Eingabe/Ausgabe (I/O = input/output), die als Verbindung zur „Außenwelt" dienen: Zeitgebereinheiten, Unterbrechungskontrollsysteme und manchmal spezialisierte Funktionen wie Bildschirmkontrolle oder Tonerzeugung. Es gibt eine Vielfalt von IA-chips: PIA (peripheral interface adaptor), VIA (versatile interface adaptor), CIA (complex interface adaptor), VIC (video interface chip) und SID (sound interface device).

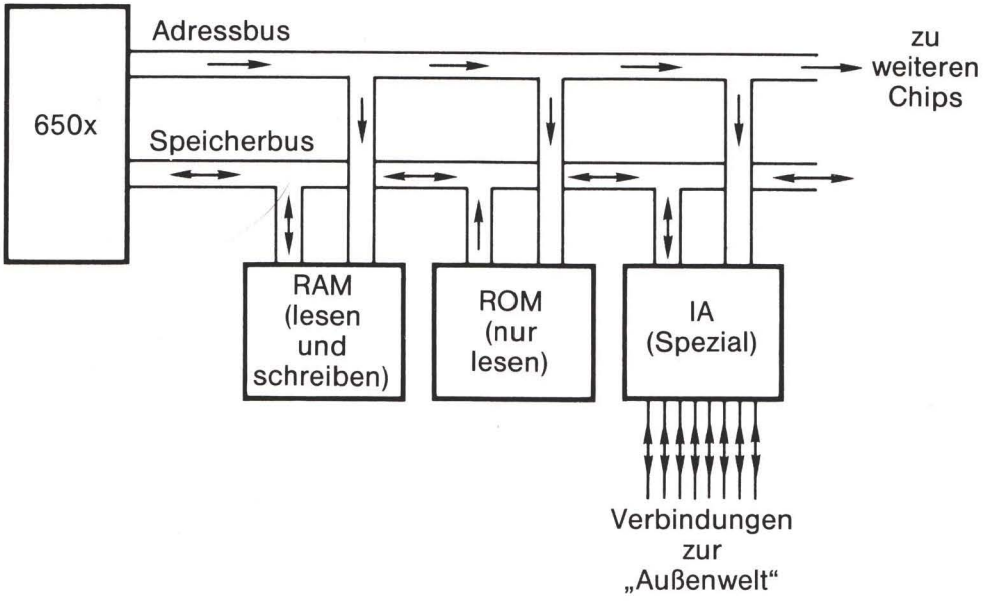


Abbildung 1.4

In einem vorgegebenen Computer können einige Adressen garnicht benutzt sein. Einige Einheiten können auch auf mehr als eine Adresse antworten, als ob sie auf zwei Speicherstellen zu stehen scheinen.

Eine Adresse kann man sich in zwei Teile aufgespaltet vorstellen. Ein Teil, gewöhnlich der obere Teil der Adresse, wählt einen bestimmten Chip aus. Der andere Teil der Adresse wählt einen bestimmten Teil des Speichers innerhalb des Chips aus. Zum Beispiel setzt im Commodore 64 die hex-Adresse $D020 (dezimal 53280) die Randfarbe des Bildschirms. Der erste Teil der Adresse ($D0..) wählt den Videochip, der letzte Teil der Adresse (..20) wählt den Teil des Chips aus, der die Randfarbe bestimmt.

Mikroprozessor Register

Innerhalb des 650x gibt es einige Speicherbereiche, die Register genannt werden. Obwohl diese Informationen speichern, werden sie nicht wie der Speicher behandelt, da ihnen keine Adresse zugeordnet ist. Sechs dieser Register sind für uns wichtig:

PC: (16 Bits)	Der Befehlszähler. Er zeigt auf die Adresse, die den nächsten Befehl enthält.
A, X und Y (jeweils 8 Bits)	Diese Register enthalten Daten.
SR	Das Statusregister, manchmal PSW (processor status word) genannt, enthält das Resultat vorhergegangener Tests, Datenbehandlung usw.
SP	Der Stapelzeiger führt Buch über einen temporären Speicherbereich.

Wir werden jedes dieser Register später im Detail besprechen. Im Augenblick wollen wir uns mit dem PC (programm counter) Register beschäftigen.

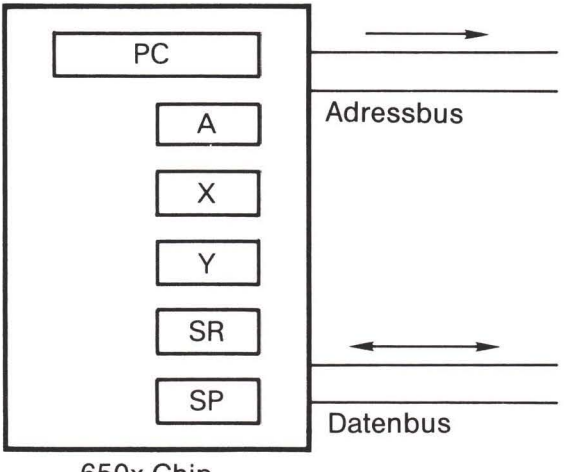


Abbildung 1.5

Befehlsausführung

Angenommen, der 650x würde angehalten (kein einfacher Trick) und das Register PC enthalte eine bestimmte Adresse: $1234. In dem Augenblick, in dem wir den Mikrocomputer wieder starten, wird diese Adresse als zu lesende Adresse auf dem Adressbus ausgegeben und der Prozessor wird zu dem Inhalt des PC (Befehlszähler) eine 1 addieren.

Das heißt, der Inhalt der Adresse $1234 wird aufgerufen und der PC wird sich auf $1235 erhöhen. Welche Information auch immer dadurch auf den Datenbus gelangt, diese wird als Befehl behandelt.

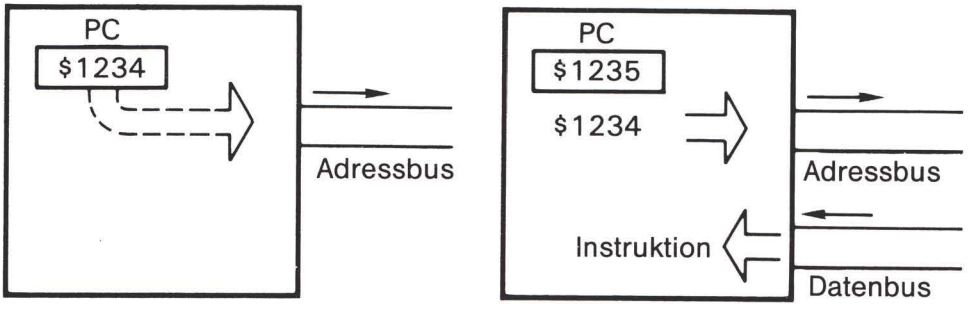


Abbildung 1.6: Der Pfeil weist auf den Adressbus

Der Mikroprozessor hat nun den Befehl, der ihn anweist, etwas auszuführen. Diese Anweisung wird ausgeführt und der gesamte Vorgang wiederholt sich für den nächsten Befehl. Mit anderen Worten, Adresse $1235 wird jetzt an den Speicher gesandt und PC erhöht sich auf $1236. Man erkennt leicht, daß der Prozessor auf die gleiche Weise wie die meisten Computersprachen arbeitet: Ein Befehl wird ausgeführt und der Computer geht zum nächsten Befehl über und wieder zum nächsten und so weiter. Wir können zwar die Reihenfolge der Befehle durch „Sprünge" oder „Verzweigungen" auf neue Speicherstellen anders durchlaufen, aber normalerweise folgt ein Befehl dem nächsten.

Datenregister: A, X und Y

Jedes dieser drei Register kann dazu benutzt werden, um 8-Bit Daten sowohl zu speichern als auch zu manipulieren. Wir können Informationen aus dem Speicher in A, X oder Y laden und wir können Informationen aus A, X oder Y in den Speicher speichern.

Sowohl beim „Laden" wie beim „Speichern" handelt es sich um Kopiervorgänge. Wenn ich A aus der Adresse $2345 lade (LDA), stelle ich eine Kopie des Inhaltes von hex 2345 in A her, aber $2345 enthält danach immer noch seinen alten Wert. Genauso, wenn ich Y in $3456 speichere, stelle ich ein Kopie des Inhalts von Y in dieser Adresse her: Y verändert sich nicht.

Der 650x bietet keine Möglichkeit, Informationen auf direktem Weg von einer Speicheradresse zu einer anderen zu übertragen. Das bedeutet, daß die Information immer die Register A, X oder Y durchlaufen muß; wir laden sie aus der alten Adresse und speichern sie in die neue Adresse.

Später werden wir sehen, daß die drei Register noch individuelle Eigenschaften besitzen. Zum Beispiel wird das Register A manchmal auch Akkumulator genannt, weil wir darin Addition und Subtraktion ausführen. Im Augenblick sind sie für uns noch austauschbar: wir können in jedes der drei laden und aus jedem von ihnen speichern.

Erstes Programmvorhaben

Hier ist eine Programmieraufgabe: die Adressen $0380 und $0381 enthalten Information. Wir wollen ein Programm schreiben, das die Inhalte der beiden Adressen austauscht. Wie machen wir das?

Wir müssen zunächst einen Plan entwerfen. Wir wissen, daß wir nicht direkt Information von Speicher zu Speicher bewegen können. Wir müssen in ein Register laden und dann speichern. Es gibt noch mehr zu bedenken. Wir dürfen Daten im Speicher nicht speichern und dadurch bestehende Daten zerstören, bis wir die Daten sicher beiseite gelegt haben. Wie erreichen wir das?

Hier ist unser Plan. Wir laden einen Wert in A (z.B. den Inhalt von $0380) und laden den anderen Wert in X (den Inhalt von $0381). Dann speichern wir A und X auf umgekehrtem Weg zurück.

Wir hätten auch ein anderes Registerpaar für unseren Plan wählen können: A und Y, oder X und Y. Bleiben wir aber bei dem ursprünglichen Plan. Diesen können wir in einer formaleren Weise codieren:

```
LDA $0380   (lade ersten Wert)
LDX $0381   (lade zweiten Wert)
STA $0381   (speichere in gegenüberliegenden Platz)
STX $0380   (noch einmal)
```

Sie werden feststellen, daß wir „lade A“ (load A) als LDA codiert haben, „lade X“ als LDX, „speichere A“ (store A) als STA und „speichere X“ als STX. Jedes Kommando hat eine Standardabkürzung aus drei Buchstaben, Mnemonic genannt. Wenn wir das Y-Register benutzt hätten, hätten wir LDY und STY codieren müssen.

Wir brauchen noch ein weiteres Kommando. Wir müssen nämlich dem Computer sagen, daß er anhalten soll, wenn er die vier Befehle beendet hat. In Wirklichkeit können wir den Computer nicht anhalten; wir benutzen jedoch das Kommando BRK (break), wodurch der Computer zum Maschinensprachemonitor (MLM = machine language monitor) zurückkehrt und auf weitere Befehle wartet. Wir werden gleich über den MLM sprechen.

Wir haben unser Programm in einer Notation geschrieben, die vom Menschen leichter verstanden wird, in Assemblersprache. Der Computer versteht jedoch diese Notation nicht. Wir müssen sie deshalb in Maschinensprache übersetzen.
Der Binärcode für LDA ist %10101101, oder hexadezimal AD. Das kann der Computer verstehen. Das ist der Befehl, den wir in den Speicher setzen müssen. Wir codieren also die erste Zeile:

```
AD 80 03    LDA $0380
```

Man ist übereingekommen, den Maschinencode auf die linke und den Quellcode auf die rechte Seite zu schreiben. Schauen wir uns näher an, was passiert ist.

LDA wurde in $AD übersetzt. Das ist der Befehlscode (op code = operation code), der sagt, was zu tun ist. Er benötigt ein Byte des Speichers. Nach dem Befehl müssen wir noch die Adresse angeben, aus der wir „laden" wollen. Das ist Adresse $0380; sie ist 16 Bits lang und wir brauchen deshalb zwei Bytes, um die Adresse zu speichern. Wir stellen die Adresse, auf die sich der Befehl bezieht (operand genannt), im Speicher direkt hinter den Befehl. Das ganze hat jedoch einen Dreh: das letzte Byte kommt zuerst, sodaß die Adresse $0380, als zwei Bytes, zuerst 80 und dann 03, gespeichert wird.

Diese Methode der Adressspeicherung - niedriges Byte zuerst - ist beim 650x ein Standard. Das scheint ungewöhnlich, aber es gibt einen guten Grund: der Computer gewinnt durch diese „Rückwärtsadresse" an Geschwindigkeit. Man muß sich daran gewöhnen, da es noch häufig vorkommen wird.

Hier sind einige Maschinensprache op codes von Befehlen, die wir benutzen wollen. Man muß sie nicht auswendig lernen.

```
LDA-AD    LDX-AE    LDY-AC    BRK-00
STA-8D    STX-8E    STY-8C
```

Jetzt können wir die Übersetzung unseres Programms vervollständigen.

```
AD 80 03    LDA $0380
AE 81 03    LDX $0381
8D 81 03    STA $0381
8E 80 03    STX $0380
00          BRK
```

Rechts steht unser Plan, links steht das eigentliche Programm, das in den Computer gespeichert werden wird. Wir wollen die rechte Seite Assemblercode und die linke Seite Maschinencode nennen, um zwischen beiden unterscheiden zu können. Einige Benutzer nennen die rechte Information den Quellcode, weil das der Beginn unseres Programmplans ist, und die linke den Objektcode, weil das das Objekt unserer Übung ist - einen Code in den Computer zu bringen. Die Übersetzung von Quellcode in Objektcode nennt man assemblieren. Wir führen diese Übersetzung dadurch aus, daß wir die op codes nachschauen und von Hand übersetzen. Das nennt man Handassemblieren.

Der Code muß nun in den Computer gebracht werden. Er besteht aus dreizehn Bytes: AD 80 03 AE 81 03 8D 81 03 8E 80 03 00. Das ist das ganze Programm. Aber nun taucht eine neue Frage auf: wohin sollen wir es speichern?

Platzwahl

Wir müssen einen geeigneten Platz für unser Programm finden. Natürlich gehört es in den RAM Speicher, aber wohin?

Im Augenblick wollen wir es in den Kassettenpuffer schreiben, beginnend bei Adresse $033C (dezimal 828). Das ist ein guter Platz für kurze Testprogramme, die wir für eine Weile schreiben wollen.

Nachdem wir diese Entscheidung getroffen haben, stehen wir erneut vor einer Hürde: wie bekommen wir das Programm hinein? Dazu brauchen wir einen Maschinensprachemonitor.

Monitor: Was ist das?

Alle Computer haben einen eingebauten Satz von Programmen, die man Betriebssystem (operating system) nennt. Dadurch erhält die Maschine gewisse Grundeigenschaften. Das Betriebssystem kümmert sich um die Kommunikation - Tastatur lesen, dafür sorgen, daß die richtigen Dinge auf dem Bildschirm erscheinen und Übertragung von Daten zwischen dem Computer und anderen Einheiten wie Disk, Band oder Drucker.

Wenn wir auf der Computertastatur tippen, benutzen wir das Betriebssystem, das die Zeichen, die wir tippen, erkennt. Es gibt aber noch einen Extrasatz von Programmen, die in den Computer eingebaut sind und entscheiden müssen, was wir meinen. Wenn wir die Sprache BASIC benutzen, kommunizieren wir mit dem BASIC-Monitor, der Kommandos wie etwa NEW, LOAD, LIST oder RUN versteht. Er enthält Editiermöglichkeiten, mit deren Hilfe wir BASIC Programme ändern können.

Wenn wir auf ein anderes System umschalten - meist eine andere Sprache - benötigen wir einen anderen Monitor. Befehle wie zum Beispiel NEW oder LIST haben für ein Maschinenspracheprogramm keinerlei Bedeutung. Wir müssen den BASIC-Monitor verlassen und eine neue Umgebung betreten: den Maschinensprachemonitor. Wir müssen dann einige neue Befehle lernen, weil wir mit dem Computer auf andere Weise kommunizieren.

Der Maschinensprachemonitor

Die meisten PET/CBM Computer haben einen einfachen Maschinensprachemonitor (MLM) eingebaut. Er läßt sich durch zusätzliche Befehle erweitern. Der Commodore PLUS/4 enthält einen sehr vielseitigen MLM. Der VIC-20 und der Commodore 64 haben keinen MLM fest eingebaut, aber man kann einen hinzufügen. Solch einen Monitor kann man entweder in das RAM laden oder als Kassette einstecken. Monitorprogramme kann man kaufen oder über Benutzerclubs beziehen.

Die meisten Maschinensprachemonitore funktionieren ähnlich und enthalten die gleichen Befehle. Um in unserem Programmvorhaben fortzufahren, brauchen wir einen MLM im Computer. Man kann den eingebauten benutzen, einen einstecken, einen laden oder einen laden und starten ..., was auch immer die Bedienungsanleitung verlangen mag. Auf einer PET/CBM Maschine schaltet man gewöhnlich auf den eingebauten Monitor um, indem man SYS 4 eintippt. Nachdem man einen MLM in einen VIC oder einen Commodore 64 geladen hat, startet man ihn gewöhnlich mit dem Befehl SYS 8. Auf dem Commodore PLUS/4 bringt das BASIC-Kommando MONITOR den Monitor ins Spiel.

Monitoranzeige

Nach Übergang zum MLM werden Sie eine Anzeige sehen, die etwa folgendermaßen aussieht:

```
B*
    PC  SR AC XR YR SP
.; 0005 20 54 23 6A F8
.
```

Der Cursor leuchtet rechts vom Punkt der untersten Zeile. Das genaue Aussehen der Bildschirminformation kann je nachdem, welchen Monitor man benutzt, unterschiedlich sein. Weitere Informationen können angezeigt werden (besonders ein Wert IRQ), die wir im Augenblick ignorieren wollen.

Die Information, die man sieht, läßt sich folgendermaßen interpretieren:

B* - Wir haben den MLM durch ein „break" Kommando erreicht. Mehr darüber später.

PC - Der Wert unter dieser Überschrift ist der Inhalt des Befehlszählers. Er zeigt, wo das Programm „stoppte". Mit anderen Worten, wenn der Wert die Adresse 0005 zeigt, stoppte das Programm bei der Adresse 0004, da der PC bereit ist, auf der folgenden Adresse fortzufahren. Der genaue Wert (0004 oder 0005) hängt vom jeweiligen MLM ab.

SR - Der Wert darunter zeigt das Statusregister und damit das Resultat eines vorangegangenen Tests oder einer Datenoperation an. Um hieraus eine genaue Information zu erhalten, müssen wir die acht Bits auseinandernehmen und einzeln betrachten. Wir tun das später.

AC, XR, YR - Die Werte, die unter diesen drei Überschriften stehen, sind die Inhalte der drei Datenregister: A, X, Y

SP - Der Wert darunter ist der Inhalt des Stapelzeigers, der einen temporären Speicherbereich anzeigt, den das Programm nutzen kann. Ein Wert von F8 zum Beispiel sagt uns, daß der nächste Inhalt, der im Stapelbereich abgelegt wird, in die Speicheradresse $01F8 geht. Mehr darüber später.

Der Punkt entspricht, grob gesagt, dem READY in BASIC. Er zeigt an, daß der Computer bereit ist, ein Kommando entgegenzunehmen.

Sie werden feststellen, daß die durch den Monitor ausgegebene Anzeige (die Registeranzeige) die internen Register des 650x Chip zeigt. Manchmal findet man in der Anzeige die Information IRQ. Diese gehört nicht dazu, da sie kein Mikroprozessorregister repräsentiert. IRQ sagt uns, zu welcher Adresse der Computer gehen wird, wenn eine Unterbrechung (interrupt) erfolgt. Diese Information ist im Speicher und nicht im 650x abgelegt.

MLM Befehle

Der Maschinensprachemonitor wartet auf die Eingabe eines Befehls. Die alten BASIC-Befehle funktionieren hier nicht mehr. LIST oder NEW oder SYS sind dem MLM unbekannt. Wir werden gleich einige geläufige Befehle nennen. Zuerst wollen wir über das Kommando sprechen, das uns ins BASIC zurückbringt.

Das .X beendet den MLM und bringt uns zum BASIC-Monitor zurück. Versuchen Sie es. Erinnern Sie sich, daß, nachdem Sie X getippt haben, die Taste RETURN betätigt werden muß. Sie werden damit zum BASIC System zurückkehren und der BASIC-Monitor schreibt READY. Sie sind in vertrauter Umgebung. Nun gehen Sie zum Monitor zurück mit SYS 4 oder SYS 8 oder MONITOR, je nach System. BASIC ignoriert Leerstellen: es macht keinen Unterschied, SYS8 oder SYS 8, Sie müssen nur die richtige Zahl für Ihre Maschine benutzen (4 für PET/CBM, 8 für VIC/64).

Achtung: BASIC Kommandos sind im MLM unbrauchbar und Maschinensprachekommandos (wie .X) sind in BASIC sinnlos. Am Anfang wird man die falschen Kommandos zur falschen Zeit geben, weil es einem noch schwerfällt, festzustellen, welcher Monitor gerade aktiv ist. Wenn Sie ein MLM Kommando eintippen, obwohl Sie gerade im BASIC sind, werden Sie als Antwort vielleicht ein ?SYNTAX ERROR erhalten. Wenn Sie ein BASIC Kommando in den Maschinensprachemonitor eintippen, erhalten Sie wahrscheinlich in derselben Zeile ein Fragezeichen.

Einige weitere MLM Befehle lauten folgendermaßen (der Antwort-Punkt ist eingeschlossen):

```
.M 1000 1010      (zeige Speicher von hex 1000 bis 1010)
.R                (zeige Register ... wieder!)
.G 033C           (gehe nach hex 033C und starte ein Programm)
```

Geben Sie das letzte Kommando (.G) nicht ein. Es gibt auf der Adresse 033C noch kein Programm. Der Computer würde sonst zufällige Anweisungen ausführen und wir würden die Kontrolle darüber verlieren.

Es gibt zwei weitere wichtige Befehle, die wir noch nicht benutzen wollen: .S für speichern und .L für laden. Sie sind recht trickreich. Benutzen Sie diese erst, nachdem Sie etwas über BASIC-Zeiger (Kapitel 6) gelernt haben.

Anzeige von Speicherinhalten

Sie werden festgestellt haben, daß es zwar einen Befehl zur Anzeige des Speicherinhalts gibt, aber einen Befehl zur Änderung des Speicherinhalts scheint es nicht zu geben. Aber selbstverständlich kann man beides durchführen.

Angenommen, wir verlangen die Anzeige des Speichers von $1000 bis $1010 mit dem Befehl

```
.M 1000 1010
```

Achten Sie darauf, daß vor jeder Adresse genau ein Leerzeichen steht. Sie erhalten eine Anzeige, die ungefähr so aussieht:

```
.:1000 11 3A E4 00 21 32 04 AA
.:1008 20 4A 49 4D 20 42 55 54
.:1010 54 45 52 46 49 45 4C 44
```

Die vierstellige Zahl am Anfang jeder Zeile stellt die Adresse des Speichers dar, der angezeigt wird. Die zweistelligen Zahlen rechts davon zeigen den Inhalt des Speichers. Denken Sie daran, daß alle Zahlen, die der MLM benutzt, hex Zahlen sind.

Im obigen Beispiel enthält $1000 einen Wert von $11, $1001 einen Wert von $3A usw. bis $1007, die einen Wert von $AA enthält. Wir fahren mit Adresse $1008 auf der nächsten Zeile fort. Die meisten Monitore zeigen acht Speicherstellen pro Zeile, einige VIC-20 Monitore zeigen nur fünf wegen des engen Bildschirms.

Wir verlangten die Darstellung bis zur Adresse $1010, aber wir erhalten in diesem Fall den Inhalt bis $1017. Der Monitor füllt immer eine Zeile, selbst wenn wir die Extrainformation nicht verlangen.

Ändern von Speicherinhalten

Nachdem wir den Inhalt eines Teils des Speichers angezeigt haben, können wir diesen Teil des Speichers leicht verändern. Alles, was wir machen müssen, ist, den Cursor über den Speicherinhalt zu bewegen, den wir verändern wollen, den neuen Wert darüberzutippen und RETURN zu drücken.

Genau wie bei der Änderung eines BASIC Programms. Man überschreibt auf dem Bildschirm und nach Drücken von RETURN ersetzt die neue Zeile die alte. Diese allgemeine Technik wird Bildschirmeditierung (screen editing) genannt.

Wenn Sie den Inhalt des Speichers wie in dem obigen Beispiel angezeigt haben, möchten Sie vielleicht einige Bereiche zu Null verändern. Vergessen Sie nicht die RETURN Taste, damit sich die Veränderung auf dem Bildschirm auf den Speicher auswirkt. Geben Sie einen weiteren .M Befehl, um sicherzustellen, daß sich der Speicherinhalt wirklich verändert hat.

Ändern der Register

Wir wollen auch die Inhalte der Register durch Überschreiben und Eingabe von RETURN verändern. Sie können eine Registeranzeige durch den Befehl .R erhalten und dann die Inhalte von PC, AC, XR und YR verändern. Lassen Sie die Inhalte von SR und SP unverändert - wenn Sie mit diesen beiden herumexperimentieren, könnten seltsame und unerwartete Dinge passieren.

Eingabe des Programms

Wir müssen unser Programm ein letztes Mal umschreiben, um die Adressen für jeden Befehl einzutragen. Sie werden sich erinnern, daß wir beschlossen haben, das Programm vom Beginn der Adresse $033C (Teil des Kassettenpuffer) zu speichern.

```
033C AD 80 03   LDA $0380
003F AE 81 03   LDX $0381
0342 8D 81 03   STA $0381
0345 8E 80 03   STX $0380
0348 00
```

Erinnern Sie sich daran, daß die Hauptveränderung an der obigen Darstellung mehr kosmetischer Natur ist. Der eigentliche Arbeitsteil des Programms besteht aus dem Satz zweistelliger hex Zahlen auf der linken Seite. Auf der äußerst linken Seite finden wir die Adressen, das ist auch Information, aber nicht das Programm selbst. Auf der rechten Seite haben wir den „Quellcode", unsere Notizen darüber, was dieses Programm bedeutet.

Wie bekommen wir das Programm hinein? Einfach. Wir müssen den Speicher verändern. Deshalb gehen wir zum MLM und zeigen den Speicher an mit

```
.M 003C 0348
```

Wir finden wahrscheinlich in diesem Teil des Speichers irgendetwas. Auf jeden Fall werden wir eine Anzeige erhalten, die folgendermaßen aussieht

```
.:033C xx xx xx xx xx xx xx xx
.:0344 xx xx xx xx xx xx xx xx
```

Sie werden natürlich keine „xx" sehen. Für jede Speicherstelle wird irgendein hex Wert angezeigt sein. Wir wollen nun den Cursor zurückbewegen und die Anzeige verändern, damit sie wie folgt aussieht

```
.:033C AD 80 03 AE 81 03 8D 81
.:0344 03 8E 80 03 00 xx xx xx
```

Tippen Sie die „xx" nicht ein - lassen Sie einfach die alten Werte dort stehen. Vergewissern Sie sich, daß Sie RETURN eingegeben haben, um jede Zeile zu aktivieren. Wenn Sie den Cursor nach unten bewegen, um in die nächste Zeile zu gelangen, ohne RETURN zu tippen, findet keine Speicherveränderung statt.

Zeigen Sie erneut den Speicherinhalt an (.M 033C 0348), um sicher zu gehen, daß das Programm korrekt in den Speicher eingegeben wurde. Vergleichen Sie die Speicheranzeige mit dem Programmlisting und vergewissern Sie sich, daß Sie verstanden haben, wie das Programm in den Speicher überschrieben wurde.

Wenn alles gut aussieht, sind Sie bereit, Ihr erstes Maschinenspracheprogramm laufen zu lassen.

Vorbereitung

Nun fehlt uns noch eine weitere Sache. Wenn wir den Inhalt der Adressen $0380 und $0381 vertauschen wollen, sollten wir besser irgendetwas in diese beiden Stellen laden, damit wir später wissen, ob die Vertauschung korrekt stattgefunden hat.

Zeigen Sie den Speicher mit .M 0380 0381 und verändern Sie die entsprechende Anzeige, sodaß die Werte so aussehen

```
.:0380 11 99 xx xx xx xx xx xx
```

Denken Sie an RETURN. Nun können wir unser Programm laufen lassen. Wir starten es mit

```
.G 033C
```

Das Programm läuft so schnell ab, daß es sofort fertig zu sein scheint (die Laufzeit beträgt weniger als eine fünfzigtausendstel Sekunde). Der letzte Befehl in unserem Programm lautete BRK (break) für „Beenden" und das bringt uns direkt zum MLM mit der Anzeige von *B (für Beenden) und der Anzeige aller Register.

Nichts scheint sich verändert zu haben? Warten Sie ab. Schauen Sie sich genau die Registeranzeige an. Können Sie sich die Werte erklären, die Sie in den Registern AC und XR sehen? Können Sie den Wert von PC erklären?

Jetzt sollten Sie die Datenwerte anzeigen, die wir verändern wollten. Geben Sie das Speicheranzeigekommando .M 0380 0381 ein. Haben sich die Speicherinhalte dieser beiden Stellen verändert?

Das wäre gut so, denn dazu haben wir ja unser Programm geschrieben.

Was wir gelernt haben

- Computer benutzen Binärzahlen. Wenn wir mit den inneren Mechanismen eines Computers arbeiten wollen, müssen wir zwangsläufig mit Ausdrücken umgehen, die Binärwerte darstellen.
- Hexadezimal Notation ist für Menschen, nicht für Computer. Für den Menschen läßt sich damit einfacher umgehen als mit Binärzahlen.
- Der 650x Mikroprozessor Chip verkehrt mit dem Speicher unter Aussendung von Adressen über seinen Speicherbus.
- Der 650x besitzt interne Arbeitsbereiche, Register genannt.
- Der Befehlszähler nennt uns die Adresse, von der der Prozessor seine nächste Instruktion erhalten wird.
- Drei Register, A, X und Y, werden benutzt, um die Daten zwischenzuspeichern und zu manipulieren. Diese können vom Speicher geladen oder in diesen abgespeichert werden.
- Adressen, die bei 650x Befehlen benutzt werden, sind vertauscht dargestellt: das niedrige Byte kommt zuerst, gefolgt von dem hohen Byte.
- Der Maschinensprachemonitor eröffnet uns eine neue Kommunikationsmöglichkeit mit dem Computer. Unter anderem erlaubt er uns, den Speicher in hexadezimaler Notation zu inspizieren und zu verändern.

Details: Programmausführung

Wenn wir .G 033C eingeben, um unser Programm zu starten, durchläuft der Mikroprozessor folgende Schritte:

1. Er fragt nach dem Inhalt von $033C. Er erhält $AD, was er als op code „lade A" erkennt. Er weiß, daß er eine Adresse von zwei Byte Länge benötigt, um diese Instruktion auszuführen.
2. Er fragt nach dem Inhalt von $033D und dann $033E. Wenn er die Werte $80 und $03 erhalten hat, fügt er diese zu einer „Befehlsadresse" zusammen.

3. Der Mikroprozessor hat jetzt den vollständigen Befehl. Der PC hat sich inzwischen auf $033F erhöht. Nun führt der 650x den Befehl aus. Er sendet die Adresse $0380 zum Adressbus: wenn er den Inhalt (vielleicht $11) erhalten hat, speichert er diesen in das Register A. Das Register A enthält jetzt $11.
4. Der 650x ist nun bereit, den nächsten Befehl entgegenzunehmen: die Adresse $033F verläßt den PC über den Adressbus und das Programm läuft weiter.

Fragen und Aufgaben

Wußten Sie, daß Ihr Computer einen Speicherteil enthält, der „Bildschirmspeicher" genannt wird? Alles, was Sie in diesen Speicherteil eingeben, erscheint auf dem Bildschirm. Sie werden das in BASIC Handbüchern als „Bildschirm POKE" oder unter dem Stichwort POKE beschrieben finden.

Der Bildschirm des PET/CBM beginnt ab $8000 aufwärts, beim VIC meist (aber nicht immer) ab $1E00 aufwärts, beim Commodore 64 liegt er gewöhnlich bei $0400 und beim PLUS/4 findet man ihn bei $0C00.

Wenn Sie ein Programm schreiben, um Informationen in den Bildschirmspeicher zu bringen, werden die entsprechenden Zeichen auf dem Bildschirm erscheinen. Vielleicht möchten Sie das versuchen. Sie können sogar, wenn Sie wollen, Zeichen auf dem Bildschirm herumspringen lassen.

Zwei Schwierigkeiten könnten jedoch auftauchen. Erstens, Sie könnten ein perfektes Programm schreiben, das Informationen an den oberen Bildschirmrand plaziert. Wenn das Programm jedoch endet, könnte der Bildschirm nach oben rollen (scroll) und das Resultat würde verschwinden. Zweitens, der VIC und der Commodore 64 benutzen Farbe und Sie könnten zufällig ein weißes Zeichen auf weißem Hintergrund produzieren. Die sind dann schwer zu erkennen.

Nun eine weitere Frage. Angenommen, ich würde Sie bitten, ein Programm zu schreiben, das den Inhalt von fünf Speicherzellen im Kreis verschiebt, von $0380 bis $0384, auf die Weise, daß der Inhalt von $0380 nach $0381, der von $0381 nach $0382 usw. verschoben wird und der von $0384 nach $0380 gelangt. Auf den ersten Blick scheinen wir auf Schwierigkeiten zu stoßen: wir haben keine fünf Register, sondern nur drei (A, X und Y). Können Sie sich einen Ausweg ausdenken, um dieses Problem zu lösen?

2. Kontrolle der Ausgabe

Dieses Kapitel behandelt:

- Aufruf von Maschinensprache-Unterroutinen
- Die PRINT-Unterroutine
- Direkte Adressierung
- Aufruf von Maschinensprache aus BASIC
- Einfache Assembler Programme
- Indizierte Adressierung
- Einfache Schleifen
- Disassemblierung

Aufruf von Maschinensprache-Unterroutinen

In BASIC kann man ein „Paket" von Programmanweisungen, das Unterroutine genannt wird, durch einen GOSUB Befehl zur Ausführung bringen. Die Unterroutine endet mit einer RETURN Anweisung. Dadurch wird das Programm gezwungen, zum Aufrufpunkt zurückzukehren; das heißt, zur Rückkehr zu dem Befehl, der der Anweisung GOSUB direkt folgt.

Derselbe Mechanismus steht in der Maschinensprache zur Verfügung. Eine Gruppe von Anweisungen können durch den Befehl „springe zur Unterroutine" (JSR = jump subroutine) aufgerufen werden. Der 650x springt zu der angegebenen Adresse und führt die dort abgespeicherten Befehle aus, bis er auf einen Befehl „kehre von Unterroutine zurück" (RTS = return from subroutine) trifft. Nun nimmt er die Ausführung der Befehle vom Aufrufpunkt an auf: beginnend mit dem Befehl, der direkt auf JSR folgt.

Wenn ich zum Beispiel in die Adresse $033C den Befehl JSR $1234 schreibe, verändert der 650x seinen PC auf $1234 und holt den nächsten Befehl von dieser Adresse. Die Befehlsausführung wird solange fortgesetzt, bis der Befehl RTS auftaucht. Jetzt schaltet der Mikroprozessor auf den Befehl zurück, der dem JSR folgt, in diesem Fall wäre das die Adresse $033F (der Befehl JSR hat eine Länge von drei Bytes).

Genau wie im BASIC können Unterroutinen verschachtelt werden, d.h. eine Unterroutine kann eine weitere aufrufen und diese Unterroutine kann wiederum eine andere aufrufen. Zu einem späteren Zeitpunkt werden wir die Unterroutine genauer behandeln. Im Augenblick wollen wir uns damit begnügen, fertige Unterroutinen aufzurufen.

Fertige Unterroutinen

Eine Reihe von brauchbaren Unterroutinen ist im ROM Speicher des Computers permanent abgelegt. Alle Commodore Maschinen besitzen einen Standardsatz von Unterroutinen, die durch Ihre Programme aufgerufen werden können. Sie liegen immer auf denselben Adressen und führen ungefähr das Gleiche aus, unabhängig davon, welche Commodoremaschine

man verwendet: PET, CBM, Commodore 64, PLUS/4 oder VIC-20. Diese Routinen werden „Kernal" - Unterroutinen genannt. Einzelheiten dazu findet man in den entsprechenden Commodore Handbüchern, wir wollen hier einige brauchbare Informationen liefern.

Die ursprüngliche Bedeutung des Begriffes „kernal" scheint verloren gegangen. Es handelte sich um eine Abkürzung für „Keyboard Entry Read, Network And Link". Heute ist es lediglich ein Kürzel, mit dem wir den Teil des Betriebssystems bezeichnen, der die Zusammenarbeit von Bildschirm, Tastatur, anderen Ein- und Ausgaben sowie von Kontrollmechanismen bewirkt. Um dieses „Kern"-Kontrollsystem zu beschreiben, wollen wir im Deutschen von den „Kern-Unterroutinen" sprechen.

Die drei Kern-Unterroutinen, mit denen wir es in den nächsten Kapiteln zu tun haben werden, sind folgende:

Adresse	Name	Wirkungsweise
$FFD2	CHROUT	Ausgabe von ASCII Zeichen
$FFE4	GETIN	Annahme von ASCII Zeichen
$FFE1	STOP	Test der RUN/STOP Taste

Mit den ersten beiden Unterroutinen können wir leicht Daten ein- und ausgeben. Die dritte erlaubt uns die Überwachung der Taste RUN/STOP, um uns gegen bestimmte Arten von Programmierfehlern abzusichern. In diesem Kapitel wollen wir CHROUT benutzen, um damit Informationen auf den Bildschirm zu schreiben.

CHROUT - Die Ausgabe Unterroutine

Die CHROUT-Unterroutine bei Adresse $FFD2 kann man für alle Ausgabearten verwenden: zum Bildschirm, zur Disk, zum Kassettenrekorder oder zu anderen Einheiten. Sie funktioniert wie PRINT und PRINT#, mit dem Unterschied, daß sie nur ein Zeichen ausgibt. Im Augenblick wollen wir CHROUT nur dazu verwenden, um Informationen an den Bildschirm zu senden.

Unterroutine: CHROUT
Adresse: $FFD2
Wirkung: schickt eine Kopie des Zeichens aus dem Register A auf den Ausgabekanal. Der Ausgabekanal ist solange der Computerbildschirm, bis er durch besondere Vorkehrungen umgeschaltet wurde.

Das ausgesandte Zeichen besteht gewöhnlich aus einem ASCII Zeichen (oder PET ASCII). Alle Spezialzeichen - Graphik, Farbcodes, Cursorbewegungen - werden, wenn sie zum Bildschirm gesandt werden, auf die übliche Weise behandelt.

Register: Alle Datenregister bleiben während eines CHROUT erhalten. Nach Rückkehr aus der Unterroutine sind A, X und Y nicht verändert.

Status: Die Statusflags könnten verändert sein. Im VIC und Commodore 64 zeigt die C (carry) Flag bestimmte Schwierigkeiten bei der Ausgabe an.

Um den Buchstaben X auf den Bildschirm zu schreiben, können wir folgende Schritte ausführen:

1. Bring das ASCII Zeichen für X ($58) in das Register A.
2. JSR zu Adresse $FFD2.

Warum nicht POKE?

Um etwas auf dem Bildschirm anzuzeigen, könnte es einen einfacheren Weg geben. Wir könnten Informationen durch POKE direkt in den Bildschirmspeicher bringen; in Maschinensprache würden wir das eher „store" als POKE nennen. In dem Augenblick, in dem wir etwas in diesem Speicherbereich verändern, wird sich auch die Information verändern, die auf dem Bildschirm angezeigt wird. Der Bildschirmspeicher liegt im allgemeinen bei den folgenden Adressen:

PET/CBM	ab $8000 (dezimal 32768)
Commodore 64:	ab $0400 (dezimal 1024)
264/364:	ab $0C00 (dezimal 3072)
VIC-20:	ab $1E00 (dezimal 7680)

Der Bildschirmspeicher des VIC-20 kann an verschiedenen Stellen des Speichers liegen. Die genaue Position hängt davon ab, wieviel zusätzlicher RAM Speicher angeschlossen ist.

Von Fall zu Fall können Bildschirm POKEs die beste Lösung darstellen. Wir wollen jedoch häufiger die Unterroutine CHROUT bei $FFD2 benutzen. Hier sind einige Gründe dafür:

- Wie bei PRINT müssen wir uns keine Gedanken darüber machen, wohin wir das nächste Zeichen setzen: es wird automatisch auf die Cursorposition gebracht.
- Wenn der Bildschirm voll ist, erfolgt ein Rollen (scroll) automatisch.
- Der Bildschirmspeicher braucht Spezialzeichen. Das Zeichen X z.B. hat den Standard ASCII Code $58, um dieses durch POKE auf den Bildschirm zu bringen, müßten wir den Code $18 benutzen. Die CHROUT Unterroutine benutzt $58.
- Der Bildschirmspeicher könnte in Abhängigkeit von dem benutzten System und Programm an unterschiedlicher Stelle liegen. Die POKE Adresse müßte dann auch verändert werden, CHROUT hingegen funktioniert weiter.
- Spezielle Kontrollzeichen werden berücksichtigt: $0D für RETURN , um auf den Zeilenanfang zu gelangen, Cursorbewegungen, Farbänderungen. Wir können sogar den Bildschirm löschen, indem wir das Bildschirm Löschzeichen ($93) laden und $FFD2 aufrufen.
- Um beim VIC oder Commodore 64 ein Bildschirm POKE auszuführen, muß das entsprechende Farbnybble (Teilbyte) mit POKE in den Speicher gebracht werden (siehe Anhang B für die zugehörige Speicheraufteilung). Durch die Unterroutine ab $FFD2 wird die Farbe automatisch gesetzt.

Ein PRINT Vorhaben

Wir wollen einige Anweisungen schreiben, um den Buchstaben H auf den Bildschirm zu bringen. Zur Abspeicherung unseres Programms wollen wir wieder die Adresse $033C, den Kassettenpuffer, benutzen. Zur Erinnerung: bevor wir mit dem Vorhaben beginnen, sollte unser Monitor geladen und bereit sein.

Zuerst der Plan; wir entwerfen unsere Befehle.

```
LDA #$48
```

Wir benutzen ein neues Symbol (#), um eine spezielle Art der Information anzuzeigen. Das kann man durch unterschiedliche Zeichen erreichen: durch das englische Zeichen für Pfund oder das englische Zeichen für Nummer (#). Der formalere Name für dieses Symbol lautet „octothorpe", was soviel wie „acht Punkte" bedeutet. Wie immer man es auch nennen mag, das Symbol weist darauf hin, „die folgende Information ist nicht als Adresse, sondern als Wert zu verstehen". Mit anderen Worten, wir wollen nicht, daß der Computer zur Adresse $48 geht, vielmehr soll er das Register A mit dem Wert $48 laden, der in ASCII das Zeichen H repräsentiert. Diese Art einer Informationsermittlung wird direkte Adressierung genannt. Anders ausgedrückt, hole die Information auf direktem Wege, gehe dazu nicht zum Speicher.

```
JSR $FFD2
```

Der vorangegangene Befehl brachte den Buchstaben H in das A Register, dieser hier schreibt ihn auf den Bildschirm. Jetzt müssen wir nur noch das Programm beenden. BRK bringt uns zum Maschinensprachemonitor zurück.

Monitor Erweiterungen

Wir könnten die Schritte des vorherigen Kapitels wiederholen: durch Handassemblierung des Quellcodes in Maschinensprache die Befehle in den Speicher bringen. Wir müßten die Befehlscodes genau kennen, und dann eine gewissenhafte Übersetzung durchführen. Es gibt jedoch einen einfacheren Weg.

Die meisten Maschinensprachemonitore enthalten Zusatzbefehle, die uns bei dieser mechanischen Form der Übersetzung helfen. Wir wollen die Assemblermöglichkeiten dieser Monitore ausnutzen.

Die meisten Monitore stellen den Befehl zum Assemblieren (.A) zur Verfügung. Die eingebauten Monitore des PET/CBM bilden eine Ausnahme. Diese können jedoch dadurch erweitert werden, daß man ein Erweiterungsprogramm für den Monitor, wie z.B. „Supermon", lädt. Die Commodore der Serie PLUS/4 enthalten einen erweiterten Monitor mit dem Befehl .A.

Diese Assembler werden oft „nicht-symbolische Assembler" genannt. Das bedeutet, daß man immer dann, wenn der Assembler eine Adresse benötigt, die exakte Adresse liefern muß. Sie können nicht statt dieser einen Namen wie CHROUT eintippen und dann erwarten, daß so ein einfacher Assembler weiß, welche Adresse dadurch repräsentiert wird. Stattdessen müssen Sie $FFD2 eingeben.

Laden Sie nun Ihren Monitor und eventuell eine Monitor Erweiterung. Führen Sie sämtliche notwendigen Vorbereitungen aus und tippen Sie das folgende Monitorkommando

```
A 033C  LDA  #$48
```

Wir fordern den Computer auf, mit der Assemblierung (.A) des Befehls LDA bei der Adresse $033C zu beginnen (beachten Sie, daß wir hier nicht das $ Zeichen benutzen): lade A direkt

mit dem Wert $48, der das ASCII Zeichen für H repräsentiert. Wenn Sie nach Eingabe dieser Zeile RETURN drücken, wird der Computer wahrscheinlich eine der folgenden Reaktionen zeigen:

1. Er reagiert überhaupt nicht außer, daß irgendwo auf der Zeile ein Fragezeichen erscheint. Das Fragezeichen deutet auf einen Fehler in der Eingabe hin. Wenn das Fragezeichen direkt hinter dem Buchstaben A erscheint, versteht Ihr Monitor den Befehl .A (assembliere) nicht. Besorgen Sie sich einen anderen Monitor oder wiederholen Sie sorgfältig die Vorbereitungsprozedur.
2. Ihre Instruktion wird korrekt übersetzt und der Objektcode in den Speicher, beginnend bei der angegebenen Adresse, abgelegt. In diesem Fall sollte in der Adresse $033C $A9 und in Adresse $033D $48 stehen. Der Monitor hilft Ihnen anschließend, indem er einen Teil der zu erwartenden nächsten Instruktion druckt. Der Computer erwartet, daß Sie folgenden Zeilenanfang tippen wollen

```
.A 033E
```

Er schreibt den ersten Teil dieser Zeile auf den Bildschirm, um Ihnen das Tippen abzunehmen. Der Bildschirm sollte jetzt so aussehen:

```
.A 033C  LDA  #$48
.A 033E
```

Jetzt können Sie den Befehl durch Eintippen von JSR $FFD2 und RETURN vervollständigen. Wieder wird der Computer Ihre nächste Zeile vorwegnehmen, indem er .A 0341 ausdruckt. Jetzt können Sie das letzte Kommando eingeben, BRK. Der Bildschirm sieht nun folgendermaßen aus:

```
.A 033C  LDA  #$48
.A 033E  JSR  $FFD2
.A 0341  BRK
.A 0342
```

Der Computer wartet noch auf einen weiteren Befehl. Da wir keinen weiteren haben, tippen wir RETURN, um ihm zu zeigen, daß wir fertig sind.

An diesem Punkt ist unser Programm in den Speicher geladen. Die Anweisungen wurden direkt in den Speicher assembliert und der Objektcode ist hoffentlich lauffähig.

Bedenken Sie, daß wir die Mühe abgenommen bekommen, den op code für jede Instruktion nachzuschauen oder ihn auswendig zu lernen. Wir müssen auch nicht die Länge jeder einzelnen Instruktion berücksichtigen, der Assembler tut das für uns.

Wenn Sie wollen, können Sie den Speicherinhalt anzeigen und das Objektprogramm inspizieren. Mit .M 033C 0341 werden Sie die Bytes Ihres Programms im Speicher sehen:

```
.:033C A9 48 20 D2 FF 00 xx xx
```

Die ersten sechs Bytes sind Ihr Programm. Die letzten zwei Bytes sind nicht von Interesse: sie stellen ein Überbleibsel in diesem Teil des Speichers dar. Wir brauchen uns darum nicht zu kümmern, da das Programm anhalten wird, wenn es bei Adresse $0341 den Befehl BRK

($00) erreicht. Es wird sich dann mit dem Inhalt der Speicherstellen $0342 und $0343 nicht mehr beschäftigen.

Überprüfung: Der Disassembler

Als wir unseren Quellcode in Objektcode veränderten, nannten wir diesen Prozeß der Übersetzung assemblieren und wir nannten ein Programm, das das ausführt, einen Assembler.

Jetzt haben wir ein Programm geschrieben und es ist sicher in den Speicher geladen. Wir haben den Speicher inspiziert und die Bytes dort gesehen. Sie sind allerdings schwierig zu lesen. Es wäre bequem, wenn wir eine umgekehrte Assemblierung durchführen, also die Speicherinhalte in den Quellcode zurückübersetzen könnten. Der Monitor besitzt diese Fähigkeit in Form eines Disassembler. Wenn wir den Computer anweisen, den Code ab $033C zu disassemblieren, wird er den dortigen Code untersuchen und feststellen, daß der Inhalt ($A9) einem Befehl „LDA direkt" entspricht. Er wird zu unserer Information LDA #$48 ausdrucken, was verständlicher ist als die beiden Originalbytes A9 48.

Geben Sie den Befehl .D 033C und RETURN ein. D steht für disassembliere, die Adresse muß darauf folgen.

Der Computer wird nun einen vollen Bildschirm zeigen. Auf der linken Seite steht die Adresse gefolgt von den Bytes, aus denen die Instruktion zusammengesetzt ist. Auf der rechten Seite steht der rekonstruierte Quellcode. Der Bildschirm zeigt wieder mehr Speicherstellen, als unser Programm benutzt. Wieder können wir alle Zeilen jenseits Adresse $0341, die die letzte Instruktion unseres Programmes enthält, ignorieren. Alles, was darauf folgt, ist „Abfall", der im Speicher übrig blieb, vom Programm aber nicht benutzt wird.

Eine interessante Eigenschaft der meisten Disassembler-Listings besteht darin, daß der Cursor auf der letzten Zeile des Listing aufleuchtet und nicht auf der Zeile darunter. Wenn Sie ein langes Programm haben, können Sie den Buchstaben D gefolgt von RETURN eintippen und der nächste Teil Ihres Programms wird sofort angezeigt. Andererseits, wenn Sie keinen weiteren Code disassemblieren möchten, bringen Sie den Cursor nach unten auf eine leere Zeile, bevor Sie den nächsten Befehl eintippen.

Durch Disassemblieren läßt sich gut auf mögliche Fehler testen. Wenn Sie im Listing einen Fehler finden, können Sie diese Zeile durch erneutes Assemblieren verbessern, indem Sie erneut den Befehl .A eingeben. Kleinere Fehler können so direkt auf der linken Seite des Disassembler-Listings korrigiert werden. Nehmen wir an, Sie hätten während der Assemblierphase fehlerhaft LDA #$58 eingetippt. Wenn Sie anschließend disassemblieren, sieht diese Zeile folgendermaßen aus:

```
., 033C A9 58   LDA #$58
```

Sie erkennen, daß die 58 heißen sollte 48. Sie können nun den Cursor nach oben bewegen - wenn Sie wollen mit der Taste CURSOR HOME - und den Wert auf der linken Seite überschreiben. In diesem Fall bewegen Sie den Cursor auf die 5, tippen die 4 ein, um die Anzeige auf 48 zu verändern, und tippen RETURN. Auf der Anzeige werden Sie sehen, daß der Fehler behoben ist.

Programmlauf

Wenn nötig, bewegen Sie den Cursor nach unten auf eine leere Zeile, tippen den Befehl .G 033C und das Programm wird ablaufen. Auch diesmal braucht es nicht lange. Der Sprung zurück in den MLM scheint augenblicklich zu erfolgen. Wo ist der Buchstabe H geblieben, den wir ausdrucken wollten? Er ist schwer zu sehen, aber er ist da. Schauen Sie auf das Kommando .G 033C und Sie werden ihn sehen.

Aufgabe für Enthusiasten: können Sie das Programm erweitern und HI ausdrucken? Der ASCII Code für den Buchstaben I ist $49. Können Sie abermals erweitern und HI auf eine eigene Zeile schreiben? Der ASCII Code für ein RETURN ist $0D. Denken Sie daran, daß alle ASCII Zeichen im Anhang D zu finden sind. Schauen Sie in der Spalte, die mit ASCII markiert ist, nach.

Kopplung mit BASIC

Bis jetzt haben wir unser Programm durch den Befehl .G (go) aus dem MLM gestartet und wir haben es mit BRK zur Rückkehr in den Monitor gezwungen. Das ist kein bequemer Weg, um ein Programm ablaufen zu lassen. Die meisten Benutzer würden lieber aus dem BASIC heraus RUN sagen und den Computer so zur Ausführung veranlassen.

Wir können in BASIC eine Verbindung mit einem Maschinenspracheprogramm tatsächlich herstellen. Wenn dieses Programm endet, kann es zu BASIC zurückkehren und dem BASIC Programm die Fortsetzung der BASIC Instruktionen ermöglichen. Wir benötigen dazu folgende Befehle

(BASIC) SYS - gehe in die angegebene Adresse der Maschinenspracheunterroutine

(Maschinensprache) RTS - kehre an die Stelle zurück, an der die Unterroutine aufgerufen wurde.

Wir wollen zuerst unser Maschinenspracheprogramm verändern. Wir müssen das BRK am Ende durch RTS (return from subroutine) ersetzen, damit das Programm nach Beendigung ins BASIC zurückkehrt. Wenn Sie wollen, können Sie direkt das Disassembler-Listing verändern: disassemblieren und überschreiben Sie die Bytes, die das BRK repräsentieren, mit dem Wert 60. Tippen Sie RETURN und Sie werden sehen, daß der Befehl zu RTS verändert ist. Alternativ können Sie erneut assemblieren mit

```
.A 033C LDA #$48
.A 033E JSR $FFD2
.A 0341 RTS
```

Kehren Sie jetzt ins BASIC zurück (mit dem Befehl .X). Der Computer antwortet mit READY. Jetzt können Sie Ihr Programm mit dem Befehl SYS aufrufen.

Die Adresse $033C lautet dezimal 828, deshalb tippen wir SYS 828. Wenn wir RETURN tippen, wird der Buchstabe H ausgedruckt.

Wir sind noch nicht fertig. Jede Maschinensprache Unterroutine kann von einem BASIC Programm aufgerufen werden. Tippen Sie NEW, um den BASIC Arbeitsbereich zu löschen. Dadurch wird unser Maschinenspracheprogramm nicht angetastet, weil NEW ein BASIC Befehl ist. Geben Sie jetzt das folgende Programm ein:

```
100 FOR J=1 TO 10
110 SYS 828
120 NEXT J
```

Wie oft wird unser Programm bei 828 ($033C) aufgerufen? Wie oft wird der Buchstabe H ausgedruckt? Werden die Buchstaben auf derselben Zeile oder auf unterschiedlichen Zeilen stehen? Tippen Sie RUN und schauen Sie es sich an.

Vorhaben für Enthusiasten: Verändern Sie wieder das Maschinenspracheprogramm, um HI zu sagen. Lassen Sie Ihre Phantasie spielen! Was möchten Sie den Computer noch sagen lassen? Möchten Sie Farbe oder die inverse Darstellung benutzen?

Wir haben nun ein wichtiges neues Niveau erreicht: BASIC und Maschinensprache arbeiten zusammen. Das ist für den Benutzer, der keine speziellen Monitorbefehle lernen möchte, einfacher. Es ist auch für den Programmierer einfacher, da er Aufgaben, die leichter in BASIC ausgeführt werden können, in dieser Sprache schreiben kann. Dagegen lassen sich Dinge, die in BASIC kompliziert sind oder zu langsam ablaufen, in Maschinensprache schreiben. Wir können die besten Seiten beider Welten nutzen.

Wir wollen die drei verschiedenen Typen von Unterroutine-Aufrufen unterscheiden:

GOSUB - ruft eine BASIC Unterroutine aus einem BASIC Programm.
SYS - ruft eine Maschinensprache Unterroutine aus einem BASIC Programm.
JSR - ruft eine Maschinensprache Unterroutine aus der Maschinensprache.

Schleifen

Wir wissen jetzt, wie man ein Zeichen nach dem anderen auf den Bildschirm sendet. Auf der anderen Seite würden lange Sätze wie z.B. „DIE SCHNELLE SCHWARZE KATZE . . .“ zu einer entnervenden Codierarbeit führen, wenn wir für jeden Buchstaben, den wir senden wollen, einen eigenen Befehl schreiben müßten. Wir sollten besser eine Programmschleife einsetzen, um den Vorgang des Ausdruckens zu wiederholen.

Wir müssen das Wort HALLO irgendwo im Speicher ablegen. Es spielt keine Rolle wo. Hauptsache, es gerät nicht mit irgendetwas anderem in Konflikt. Ich will die Adressen $034A bis $034F willkürlich wählen. Dort wollen wir das Wort gleich speichern. Erinnern Sie sich, daß die Zeichen, die das Wort HALLO (plus das RETURN) bilden, keine Programmbefehle sind, es sind einfache Daten. Wir müssen sie mit einer Speicherveränderung an ihre Stelle bringen - wir dürfen nicht versuchen, sie zu assemblieren.

Während wir die Zeichen senden, müssen wir sie zählen. Da wir sechs Zeichen senden wollen, stellt die Zahl sechs unsere obere Grenze dar. Wir wollen das X Register beim Zählen als Merkregister benutzen. Zuerst müssen wir X auf null setzen.

```
.A 033C LDX #$00
```

Achten Sie auf das #-Symbol, um einen direkten Wert anzuzeigen: wir wollen X mit dem Wert null laden, nicht irgendetwas aus der Adresse 0. Jetzt wollen wir etwas Neues versuchen. Ich möchte ein auszudruckendes Zeichen aus der Adresse $034A nehmen. Aber langsam, das ist nur das erste von mehreren Malen. Wenn wir zu diesem Punkt innerhalb der Schleife zurückkehren, möchte ich ein Zeichen aus $034B und dann aus $034C nehmen usw.

Wie können wir das bewerkstelligen? Es sieht zunächst so aus, daß wir eine Adresse in die Instruktion LDA schreiben müssen und daß diese Adresse nicht verändert werden kann. Es gibt jedoch einen Ausweg.

Wir können den Computer beauftragen, die Adresse, die wir liefern, anzunehmen, und den Inhalt von X oder Y zu dieser Adresse zu addieren, bevor wir sie benutzen. Die errechnete Adresse wird „Effektivadresse" genannt.

Wir wollen schauen, wo wir gerade stehengeblieben sind. Nach dem ersten Durchgang durch die Schleife zählt X die Zeichen und hat einen Wert von Null. Wenn wir unsere Adresse mit 034A + X bezeichnen, wird die Effektivadresse zu 034A. Das ist die Stelle, an der wir den Buchstaben H gespeichert haben müssen.

Wenn wir erneut aus der Schleife zurückkehren - wir haben diesen Teil noch nicht geschrieben - sollte X gleich 1 sein. Eine Adresse von 034A + X würde eine Effektivadresse von 034B ergeben. Der Computer würde dorthin gehen und den Buchstaben A holen. Wenn wir die Schleife weiter durchlaufen, werden die Buchstaben L, L, O und RETURN nach Bedarf hereingeholt werden.

Wenn wir den Befehl LDA eingeben, tippen wir das Pluszeichen nicht mit ein. Stattdessen zeigen wir die Indizierung durch ein Komma an: LDA $034A,X. Wir können entweder X oder Y zum Indizieren benutzen. Diese werden deshalb auch manchmal „Indexregister" genannt. In unserem Fall benutzen wir X. Wir codieren:

```
.A 033E  LDA $034A,X
.A 0341  JSR $FFD2
```

Wenn der Computer zum ersten Mal die Schleife durchläuft, lädt er den Inhalt von Adresse $034A (den Buchstaben H von HALLO) und druckt ihn aus. Wenn die Schleife hierher zurückkehrt, mit dem Inhalt von X gleich 1, wird dieser Befehl den Inhalt von $034B laden und den Buchstaben A drucken.

Das X Register zählt die Anzahl der zu druckenden Buchstaben. Deshalb müssen wir zum Inhalt von X Eins addieren. Es gibt einen speziellen Befehl, mit dem wir Eins zum Inhalt von X addieren können: INX für „inkrementiere X". Ein ähnlicher Code, INY, erlaubt die Inkrementierung von Y. DEX (dekrementiere X) und DEY (dekrementiere Y) erlauben den Inhalt von X oder Y zu dekrementieren oder um Eins zu erniedrigen. Wir benutzen jetzt INX zum zählen:

```
.A 0344  INX
```

Nun können wir X testen, um zu sehen, ob es schon den Wert sechs erreicht hat. Nach der ersten Runde wird das wohl nicht der Fall sein, da X mit Null begann und zum Wert Eins inkre-

mentiert wurde. Wenn X nicht gleich sechs ist, wollen wir nach $033E zurückkehren und einen weiteren Buchstaben drucken. Hier ist unsere Codierung:

```
.A 0345  CPX #$06
.A 0347  BNE $033E
```

CPX steht für „Vergleiche (compare) X". Beachten Sie, daß wir für einen direkten Wert von sechs testen, weshalb wir das #-Symbol benutzen. BNE bedeutet „Verzweige, wenn ungleich (branch not equal)". Wenn X nicht gleich sechs ist, kehren wir zur Adresse $033E zurück.

Ein wenig Überlegung ergibt, daß das Programm fünfmal für eine Gesamtzahl von sechs Runden zurückkehrt. Genau das wollen wir.

Wir wollen uns jetzt den gesamten Code inklusive RTS ansehen:

```
.A 033C  LDX #$00
.A 033E  LDA $034A,X
.A 0341  JSR $FFD2
.A 0344  INX
.A 0345  CPX #$06
.A 0347  BNE $033E
.A 0349  RTS
```

Wir können nun die Zeichen für HALLO in den Speicher eingeben. Es handelt sich dabei um Daten und nicht um Befehle, weshalb wir sie nicht assemblieren dürfen. Stattdessen verändern wir den Speicher wie üblich, indem wir ihn anzeigen und überschreiben. Wir geben den Befehl .M 034A 034F ein und überschreiben die Anzeige, damit sie folgendes Aussehen erhält:

```
.:034A  48 41 4C 4C 4F 0D xx xx
```

Durch einen glücklichen Zufall passen diese Daten genau hinter unser Programm.

Jetzt sollte alles fertig sein. Disassemblieren Sie das Programm ab $033C und überprüfen Sie es. Sie werden feststellen, daß die Daten ab $034A nicht richtig disassembliert werden, was zu erwarten war: diese Bytes sind keine Instruktionen und können deshalb auch nicht decodiert werden.

Wenn sonst alles vernünftig aussieht, kehren Sie ins BASIC zurück (mit .X) und versuchen Sie SYS 828. Der Computer sollte mit HALLO antworten.

Jetzt schreiben wir wieder ein Schleifenprogramm in BASIC:

```
100 FOR J=1 TO 3
110 SYS 828
120 NEXT J
```

Eine Bemerkung zu SAVE

Wenn Sie das Programm auf Kassette speichern möchten, werden Sie beim VIC oder Commodore 64 auf eine Schwierigkeit stoßen. Das Maschinenspracheprogramm befindet sich

nämlich im Kassettenpuffer. Ein Befehl „Speichere auf Kassette" würde dazu führen, daß der Inhalt des Puffers gelöscht würde, bevor das Programm auf Band geschrieben werden könnte. Sogar Disk-Befehle würden nicht sicher funktionieren: die Disk-Befehle der BASIC Version 4.0 benutzen ebenfalls den Kassettenpuffer als Arbeitsbereich. Wenn wir diese Befehle benutzen, würden wir mit großer Wahrscheinlichkeit unser Maschinenspracheprogramm zerstören.

Unser Hauptproblem besteht aber nicht darin, das Programm zu speichern. Ein korrekt gespeichertes Programm kann auch noch zu Schwierigkeiten führen, wenn man versucht, es zurückzuladen und sicher zum Laufen zu bringen. Diese Schwierigkeit hängt mit den BASIC Zeigern zusammen, speziell mit dem Zeiger, der auf den Anfang der Variablentabelle deutet. Dieses Problem und seine Lösung wird in Kapitel 6 genauer behandelt werden.

Ein Lückenfüller für SAVE

Wir können kurze Programme dadurch speichern, daß wir sie als Teil von DATA Zeilen codieren. Die Vorgehensweise ist einfach, wenn wir die Bildschirmeditierung intelligent einsetzen.

Wir wissen, daß sich das Programm von $033C bis $034F erstreckt, einschließlich der Botschaft (HALLO) am Ende. Die dezimale Entsprechung dieser Adressen lautet 828 bis 847. Geben Sie die folgende BASIC Zeile ein:

```
FOR J=828 TO 847:PRINT PEEK(J);:NEXT J
```

Wenn Sie sich diese Zeile genau anschauen, werden Sie erkennen, daß BASIC aufgefordert wird, durch den Teil des Speichers zu gehen, der das Maschinenspracheprogramm enthält, und dessen Inhalt (natürlich in dezimaler Notation) darzustellen. Sie werden ein Resultat erhalten, das ungefähr folgendermaßen aussieht:

```
162 0 189 74 3 32 210 255 232 224 6 208 245 96 72 65 76 76 79 13
```

Hierbei handelt es sich tatsächlich um die Bytes, die Ihr Programm bilden. Mit ein wenig Nachdenken können Sie rekonstruieren, daß die Kombination 162-0 bedeutet LDX #$00 oder die Kombination 72-65-76-76-79 am Ende das Wort HALLO in ASCII darstellt. Das sieht in dezimaler Schreibweise zwar unterschiedlich aus, aber es handelt sich um dieselben Zahlen.

Um die folgende Art der Bildschirmeditierung auszuführen, brauchen Sie vielleicht ein wenig Geschicklichkeit und Artistik. Sie können aber auch die Zahlen in DATA Zeilen in der Weise neu tippen, wie es unten gezeigt ist. Wie auch immer, ordnen Sie die Zahlen in folgender Weise an:

```
50 DATA 162,0,189,74,3,32,210,255,232,224,6
60 DATA 208,245,96,72,65,76,76,79,13
```

Wir haben eine genaue Kopie unseres Programms, wie es im Speicher erscheint, lediglich in DATA Zeilen abgelegt. Die DATA Anweisungen sind nun Teil eines normalen BASIC Programms und können mit SAVE und LOAD ohne Schwierigkeiten gespeichert beziehungsweise geladen werden.

Wir können jetzt unser Maschinenspracheprogramm dadurch rekonstruieren, daß wir es durch ein einfaches BASIC Programm mit Hilfe von POKE in den Speicher bringen:

```
80 FOR J=828 TO 847:READ X:POKE J,X:NEXT J
```

Jetzt ist unser Programm sicher und, man höre und staune, es läßt sich wie BASIC handhaben, führt aber eine Maschinensprache Aufgabe durch, wie wir es verlangen. Wir wollen jetzt noch einmal das gesamte BASIC Programm zeigen:

```
 50 DATA 162,0,189,74,3,32,210,255,232,224,6
 60 DATA 208,245,96,72,65,76,76,79,13
 80 FOR J=828 TO 847:READ X:POKE J,X:NEXT J
100 FOR J=1 TO 3
110 SYS 828
120 NEXT J
```

Diese Methode, Maschinenspracheprogramme zu speichern, ist sauber und frei von Schwierigkeiten, wird aber unpraktisch, wenn es sich dabei um lange Programme handelt. Wir wollen in Kapitel 6 eine weiterentwickelte Methode besprechen.

Was wir gelernt haben

- Unterroutinen können mit dem Befehl JSR aus der Maschinensprache heraus aufgerufen werden. Es gibt eine Reihe nützlicher Kernunterroutinen, die permanent verfügbar sind.
- Ein BASIC Programm kann ein Maschinenspracheprogramm als Unterroutine aufrufen: das BASIC Kommando lautet SYS. Die Maschinensprache Unterroutine kehrt durch den Befehl RTS (return from subroutine) an den Punkt des Aufrufs zurück.
- Die Unterroutine CHROUT ab Adresse $FFD2 erlaubt die Ausgabe von Zeichen, gewöhnlich zum Bildschirm. Außer den druckbaren Zeichen können spezielle Cursor- und Farbkontrollzeichen ausgegeben werden.
- Die meisten Maschinensprachemonitore beinhalten einen kleinen Assembler, der bei der Programmvorbereitung hilft, und einen Disassembler, der bei den Programmtests zur Seite steht.
- Der direkte Modus wird durch das #-Symbol angezeigt. Der Computer wird aufgefordert, den angegebenen Wert zu übernehmen, statt an eine spezifizierte Adresse zu springen.
- X und Y werden Indexregister genannt. Wir können den Inhalt von X oder Y zu einer angegebenen Adresse addieren, um eine Effektivadresse zu bilden, die sich während des Programmablaufs verändern läßt. Diese Addition wird Indizierung genannt.
- Auf X und Y wirken spezielle Befehle, die die gewählten Register um Eins erhöhen oder erniedrigen. Diese Befehle werden Inkrementieren oder Dekrementieren genannt und durch INX, INY, DEX und DEY codiert.

Fragen und Aufgaben

Schauen Sie sich in Anhang D die Tabelle der ASCII Zeichen an. Hex 93 bedeutet „lösche Bildschirm". Schreiben Sie ein Programm, das den Bildschirm löscht und „HA HA" schreibt.

Sie werden bemerkt haben, daß wir in unserem Beispiel das Register X von Null bis zu dem gewünschten Wert aufwärts haben zählen lassen. Was würde passieren, wenn Sie bei X mit 5 anfangen und rückwärts zählen würden? Versuchen Sie es, wenn Sie Lust dazu haben.

Erinnern Sie sich, daß Sie auch Cursorbewegungen, Farbcodes (falls Ihre Maschine Farbe hat) und andere spezielle ASCII Zeichen mit verwenden können. Könnten Sie den Code so verändern, daß Sie einen Kasten malen können? (Versuchen Sie es zuerst in BASIC). Malen Sie einen Kasten mit dem Wort „HALLO“ darin.

3. Flags, Logische Entscheidungen und Eingabe

Dieses Kapitel behandelt:

- Flags für die Statusinformation
- Prüfbare Flags: Z, C, N und V
- Zahlen mit Vorzeichen
- Das Statusregister
- Grundbegriffe der Unterbrechung
- Logische Operatoren: OR, AND, EOR
- Die GETIN-Unterroutine für die Eingabe
- Die STOP-Unterroutine

Flags

Gegen Ende des zweiten Kapitels schrieben wir ein Programm, das die scheinbar natürliche Folge enthielt:

```
CPX #$06
BNE $...
```

Das war sinnvoll: vergleiche X mit dem Wert 6 und verzweige zurück, wenn diese Werte nicht gleich sind. Dieser Programmteil setzt jedoch etwas ungewöhnliches voraus: die beiden Befehle sind irgendwie miteinander gekoppelt.

Wir wollen für einen Augenblick den Dingen etwas vorgreifen. Selbst wenn Sie ein Maschinenspracheprogramm laufen lassen, geschieht es, daß der Computer sechzigmal in der Sekunde „einfriert". Der Computer beschäftigt sich dann mit einer speziellen Aufgabe, „Unterbrechungsbearbeitung (interrupt processing)" genannt. Was er auch immer im Augenblick tut, er hält an und schaltet auf einen neuen Satz von Programmen um, die verschiedene Aufgaben erfüllen: Blinken des Cursors, Abfrage der Tastatur, die Uhr auf den neuesten Stand bringen und prüfen, ob der Kassettenmotor mit Spannung versorgt werden muß. Wenn er damit fertig ist, „taut" er das Hauptprogramm auf und setzt es an der Stelle fort, an der er es verlassen hat.

Eine solche Unterbrechung könnte zwischen den beiden obigen Befehlen stattgefunden haben, d.h. nach dem CPX und vor dem BNE. Sogar hunderte von Unterbrechungsbefehlen könnten zwischen diesen beiden Instruktionen ausgeführt werden, ohne daß etwas schief läuft. Die beiden Instruktionen arbeiten perfekt zusammen, um den gewünschten Erfolg zu erzielen. Wie stellt der Computer das an?

Die beiden Instruktionen sind mit Hilfe einer Flag verknüpft. Diese bildet einen Teil des 650x, der darüber Buch führt, daß etwas geschehen ist. Der Befehl CPX untersucht X und hebt oder senkt eine spezielle „Flagge" (flag), um damit das Ergebnis dieses Vergleichs anzuzeigen: gleich oder ungleich. Der Befehl BNE untersucht nun seinerseits diese Flag. Ist sie eingeschaltet (was gleich bedeutet), findet keine Verzweigung statt und das Programm fährt mit der nächstfolgenden Instruktion fort. Ist sie ausgeschaltet (was ungleich bedeutet), findet eine Verzweigung statt.

Mit anderen Worten, einige Befehle hinterlassen eine „Spur" von Statusinformationen. Andere Befehle können diese Informationen testen. Die Statusinformation wird „Flag" genannt. Es gibt vier Flags, die geprüft werden können: Z, C, N und V. Mit diesen wollen wir uns nun beschäftigen.

Z Flag

Die Z (zero) Flag hat vielleicht einen falschen Namen. Sie hätte eher E (für „equals" - ist gleich) Flag genannt werden sollen. Nach jedem Vergleich (CPX vergleiche X, CPY vergleiche Y oder CMP vergleiche A) wird die Z Flag auf „an" geschaltet, wenn die verglichenen Werte gleich sind, andernfalls wird sie auf „aus" geschaltet. In einigen Fällen testet die Z Flag auf Null-Gleichheit, woher auch ihr Name rührt, Z für zero. Das trifft für jede Aktivität zu, die eines der drei Datenregister verändern könnte. Daraus folgt, daß jeder Ladebefehl den Status der Z Flag beeinflussen kann. Dasselbe gilt für die Befehle Inkrementiere und Dekrementiere, die offensichtlich die Register verändern. Später werden wir auf weitere Operationen stoßen, wie Addition und Subtraktion, die auch die Z Flag beeinflussen können.

Daneben gibt es viele Befehle, die auf die Z Flag (oder irgendeine Flag) keinen Einfluß haben. Speicherbefehle (STA, STX, STY) verändern nie eine Flag. Verzweigungsbefehle testen zwar Flags, aber sie verändern sie nicht.

Ein Beispiel soll illustrieren, auf welche Weise einige Befehle Flags verändern, andere dagegen nicht. Untersuchen Sie den folgenden Code:

```
LDA #$23  (lade 23 nach A)
LDX #$00  (lade Null nach X)
STA $1234 (speichere 23 nach Adresse $1234)
BEQ $....
```

Wird die Verzweigung (BEQ) gewählt oder wird der 650x mit dem nächsten Befehl fortfahren? Wir wollen die Aktivität der Z Flag Schritt für Schritt analysieren. Der erste Befehl (LDA #$23) setzt die Z Flag zurück, weil 23 nicht gleich Null ist. Der zweite Befehl (LDX #$00) setzt die Z Flag, weil dieser Wert Null ist. Der dritte Befehl (STA $1234) beeinflußt die Z Flag nicht. Tatsächlich werden durch Speicherbefehle überhaupt keine Flags beeinflußt. In dem Augenblick also, in dem wir den Befehl BEQ erreichen, wird die Z Flag auf „an" geschaltet und die Verzweigung findet statt.

650x Handbücher zeigen die spezifischen Flags, die durch jeden Befehl beeinflußt werden. Im Zweifelsfall kann man dort einfach nachschauen.

Die Z Flag ist ziemlich stark beschäftigt. Sie wird sehr oft „an" und „aus" geschaltet, da sie von vielen Befehlen beeinflußt wird. Es handelt sich um eine wichtige Flag.

Wenn die Z Flag auf „an" gesetzt ist, verzweigt der Befehl BEQ (verzweige wenn gleich - branch equals) zu der angegebenen Adresse, andernfalls wird dieser Befehl ignoriert und der darauffolgende Befehl ausgeführt. Wenn die Z Flag auf „aus" zurückgesetzt ist, verzweigt der Befehl BNE (verzweige bei ungleich - branch not equals).

Wir können die Arbeitsweise unseres Programms aus Kapitel 2 etwas genauer betrachten. CPX #$06 bewirkt, daß die Z Flag auf „an" gesetzt wird, wenn X den Wert 6 enthält, andernfalls setzt er die Z Flag auf „aus" zurück. BNE testet diese Flag und verzweigt nur in dem Fall zur Schleife zurück, in dem die Z Flag auf „aus" geschaltet ist - mit anderen Worten, nur dann, wenn der Inhalt von X nicht gleich sechs ist.

C Flag

Die C (carry) Flag hat möglicherweise auch einen falschen Namen erhalten. Diese hätte eher GE (greater equal) Flag genannt werden sollen, da die C Flag nach einem Vergleich (CPX, CPY oder CMP) dann auf „an" geschaltet wird, wenn eines der zugehörigen Register (X, Y oder A) größer oder gleich dem Wert ist, mit dem es verglichen wurde. Wenn das zugehörige Register einen kleineren Wert enthält, wird die C Flag auf „aus" zurückgeschaltet.

Die C Flag ist nicht so beschäftigt, wie die Z Flag. Die C Flag wird nur durch Vergleichsbefehle und durch Rechenoperationen beeinflußt (Addition, Subtraktion und einen bestimmten Typ der Multiplikation und Division, Rotieren oder Schieben genannt). Die C Flag trägt ihren Namen dann zu Recht, wenn sie bei arithmetischen Operationen benutzt wird, da sie als ein „Übertrag" Bit bei der Verrechnung verschiedener Spalten fungiert. Beispielsweise beeinflußt ein LDA Befehl immer die Z Flag, da ein Register verändert wurde. Er beeinflußt aber niemals die C Flag, da keine Rechen- oder Vergleichsoperation durchgeführt wurde.

Wenn die C Flag auf „an" geschaltet wurde, verzweigt der Befehl BCS (verzweige wenn Übertrag gesetzt - branch carry set) zur angegebenen Adresse. Andernfalls wird er ignoriert und der nächstfolgende Befehl ausgeführt. Wenn die C Flag auf „aus" zurückgesetzt ist, verzweigt der Befehl BCC (verzweige wenn Übertrag null - branch carry clear).

Die C Flag kann direkt mit den Befehlen SEC (set carry) und CLC (clear carry) gesetzt oder gelöscht werden. Wenn wir uns mit der Addition und Subtraktion zu beschäftigen beginnen, werden wir diese Befehle benutzen.

Wenn Sie das letzte Programm in Kapitel 2 untersuchen, werden Sie sehen, daß man den Befehl BNE durch BCC ersetzen kann. Anstelle von „verzweige zurück, wenn nicht gleich sechs" könnten wir auch schreiben „verzweige zurück, wenn kleiner als sechs". Dieser Befehl würde in jedem Fall gleichbedeutend sein.

N Flag

Auch die N (negative) Flag hat einen unglücklichen Namen erhalten. Sie wäre besser HB (high bit) Flag genannt worden, da Zahlen nur dann positiv oder negativ sind, wenn sie in einer be-

stimmten Weise benutzt werden. Die N Flag wird gesetzt, um damit anzuzeigen, daß einem Register ein Wert zugewiesen wurde, dessen oberes Bit (high bit) gesetzt ist.

Die N Flag ist ähnlich stark beschäftigt wie die Z Flag. Sie ändert sich mit jedem Befehl, der ein Register bearbeitet. Die N Flag wird durch Vergleichsoperationen beeinflußt, aber in diesem Fall ist ihr Zustand für den Programmierer gewöhnlich ohne Bedeutung.

Um die Wirkungsweise der N Flag zu erklären, ist es wichtig, sich mit der Umwandlung von hexadezimal nach binär vertraut zu machen. Setzt beispielsweise der Befehl LDA #$65 die N Flag? Wenn wir in binär umwandeln, ist $65 gleichbedeutend mit »01100101. Jetzt erkennen wir, daß das obere Bit nicht gesetzt ist, was bedeutet, daß nach Laden dieses Wertes die N Flag auf „aus" stehen wird. Ein anderes Beispiel: angenommen, wir sagen LDX #$DA. Hex DA ist binär 11011010. Hier ist das obere Bit eingeschaltet und deshalb auch die N Flag gesetzt.

Wenn die N Flag auf „an" gesetzt ist, verzweigt der Befehl BMI (branch minus) zur angegebenen Adresse, andernfalls wird er ignoriert und der nächstfolgende Befehl ausgeführt. Wenn die N Flag auf „aus" gesetzt ist, verzweigt der Befehl BPL (branch plus).

Ein kurzer Ausflug: Zahlen mit Vorzeichen

Wie kann eine Speicherstelle, von der man gewöhnlich annimmt, daß sie einen Dezimalwert zwischen 0 und 255 enthalten kann, eine negative Zahl enthalten? Es hängt vom Programmierer ab. Er muß entscheiden, ob ein Speicherwert ohne Vorzeichen ist, dann kann er einen Wert zwischen 0 und 255 annehmen, oder ob ein Speicherwert mit Vorzeichen versehen ist, dann kann er Werte zwischen -128 und +127 haben. In beiden Fällen gibt es für die Speicherwerte 256 Unterscheidungsmöglichkeiten. Der Computerspeicher enthält nur Bits, der Programmierer dagegen entscheidet darüber, wie diese Bits im speziellen Fall benutzt werden.

Mathematisch läßt sich das folgendermaßen beschreiben: Zahlen mit Vorzeichen kann man, wenn man möchte, in der Form eines „Zweierkomplement" speichern. Wir können -1 als hex FF, -2 als hex FE und so fort bis -128 als hex 80 speichern. Sie werden festgestellt haben, daß in all diesen Beispielen von Negativzahlen das obere Bit gesetzt ist.

Wir brauchen noch ein wenig mehr Vorstellungskraft. Wenn der Computer mit dem Befehl LDA #$C8 den Dezimalwert 200 in das Register A lädt, wird die N Flag gesetzt und damit scheinbar angedeutet, daß 200 eine negative Zahl ist. Es ist vielleicht bequemer, sich lediglich vorzustellen, daß es sich bei 200 um eine Zahl handelt, deren oberes Bit gesetzt ist. In einem gewissen Sinn könnte 200 auch eine negative Zahl sein, wenn wir sie dazu bestimmen. Wir wollen diese Situation mit Hilfe von Beispielen genauer untersuchen.

Wenn ich dazu aufgefordert würde, von hexadezimal 10 abwärts zu zählen, würde ich sagen $10, $0F, $0E, $0D und so fort bis herunter zu $02, $01 und $00. Wenn ich nun fortfahren müßte, würde ich über $00 mit $FF weiter abwärts zählen. In diesem Fall würde hex FF eindeutig eine Negativzahl darstellen. Weiter würden FE, FD und FC die Zahlen -2, -3 und -4 repräsentieren. Das obere Bit würde bei all diesen „negativen" Zahlen gesetzt sein.

Wir wollen über eine dezimale Analogie sprechen. Angenommen, Sie hätten einen Kassettenrekorder mit einem Bandzählwerk und der Zähler zeige 0025. Wenn Sie den Rekorder um

30 Einheiten zurückspulen, würde es Sie nicht überraschen, einen Wert von 9995 auf dem Zähler zu sehen. Sie würden sogleich verstehen, daß das einer Position von -5 entspricht. Wenn Sie ein Auto mit einem Kilometerstand von 1500 km um 1501 km rückwärts fahren, würden Sie einen Kilometerstand von 99999 erhalten, was gleichbedeutend mit -1 ist (der Autor kennt diesen Fall nicht aus persönlicher Erfahrung. Viele Studenten haben ihm jedoch versichert, daß das stimmt). Basierend auf dem Zehner-System werden in diesen Fällen die negativen Zahlen „Zehnerkomplement" genannt.

V Flag

Wie sollte es anders sein, auch die V (overflow) Flag ist mit einem mißverständlichen Namen getauft. Sie wäre besser SAO (signed arithmetic overflow) Flag genannt worden, da sie nur durch Additions- und Subtraktionsbefehle beeinflußt wird und auch nur dann von Bedeutung ist, wenn die betroffenen Zahlen als Zahlen mit Vorzeichen angesehen werden.

Die V Flag wird bei typischen 650x Codierungen nur von Fall zu Fall benutzt. Viele Maschinenspracheprogramme benutzen überhaupt keine Zahlen mit Vorzeichen. Die typischste Anwendung der V Flag findet sich im Zusammenhang mit einem ziemlich spezialisierten Befehl: BIT (bit test). Bei diesem Befehl zeigt die V Flag den Zustand von Bit 6 der Speicherstelle an, die gerade getestet wird. In diesem Fall arbeiten V und N ähnlich: N repräsentiert das obere Bit, Bit 7, V das nächstfolgende Bit, Bit 6. Der Befehl BIT wird hauptsächlich zum Testen der Eingabe/Ausgabekanäle der IA (interface adaptor) Bausteine verwendet.

Wenn die V Flag auf „an" gesetzt ist, verzweigt der Befehl BVS (branch overflow set) zur angegebenen Adresse, andernfalls wird er ignoriert und der nächstfolgende Befehl ausgeführt. Wenn die V Flag auf „aus" zurückgesetzt ist, verzweigt der Befehl BVC (branch overflow clear).

Man kann die V Flag direkt mit dem Befehl CLV (clear overflow) zurücksetzen. Es gibt keinen äquivalenten Befehl, um diese Flag zu setzen.

Eine spezielle Eigenschaft der V Flag: bei manchen 650x kann die V Flag von der Hardware gesetzt werden. Es gibt an diesem Baustein einen Eingang, der dazu benutzt werden kann, die V Flag durch ein externes logisches Signal zu triggern.

Ein kurzer Ausflug: Der Überlauf

Der Begriff Überlauf (overflow) bedeutet, „das Resultat ist zu groß und kann nicht durch ein Byte dargestellt werden". Wenn ich zum Beispiel 200 und 200 addiere, erhalte ich 400. Dieses Ergebnis paßt jedoch nicht in ein einzelnes Byte. Wenn uns zum Abspeichern dieses Resultats nur ein Byte zur Verfügung steht, sagen wir, daß die Addition einen Überlauf produziert hat. Wir können dann keine sinnvolle Antwort geben.

Wenn wir Zahlen ohne Vorzeichen benutzen, teilt uns die C Flag einen Überlauf mit, bei Zahlen mit Vorzeichen erfüllt V diese Aufgabe. Wir wollen das im nächsten Kapitel noch einmal aufgreifen.

Zusammenfassung der Flags

Eine kleine Tabelle soll uns einen Überblick über die vier testbaren Flags geben.

Flag name	Kurze Bedeutung	Aktivitäts-niveau	Verzweigt „an“	bei: „aus“
Z	Null, gleich	hoch	BEQ	BNE
C	Übertrag, größer/gleich	niedrig	BCS	BCC
N	Negativ, oberes Bit	hoch	BMI	BPL
V	Überlauf Vorzeichenarithmetik	niedrig	BVS	BVC

Das Statusregister

Die eben behandelten Flags können, zusammen mit drei weiteren, im Statusregister (SR) betrachtet werden. Sie werden sich erinnern, daß der Maschinensprachemonitor eine Anzeige des SR ermöglicht. Wenn Sie diese zu lesen verstehen, können Sie den Zustand aller Flags daraus erkennen.

Jede Flag stellt ein Bit innerhalb des Statusregister dar. Hier zeigt sich wieder, daß es zum Erkennen der einzelnen Flags nützlich ist, wenn man die hexadezimale Anzeige leicht übersetzen kann. Die Flags sind innerhalb des Statusregister folgendermaßen angeordnet:

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

Wenn wir die Bits einzeln vom oberen zum unteren betrachten, bedeutet:

N - die N Flag, siehe oben.

V - die V Flag, s.o.

Bit 5 - unbenutzt. Sie werden dieses Bit oft in Stellung „an“ finden.

B - (break) Unterbrechungsanzeige. Wenn ein Interrupt erfolgt, zeigt dieses Bit an, ob die Unterbrechung durch den Befehl BRK ausgelöst wurde oder nicht.

D - Dezimalmodusanzeige. Dieses verändert sich in der Weise, in der die Additions- und Subtraktionsbefehle arbeiten. In Commodore Maschinen wird diese Flag immer „aus“ geschaltet sein. Schalten Sie es nur an, wenn Sie ganz genau wissen, was Sie damit bewirken. Diese Flag kann man mit dem Befehl SED (set decimal) „an“ und mit dem Befehl CLD (clear decimal) “aus“ schalten.

I - (interrupt disable) Interrupt gesperrt. Genauer gesagt schaltet dieses Bit die Aktivität des Hardwareeingangs IRQ (interrupt request) aus. Mehr über dieses Kontrollbit erfahren wir später. Diese Flag kann man mit dem Befehl SEI (set interrupt disable) „an“ und mit dem Befehl CLI (clear interrupt disable) „aus“ schalten.

Z - die Z Flag, s.o.

C - die C Flag, s.o.

Die Flags B, D und I sind in dem Sinne keine prüfbaren Flags, da sie von keinem Verzweigungsbefehl direkt getestet werden. D, die Dezimalmodusanzeige, und I, die Flag für Interruptsperre, kann man als Kontrollflags auffassen. Diese zeigen weniger einen Zustand während des Programmlaufs an, sie kontrollieren vielmehr den Programmlauf selbst.

Wir können einen Wert, der in SR, dem Statusregister, angezeigt wird, untersuchen, um den Zustand der Flags zu bestimmen, besonders den der prüfbaren Flags Z, C, N und V. Wenn wir beispielsweise einen SR Wert von $B1 finden, übersetzen wir in binär %10110001 und wissen, daß die N Flag an, die V Flag aus, die Z Flag aus und die C Flag an ist.

Sie können diese Flags durch Überschreiben der im Maschinensprachemonitor angezeigten Werte verändern. Seien Sie aber vorsichtig, daß Sie nicht zufällig die D oder I Flag setzen.

Eine Bemerkung zum Vergleich

Wenn wir zwei Bytes miteinander vergleichen wollen, müssen wir eine Vergleichsoperation durchführen. Ein Wert muß in einem Register (A, X oder Y), der andere entweder im Speicher abgelegt sein oder es muß sich um einen direkten Wert handeln, den wir in einem Befehl benutzen.

Wir wollen die den Registern entsprechenden Vergleichsbefehle benutzen, CMP für Register A, CPX für das X Register und CPY für das Register Y. Nach dem Vergleichsbefehl können wir einen der folgenden Verzweigungsbefehle anwenden:

BEQ - verzweige, wenn die zwei Bytes gleich sind.
BNE - verzweige, wenn die zwei Bytes nicht gleich sind.
BCS - verzweige, wenn der Wert im Register größer oder gleich dem anderen Wert ist.
BCC - verzweige, wenn der Wert im Register kleiner als der andere Wert ist.

Nach einem Vergleich können wir mehr als einen Verzweigungsbefehl benutzen. Angenommen, unser Programm möchte das Y Register auf einen Wert gleich oder kleiner als 5 testen. Wir können dann codieren

```
CPY #$05
BEQ .. irgendwohin
BCC .. irgendwohin
```

Wir sehen, daß unser Code dann verzweigen wird, wenn der Wert gleich 5 ist (er benutzt BEQ) oder wenn er kleiner als 5 ist (er benutzt BCC). Andernfalls wird er ohne Verzweigung fortfahren. In obigem Fall könnten wir durch folgende Veränderung den Code effizienter machen.

```
CPY #$06
BCC .. irgendwohin
```

Ein wenig gesunder Menschenverstand sagt uns, daß es gleichbedeutend ist, wenn wir eine Zahl darauf testen, ob sie kleiner als 6 ist, oder darauf testen, ob sie kleiner oder gleich 5 ist. Gesunder Menschenverstand ist eines der wertvollsten Dinge beim Programmieren.

Befehle: ein Überblick

In Hinblick auf die Datenregister (A, X und Y) haben wir drei Befehlstypen kennengelernt. Mit diesen können wir folgendes bewirken:

Laden: LDA, LDX, LDY
Speichern: STA, STX, STY
Vergleichen: CMP, CPX, CPY

Bis jetzt haben die Register identische Funktionen und wir können sie alle für irgendeine dieser Funktionen benutzen. Aber neue Befehle tauchen am Horizont auf, die jedem dieser drei Register eine eigene Persönlichkeit verschaffen.

Wir haben schon kennengelernt, daß INX, INY, DEX und DEY für Inkrementiere und Dekrementiere allein auf X und Y beschränkt sind. Darüberhinaus wissen wir, daß X und Y zum Indizieren eingesetzt werden können. Wir werden bald einige der Funktionen des A Registers untersuchen, das aufgrund seiner Fähigkeit Rechenoperationen durchführen zu können, Akkumulator genannt wird.

Wir haben den Befehl JSR gesehen, mit dem wir eine Unterroutine, bestehend aus mehreren Instruktionen, aufrufen können. Wir haben auch RTS benutzt, was soviel bedeutet wie „kehre zum Aufrufpunkt zurück", sogar dann, wenn der aufrufende Befehl im BASIC Programm ist. Wir haben auch den Befehl BRK schon behandelt, der ein Programm stoppt, und zum Maschinensprachemonitor zurückführt. BRK kann man beim Überprüfen von Programmen sinnvoll einsetzen, speziell dadurch, daß wir nach Einsetzen von BRK ein Programm zu jeder Zeit abbrechen können. Das erlaubt die Prüfung, ob ein Programm sich korrekt verhält und die Dinge, die wir geplant haben, ausführt.

Es gibt darüberhinaus acht Verzweigungsbefehle. Diese haben wir schon behandelt. Es gibt jedoch einen weiteren Punkt, den wir uns unbedingt merken müssen. Alle Verzweigungen sind soweit brauchbar, als wir kurze Sprünge von bis zu etwa hundert Speicherstellen damit ausführen können. Solange wir nur kurze Programme schreiben, wird das keine Begrenzung darstellen; wir werden uns aber in Kapitel 5 näher damit beschäftigen.

Logische Operatoren

Drei Befehle führen das aus, was man eine logische Operation nennt. Diese heißen: AND (logisch UND), ORA (logisch ODER) und EOR (Exklusives ODER). Diese Befehle wirken nur auf das Register A.

Mathematiker nennen diese Operationen kommutativ. Damit ist gemeint, daß eine Operation „$3A AND $57" exakt dasselbe Ergebnis bringt wie „$57 AND $37". Die Reihenfolge spielt keine Rolle. Wir benutzen diese Funktionen jedoch oft in einer bestimmten Reihenfolge und wir denken auch in dieser Reihenfolge. Das ist bei der Addition ähnlich, bei der wir von einer „Summe" sprechen, zu der ein „Betrag" addiert wird, um eine „neue Summe" zu bilden. Bei logischen Operationen denken wir oft an einen „Wert", den wir mit einer „Maske" manipulieren, um einen „modifizierten Wert" zu erhalten.

Logische Operatoren wirken so, daß jedes Bit innerhalb eines Byte unabhängig von den anderen Bits bearbeitet wird. Das macht diese Befehle zu einem idealen Instrument, um Bits zu extrahieren oder um bestimmte Bits zu manipulieren ohne dabei andere Bits zu beeinflussen.

Wir wollen zwar auf formale Definitionen achten, aber die folgenden intuitiven Konzepte sind für den Programmierer nützlicher:

AND - schaltet Bits aus
ORA - schaltet Bits an
EOR - invertiert Bits

AND - Logisches UND mit A

AND hat auf jedes Bit im A Register folgende Wirkung:

Original A Bit	Maske	Resultierendes A Bit
0	0	0
1	0	0
0	1	0
1	1	1

Untersuchen Sie zunächst die obere Hälfte der Tabelle. Wenn die Maske 0 enthält, wird das Originalbit in A zu 0 verändert. Die untere Hälfte zeigt, wenn die Maske 1 ist, bleibt das Originalbit unverändert. Daraus folgt, AND kann Bits selektiv ausschalten.

Beispiel: Schalte die Bits 4, 5 und 6 im folgenden Wert aus: $C7

```
Originalwert:     11000111
Maske:      AND   10001111 (hex 8F)
Resultat:         10000111
                    xxx
```

Die mit x markierten Bits wurden ausgeschaltet, während alle anderen Bits unverändert blieben.

ORA - Logisches ODER mit A

Auf jedes Bit im A Register hat ORA folgende Wirkung:

Original A Bit	Maske	Resultierendes A Bit
0	0	0
1	0	1
0	1	1
1	1	1

Untersuchen Sie die obere Hälfte der Tabelle. Wenn die Maske 0 enthält, bleibt das Originalbit in A unverändert, wogegen die untere Hälfte zeigt, daß, wenn die Maske 1 enthält, das Originalbit angeschaltet wird. ORA kann also Bits selektiv anschalten.

Beispiel: Schalte die Bits 4, 5 und 6 im folgenden Wert an: $C7

```
Originalwert:      11000111
Maske:      ORA    01110000 (hex 70)
Resultat:          11110111
                     xxx
```

Die mit x markierten Bits wurden angeschaltet, während alle anderen Bits unverändert blieben.

EOR - Exklusives ODER mit A

Original A Bit	Maske	Resultierendes A Bit
0	0	0
1	0	1
0	1	1
1	1	0

Untersuchen Sie die obere Hälfte der Tabelle. Wenn die Maske 0 enthält, bleibt das Originalbit in A unverändert, wogegen die untere Hälfte zeigt, daß, wenn die Maske 1 enthält, das Originalbit invertiert wird. D.h. 0 wird zu 1 und 1 zu 0. EOR kann also Bits selektiv vertauschen.

Beispiel: Invertiere die Bits 4, 5 und 6 im folgenden Wert: $C7

```
Originalwert:      11000111
Maske:      EOR    01110000 (hex 70)
Resultat:          10110111
                     xxx
```

Die mit x markierten Bits wurden invertiert, während alle anderen Bits unverändert blieben.

Wozu logische Operationen?

Wir benutzen diese drei Befehle (AND, ORA und EOR), um einzelne Bits innerhalb eines Informationsbyte zu verändern oder zu kontrollieren. Der Befehl mag deshalb ungewohnt sein, weil jedes Bit unabhängig von den anderen manipuliert werden kann.

Wenn wir diese Befehle benutzen, beschäftigen wir uns nicht mit Zahlen. Wir arbeiten vielmehr mit jedem einzelnen Bit, indem wir es je nach Wunsch an- oder ausschalten.

Wozu sollen wir aber einzelne Bits an- oder ausschalten? Dafür gibt es mehrere mögliche Gründe. Wir möchten zum Beispiel externe Geräte mit Hilfe des IA (interface adaptor) kontrollieren. Innerhalb der IA Eingabe- Ausgabekanäle könnte jedes der acht Bit ein unterschiedliches Signal kontrollieren. Wir möchten vielleicht eine Kontrolleitung an- oder ausschalten, ohne die anderen Leitungen zu beeinflussen.

Wenn wir die Eingabe eines IA Kanals ansehen, lesen wir oft verschiedene Eingabeleitungen, die in einem Byte gemischt übermittelt wurden. Wenn wir ein spezifisches Bit testen möch-

ten, um zu sehen, ob es an oder aus ist, könnten wir alle anderen Bits mit einem AND Befehl maskieren (d.h. unerwünschte Bits auf 0 schalten). Wenn das überbleibende Bit 0 ist, wird das gesamte Byte danach 0 und außerdem wird die Z Flag gesetzt sein.

Wozu möchten wir Bits vertauschen? Viele „oszillierende" Effekte - Bildschirmblinken oder musikalische Noten - können auf diese Weise hervorgerufen werden.

Letztlich können logische Operatoren bei der Codeübersetzung nutzbringend verwandt werden. Wir haben hier zum Beispiel die Werte für ASCII 5 und binär 5:

```
ASCII    %00110101
binär    %00000101
```

Wir müssen den ASCII Wert für die Ein- oder Ausgabe verwenden. Wir müssen den Binärwert zum Rechnen verwenden, besonders für Addition und Subtraktion. Wie können wir von einem zum anderen gelangen? Einfach dadurch, daß wir Bits herausnehmen (AND) oder Bits einfügen (ORA). Alternativ könnten wir zwar Addition oder Subtraktion benutzen, die logischen Operatoren sind jedoch einfacher.

Eingabe: Die GETIN-Unterroutine

Wir haben schon kennengelernt, wie wir CHROUT bei $FFD2 benutzen können, um eine Bildschirmausgabe zu erzeugen. Wir wollen uns jetzt der Eingabeseite zuwenden: wie wir also die GETIN-Unterroutine bei $FFE4 dazu benutzen können, um Zeichen aus dem Tastaturpuffer zu übernehmen.

Vielleicht sind Sie mit der BASIC Anweisung GET vertraut. Wenn ja, werden Sie bei GETIN die gleichen Eigenschaften finden:

- Die Eingabe wird aus dem Tastatur-, nicht aus dem Bildschirmpuffer geholt.
- Wenn eine Taste gehalten wird, wird sie nur einmal abgefragt.
- Die Unterroutine kehrt sofort zurück.
- Wenn kein Tastendruck festgestellt wurde, wird binär 0 in A abgelegt.
- Wurde eine Taste gedrückt, wird sein ASCII Wert in A abgelegt.
- Spezialtasten, wie RETURN, RVS oder Farbcodes werden erkannt.

Um eine Taste auf der Tastatur abzufragen, codieren Sie JSR $FFE4. Die Werte in X und Y bleiben nicht immer erhalten. Deshalb sollte man diese vorher im Speicher ablegen, falls sich wichtige Informationen darin befinden.

Unterroutine: GETIN
Adresse: $FFE4
Wirkung: Nimmt ein Zeichen aus dem Eingabekanal und setzt es in das A Register. Der Eingabekanal ist solange der Tastatureingabepuffer, bis er über besondere Anweisungen umgeschaltet wird.

Das empfangene Zeichen ist gewöhnlich ein ASCII Zeichen (oder PET ASCII). Wenn von der Tastatur gelesen wird, gleicht die Wirkung der einer BASIC GET Anweisung: ein Zeichen wird

aus dem Puffer geholt, aber nicht auf dem Bildschirm angezeigt. Wenn kein Zeichen im Tastatureingabepuffer verfügbar ist, wird der Wert binär 0 in das A Register gesetzt. Die Unterroutine wartet nicht auf einen Tastendruck, sondern kehrt immer sofort zurück.

Register: Das A Register wird natürlich immer beeinflußt. X und Y können verändert werden. Bei Aufruf von GETIN sollte man keine Daten in diesen beiden Registern haben.

Status: Die Statusflags können sich verändern.

Wenn wir die Tastatureingabe auf dem Bildschirm anzeigen wollen, müssen wir dem Aufruf von GETIN, $FFE4 einen Aufruf von CHROUT, $FFD2 folgen lassen, damit das empfangene Zeichen ausgedruckt wird.

Stop

Maschinenspracheprogramme ignorieren die Taste RUN/STOP, es sei denn, das Programm selbst prüft diese Taste. Man kann das durch einen Aufruf der STOP-Unterroutine bei Adresse $FFE1 erreichen. Diese überprüft die RUN/STOP Taste im Augenblick des Aufrufs. Um die Taste richtig zur Wirkung zu bringen, muß $FFE1 wiederholt aufgerufen werden. Dem Aufruf von FFE1 sollte ein BEQ zu einem Programmausgang folgen, so daß das Programm bei Drücken von RUN/STOP endet.

Die RUN/STOP Taste spielt beim Test von Programmen eine wichtige Rolle, da man damit ein Programm auch dann noch beenden kann, wenn dieses sich selbst „festgefressen" hat. Wenn der Programmtest erfolgreich abgeschlossen ist, wird der Teil des Programms, der auf RUN/STOP untersucht, oft wieder aus dem Programm herausgenommen, da man davon ausgeht, daß niemand ein perfektes Programm anhalten möchte. Wenn Sie andererseits davon überzeugt sind, nur hundertprozentig perfekte Programme zu schreiben, werden Sie diese Unterroutine kaum brauchen.

Unterroutine: STOP
Adresse: $FFE1

Wirkung: Testet die RUN/STOP Taste., Wenn in diesem Augenblick RUN/STOP gedrückt ist, wird nach Rückkehr aus der Unterroutine die Z Flag gesetzt.

Im PET/CBM kehrt das System nach tippen von RUN/STOP ins BASIC zurück und zeigt auf dem Bildschirm READY. In diesem Fall kehrt es nicht zu dem aufrufenden Maschinenspracheprogramm zurück.

Register: Das A Register wird beeinflußt. X wird nur beeinflußt, wenn RUN/STOP gedrückt wurde.

Status: Z zeigt an, ob RUN/STOP gedrückt wurde.

Programmiervorhaben

Hier ist unsere neue Aufgabe: Wir möchten eine Unterroutine schreiben, die darauf wartet, daß eine Zahlentaste gedrückt wird. Alle anderen Tasten (ausgenommen RUN/STOP) sollen ignoriert werden.

Wenn eine Zahlentaste gedrückt wurde, soll das auf dem Bildschirm angezeigt und dann die Unterroutine beendet werden. Noch etwas: das numerische Zeichen wird von der Tastatur in ASCII kommen, wir möchten es, bevor wir die letzte RTS Anweisung geben, in eine Binärzahl umwandeln. Die letzte Operation hat außer der Übung noch keinen praktischen Nutzen. Wir wollen es aber im nächsten Kapitel weiterverwenden.

Papier und Stift bereit? Auf geht's!

```
.A 033C  JSR $FFE1
```

Zuerst wollen wir die Taste RUN/STOP testen. Halt! Wohin wollen wir gehen, wenn wir feststellen, daß die Taste gedrückt wurde? Natürlich nach RTS. Wir wissen aber noch nicht, wo das liegen wird. Unter diesen Umständen erlauben wir uns eine grobe Abschätzung, die wir später korrigieren können. Schreiben Sie nun folgendes auf

```
.A 033F  BEQ $0351
.A 0341  JSR $FFE4
```

Wir haben jetzt ein Zeichen erhalten und müssen untersuchen, ob es sich dabei um eine gültige Zahl handelt. Der Satz der ASCII Zahlen 0 bis 9 hat die hex Werte $30 bis $39. Wenn also der Wert kleiner als $30 ist, handelt es sich nicht um eine Zahl. Wie können wir sagen „kleiner als"? Nach einem Vergleich verzweigen wir mit BCC (branch carry clear). Wir schreiben

```
.A 0344  CMP #$30
.A 0346  BCC $033C
```

Ist Ihnen die Benutzung der direkten Adressierungart bei $0344 aufgefallen? Vergewissern Sie sich, daß Sie die Logik verstanden haben. Etwas anderes: was passiert, wenn keine Taste gedrückt wurde? Wir haben es richtig gemacht. Dann befindet sich im A Register eine 0, die natürlich kleiner als hex 30 ist. Das wird uns zu einer neuen Tastaturabfrage zurückbringen.

Nun zum oberen Ende. Wenn die Zahl größer als hex 39 ist, müssen wir sie auch zurückweisen, da es sich dann in ASCII nicht um eine Zahl handeln kann. Instinktiv würden wir CMP #$39 und BCS codieren. Einen Augenblick noch! BCS (branch carry set) bedeutet „Verzweige, wenn größer oder gleich als". Unser vorgeschlagener Code würde die Zahl 9 auch zurückweisen, da bei einem Vergleich mit dem Wert hex 39 die C Flag gesetzt würde.

Wir müßten gegen einen Wert, der größer als $39 ist, testen. Denken Sie aber daran, wir sind im Hexadezimalsystem und der nächste Wert lautet $3A. Wir codieren folgendermaßen

```
.A 0348  CMP #$3A
.A 034A  BCS $033C
```

Bis hierher müßten wir ein ASCII Zeichen zwischen 0 und 9 erhalten können. Wir wollen es auf

den Bildschirm schreiben, damit der Benutzer eine visuelle Kontrolle darüber hat, daß die richtige Taste gedrückt wurde:

```
.A 034C  JSR $FFD2
```

Nun zum Ende unserer Aufgabe. Wir sollten das ASCII Zeichen in ein Binärzeichen verwandeln. Das erreichen wir dadurch, daß wir die oberen Bits ausschalten. Natürlich erinnern wir uns daran, daß wir zum Ausschalten von Bits AND benutzen müssen:

```
.A 034F  AND #$0F
.A 0351  RTS
```

Es war schon richtig, daß wir zuerst das Zeichen ausgedruckt und dann in ein Binärzeichen umgewandelt haben. Das Zeichen muß, um korrekt ausgedruckt zu werden, ein ASCII Zeichen sein.

Noch etwas zum Schluß. Wir haben eine Verzweigung (auf die RUN/STOP Taste), die noch mit RTS verknüpft werden muß. Hatten Sie sich gemerkt, daß Sie darauf noch einmal zurückkommen müssen? Jetzt ist dazu der richtige Zeitpunkt gekommen. Bevor Sie aber darauf eingehen, beenden Sie den Assembler mit einem eigenen RETURN auf der Tastatur (der Assembler wäre andernfalls verwirrt, wenn Sie ihm auf seine Frage nach einer Adresse eine weitere geben würden. Gehen Sie aus dem Assembler heraus, bevor Sie zurückkehren.)

Durch einen unwahrscheinlichen Zufall haben wir für BEQ bei Adresse $033F die richtige Adresse geschätzt. Wäre das nicht der Fall gewesen, wüßten Sie sicher, wie Sie diese ändern könnten. Oder?

Überprüfen Sie Ihr Programm, disassemblieren Sie es, kehren Sie ins BASIC zurück und lassen Sie es mit SYS 828 laufen. Tippen Sie einige Buchstabentasten - nichts sollte passieren. Tippen Sie eine Zahlentaste - und siehe da, sie erscheint auf dem Bildschirm. Das Programm wird enden. Starten Sie es erneut mit SYS und probieren Sie, ob RUN/STOP funktioniert. Versuchen Sie eine Schleife in BASIC, um sich zu vergewissern, daß BASIC und Maschinensprache zusammenarbeiten.

Vorhaben für Enthusiasten: Versuchen Sie das Programm dahingehend zu ändern, daß es nur auf Buchstabenzeichen testet. Buchstaben sind die hex Zeichen von $41 bis einschließlich $5A.

Was wir gelernt haben

- Flags werden dazu benutzt, Befehle miteinander zu koppeln. Dabei kann es sich um Operationen wie Speichern oder Vergleichen handeln, gefolgt von einem Test, der unter bestimmten Bedingungen zu einer Verzweigung führt.
- Einige Befehle beeinflussen eine oder mehrere Flags, andere überhaupt keine. Deshalb sollte einem Befehl, der eine Flag setzt, nicht sofort ein Befehl folgen, der diese Flag testet oder benutzt.
- Es gibt vier prüfbare Flags: Z (zero - 0 oder gleich); C (carry - Übertrag oder größer/gleich); N (negative - Negativ oder oberes Bit) und V (Überlauf bei Rechnen mit Zahlen mit

Vorzeichen). Die Flags werden durch Verzweigungsbefehle wie zum Beispiel BEQ (branch equal) oder BNE (branch not equal) getestet.

- Flags werden im Statusregister (SR) gespeichert, manchmal auch Prozessorstatuswort genannt. Das SR enthält die vier prüfbaren Flags und drei weitere Flags: B (break - Unterbrechungsanzeige); D (Dezimalmodus für Addition und Subtraktion) und I (Interruptsperre). Der hex Wert in SR kann in einen Binärwert umgewandelt und dann dazu benutzt werden, den genauen Zustand aller Flags zu bestimmen.
- Der Prozessor wird normalerweise sechzigmal pro Sekunde unterbrochen, um spezielle vordringliche Aufgaben zu erfüllen. Alles, einschließlich der Flags des Statusregister, wird dabei sorgfältig aufbewahrt, sodaß das Hauptprogramm fortfahren kann, als sei nichts geschehen.
- Eine Zahl, die im Speicher abgelegt ist, kann, wenn wir es bestimmen, als eine Zahl mit Vorzeichen verstanden werden. Der Wert einer Zahl mit Vorzeichen wird in Form eines Zweier-Komplement gespeichert. Das obere Bit der Zahl ist 0, wenn diese positiv, und 1, wenn sie negativ ist. Der Computer kümmert sich darum nicht. Er behandelt die Bits gleich, ob es sich nun um Zahlen mit Vorzeichen handelt oder nicht. Wir dagegen müssen beim Schreiben unseres Programms berücksichtigen, welchen Typ von Zahl wir benutzen.
- Es gibt drei Befehle, die mit logischen Operatoren umgehen: AND, ORA und EOR. Mit diesen können wir Bits innerhalb des A Register selektiv verändern. AND schaltet Bits aus, ORA schaltet Bits an und EOR invertiert Bits oder schaltet sie um.

Fragen und Aufgaben

Schreiben Sie ein Programm, das sowohl Zahlen, als auch Buchstaben akzeptiert, aber nichts anderes.

Schreiben Sie ein Programm, das nur Buchstaben annimmt. Nachdem jedes ASCII Zeichen empfangen wurde, schalten Sie dessen oberes Bit mit ORA #$80 an und geben es auf dem Bildschirm aus. Wie hat sich das Zeichen verändert?

Schreiben Sie ein Programm, das nur einzelne Ziffern annimmt. Nachdem jedes ASCII Zeichen empfangen wurde, schalten Sie dessen unteres Bit mit AND #$FE aus und schreiben es auf den Bildschirm. Was passiert mit den Zahlen? Können Sie erkennen, warum?

4. Zahlen, Arithmetik und Unterroutinen

Dieses Kapitel behandelt:

- Zahlen: mit und ohne Vorzeichen
- Große Zahlen: Mehrfachbytes
- Arithmetik: Addieren und Subtrahieren
- Die Befehle Rotiere und Schiebe
- Multiplikation
- Eigene Unterroutinen

Zahlen: mit und ohne Vorzeichen

Wir haben uns schon kurz mit der Frage von Zahlen mit und ohne Vorzeichen beschäftigt. Am wichtigsten ist, daß Sie als Programmierer bestimmen, ob eine Zahl als Zahl mit Vorzeichen (für ein einzelnes Byte im Dezimalbereich −128 bis +127) oder als eine Zahl ohne Vorzeichen behandelt wird (Einzelbytebereich 0 bis 255).

Für den Computer spielt diese Unterscheidung keine Rolle. Wenn Sie eine Zahl als Zahl mit Vorzeichen betrachten, können Sie das Vorzeichen mit Hilfe der N Flag testen, andernfalls brauchen Sie den Test nicht durchzuführen.

Große Zahlen: Mehrfachbytes

Zur Zahlendarstellung kann man mehr als ein Byte benutzen. Das liegt wiederum in Ihrer Entscheidung. Wenn Sie glauben, daß Zahlen Werte von bis zu einer Million annehmen können, können Sie drei Bytes zur Verfügung stellen (oder mehr oder weniger). Wenn Sie mit Zahlen, die aus mehreren Bytes zusammengesetzt sind, Rechnungen durchführen, hilft Ihnen der Computer dadurch, daß er mit der Carry Flag anzeigt, daß etwas von einem niedrigen Byte zu einem höheren übertragen werden muß. Es liegt jedoch an Ihnen, den Code zu schreiben, der die Extrabytes handhabt.

Zahlen kann man nach folgender Tabelle in Größengruppen einteilen:

	ohne Vorzeichen:	mit Vorzeichen:
1 Byte	0 bis 255	−128 bis +127
2 Bytes	0 bis 65535	−32768 bis +32767
3 Bytes	0 bis 16777215	−8388608 bis +8388607
4 Bytes	bis über 4 Milliarden	−2 Milliarden bis +2 Milliarden

Man kann auch mit Binärbrüchen arbeiten. Das gehört jedoch nicht zum Thema dieses Bu-

ches. Bei vielen Anwendungen werden Zahlen mit einem einheitlichen Maßstab versehen. Zum Beispiel können DM- und Pfennigbeträge als ganzzahlige Größen in Pfennigen gespeichert werden. Demnach könnten zwei Bytes ohne Vorzeichen Werte von bis zu DM 655,35 und drei Bytes bis zu DM 167.772,15 aufnehmen.

Wenn Zahlen mit Vorzeichen in Mehrfachbytes gespeichert sind, dann stellt nur das höchste Bit des höchsten Byte das Vorzeichen dar.

Wir wollen uns hier auf die Grundsätze des Rechnens konzentrieren und deshalb mit Einzelbytes beschäftigen. Wir behandeln Mehrfachbytezahlen als eine Verallgemeinerung dieser Grundsätze am Rande.

Addition

Die Addition wird im Prinzip genauso ausgeführt, wie man es mit Dezimalzahlen gewohnt ist: für die Dezimalstelle muß man jedoch das Byte einsetzen. Wir wollen uns eine einfache Dezimaladdition anschauen:

```
  142856
+ 389217
```

Regel 1: Wir beginnen mit der rechten Spalte (dem niedrigen Byte).

Regel 2: Wir addieren die beiden Werte zusammen mit einem Übertrag aus der vorhergehenden Spalte. Ein neuer Übertrag kann dabei entstehen, der nie größer als eins sein kann. (Der Befehl ADC berücksichtigt jeden Übertrag einer vorangegangenen Stellenaddition und erzeugt ein neues Carry Bit, das entweder 0 oder 1 sein kann).

Regel 3: Wenn wir bei der äußerst rechten Stelle beginnen, gibt es für die erste Addition keinen Übertrag. (Bevor wir eine neue Addition beginnen, müssen wir die Carry Flag mit dem Befehl CLC löschen).

Regel 4: Haben wir die gesamte Addition beendet und einen Übertrag erhalten, jedoch keine Stelle, um diesen aufzunehmen, dann sprechen wir davon, daß das Resultat nicht „paßt". (Wenn eine Additionsfolge von Zahlen ohne Vorzeichen damit endet, daß die Carry Flag gesetzt ist, handelt es sich um einen Überlauf.)

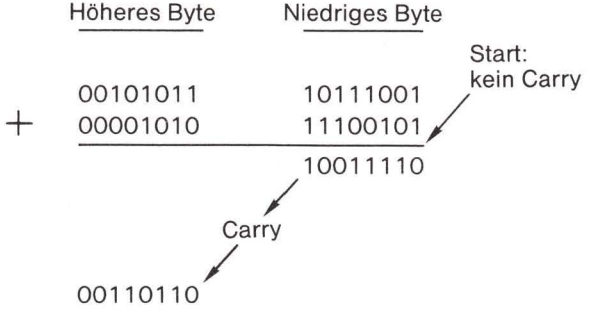


Abbildung 4.1

Wie übersetzen wir diese Additionsregeln in Maschinensprache?

1. Bevor wir eine Additionsfolge beginnen, löschen wir Carry mit CLC .
2. Wenn die Zahlen eine Größe von mehr als 1 Byte haben, beginnen Sie mit dem niedrigen Byte und arbeiten sich zu den höheren vor. Die Addition wird ausschließlich im A Register durchgeführt. Sie könnten den Inhalt einer Adresse oder einen direkten Wert addieren. Die Carry Flag wird jede Form von Übertrag anzeigen.
3. Wenn die Additionsfolge beendet ist, prüfen Sie auf Überlauf:
   a) wenn es sich um Zahlen ohne Vorzeichen handelt, zeigt eine gesetzte C Flag einen Überlauf an;
   b) handelt es sich um Zahlen mit Vorzeichen, zeigt eine gesetzte V Flag den Überlauf an.

Um zwei Zahlen ohne Vorzeichen zu addieren, die sich in Adresse $0380 und $0381 befinden und deren Resultat in $0382 abgespeichert werden soll, können wir folgendes Programm schreiben:

```
CLC
LDA $0380
ADC $0381
STA $0382
```

Wir können nach Belieben auch mit dem Befehl BCS zu einer Fehlerroutine verzweigen.

Wir wollen zwei Zahlen addieren, die aus zwei Bytes bestehen. Die erste Zahl ist in $03A0 (niedriges Byte) und $03A1 (hohes Byte) gespeichert, die zweite Zahl in $03B0 (niedriges Byte) und $03B1 (hohes Byte). Das Resultat soll in $03C0/1 abgelegt werden:

```
CLC
LDA $03A0
ADC $03B0
STA $03C0
LDA $03A1
ADC $03B1
STA $03C1
```

Wir können auch hier wieder mit BCS zu einer Überlauf-Fehlerroutine verzweigen. Hätten wir in denselben Speicherstellen Zweibyte-Zahlen, würde eine Additon auf exakt die gleiche Weise mit dem gleichen Programm erfolgen. In diesem Fall müßten wir allerdings dadurch auf Überlauf testen, daß wir mit dem Befehl BVS zu einer Fehlerroutine verzweigen. Hier hätte die Carry Flag am Ende der Additionsfolge keine Bedeutung.

Subtraktion

Die Subtraktion könnte man als „auf den Kopf gestellte" Addition bezeichnen. Die Carry Flag dient wieder dazu, die Teile einer Mehrbyte Subtraktion miteinander zu verbinden, ihre Rolle ist jedoch umgekehrt. Wenn die Carry Flag bei der Subtraktion benutzt wird, wird sie manchmal auch als „inverted borrow" bezeichnet (umgekehrtes borgen). Bevor wir eine Subtraktion ausführen, müssen wir die C Flag mit SEC setzen. Wenn wir beim Umgang mit Zahlen oh-

ne Vorzeichen einen Überlauf befürchten, achten wir darauf, daß Carry nach Beendigung der Subtraktion gesetzt ist. Wenn Carry null ist, sind Schwierigkeiten aufgetaucht.

Um eine Subtraktion durchzuführen, befolgen wir diese Regeln:

1. Bevor wir eine Subtraktionsfolge beginnen, setzen wir Carry mit SEC.
2. Wenn die Zahlen eine Größe von mehr als 1 Byte haben, beginnen Sie mit dem niedrigen Byte und arbeiten sich zu den höheren vor. Die Subtraktion wird ausschließlich im A Register durchgeführt. Sie könnten den Inhalt einer Adresse oder einen direkten Wert subtrahieren. Die Carry Flag wird jede Form von Übertrag anzeigen.
3. Wenn die Subtraktionsfolge beendet ist, prüfen Sie auf Überlauf:
   a) wenn es sich um Zahlen ohne Vorzeichen handelt, zeigt eine rückgesetzte C Flag einen Überlauf an;
   b) handelt es sich um Zahlen mit Vorzeichen, zeigt eine gesetzte V Flag den Überlauf an.

Um zwei Zahlen ohne Vorzeichen zu subtrahieren, die sich in Adresse $0380 und $0381 befinden, deren Resultat in $0382 abgespeichert werden soll, können wir folgendes Programm schreiben:

```
SEC
LDA $0380
SBC $0381
STA $0382
```

Wir können mit BCC zu einer Fehlerroutine verzweigen.

Zahlenvergleiche

Wenn wir zwei Zahlen ohne Vorzeichen haben und wissen möchten, welche von beiden größer ist, können wir die entsprechenden Vergleichsanweisungen benutzen - CMP, CPX oder CPY - und anschließend die Carry Flag prüfen. Das kennen wir schon von früher. Sind die Zahlen länger als ein Byte, ist das jedoch nicht ganz so einfach. Wir müssen dann eine neue Technik benutzen.

Der einfachste Weg für solch einen Vergleich besteht darin, eine Zahl von der anderen zu subtrahieren. Sie müssen das Resultat nicht aufheben, sondern sich nach Beendigung der Subtraktion nur um die Carry Flag kümmern. Wenn die Carry Flag gesetzt ist, ist die erste Zahl (die, von der Sie abgezogen haben) größer oder gleich der zweiten Zahl. Warum? Weil ein gesetztes Carry anzeigt, daß die Subtraktion von Zahlen ohne Vorzeichen gültig war. Wir haben die beiden Zahlen subtrahiert und ein positives (vorzeichenloses) Resultat erhalten. Wenn andererseits die C Flag zurückgesetzt ist, bedeutet das, daß die erste Zahl kleiner als die zweite ist. Die Subtraktion konnte nicht korrekt durchgeführt werden, da das Resultat - eine negative Zahl - beim Rechnen ohne Vorzeichen nicht dargestellt werden kann.

Linksschieben: Multiplikation mit Zwei

Wenn wir die Dezimalzahlen 100 und 200 in binärer Form schreiben, können wir ein interessantes Muster erkennen:

```
100:  %01100100
200:  %11001000
```

Bei der verdoppelten Zahl ist jedes Bit um eine Position nach links verschoben. Das ist durchaus sinnvoll, da jedes Bit das zweifache numerische „Gewicht" des rechts von ihm stehenden Bit hat.

Der Befehl, ein Byte mit zwei zu multiplizieren, ist gleichbedeutend mit dem Befehl ASL (arithmetic shift left). Ein Bit null wird in die niedrige (oder „rechte") Position des Byte geschoben. Alle Bits bewegen sich um eine Position nach links und das Bit, das aus dem Byte „herausfällt" - in diesem Fall ein Bit null - rückt in die Carry Position. Das kann folgendermaßen dargestellt werden:

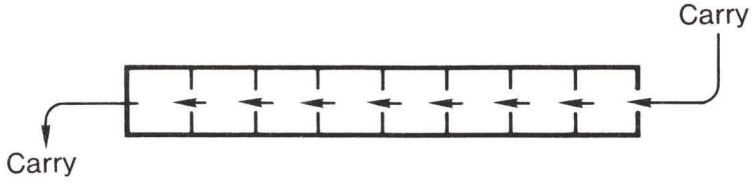
Bei ASL (arithmetic shift left) rückt jedes Bit um eine Position nach links. Eine null rückt in das unterste Bit.

Abbildung 4.2

Das geht sehr einfach, wenn wir den Wert eines einzelnen Byte verdoppeln. Wenn ein „eins" Bit in die Carry Flag rückt, können wir das als Überlauf behandeln. Wie steht es aber mit Mehrfachbytes?

Ideal wäre, wenn wir einen anderen Befehl hätten, der genauso wie ASL wirken würde. Anstatt ein Null Bit auf die rechte Seite eines Byte zu schieben, sollte es das Carry Bit hineinschieben, d.h. das Bit, das bei der letzten Operation „herausgefallen" ist. Es gibt so einen Befehl: ROL.

ROL (rotate left) wirkt exakt wie ASL mit dem einzigen Unterschied, daß das Carry Bit in das nächste Byte geschoben wird. Wir können das wie folgt bildlich darstellen:



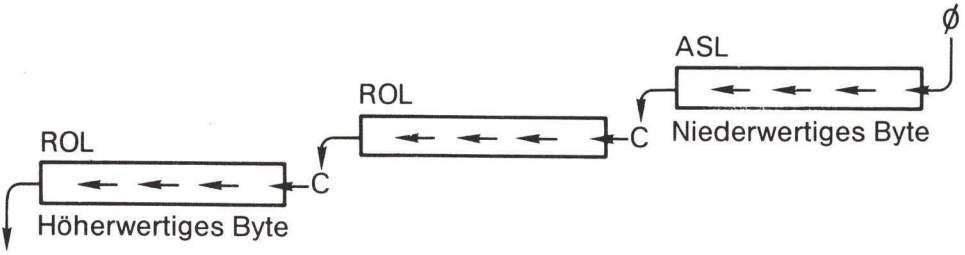
Bei ROL (rotate left) rückt das Carry in das unterste Bit. Jedes Bit rückt um eine Position nach links. Das oberste Bit wird zum neuen Carry.

Abbildung 4.3

Auf diese Weise können wir zwei oder mehr Bytes zusammenkoppeln. Wenn diese eine einzelne Mehrbyte-Zahl enthalten, können wir diese Zahl verdoppeln, indem wir beim niederwertigen Ende beginnen. Wir schieben mit ASL den ersten Wert und mit ROL den Rest. Wenn die Bits aus jedem Byte herausfallen, werden sie im nächsten Byte eingesammelt.

Multiplikation

Die Multiplikation mit zwei scheint uns alleine noch nicht viele Möglichkeiten zu bieten. Wir können jedoch darauf aufbauend die Multiplikation mit jeder beliebigen Zahl durchführen.



Um eine Dreibyte-Zahl mit zwei zu multiplizieren, schieben wir das niederwertige Byte mit ASL. Dann benutzen wir ROL, um mit der C Flag ein Byte mit dem nächsten zu „verbinden".

Abbildung 4.4

Wir wollen uns hier nicht mit einer allgemeinen Multiplikationsroutine beschäftigen, sondern einige spezifische Beispiele zeigen.

Wie können wir mit vier multiplizieren? Durch zweifache Multiplikation mit zwei. Wie können wir mit acht multiplizieren? Durch dreifache Multiplikation mit zwei.

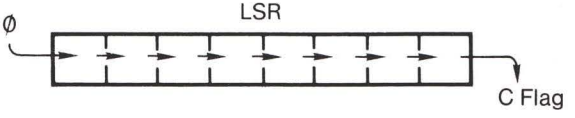
Hier ist ein wichtiges Beispiel. Oft möchten wir mit zehn multiplizieren. Wenn zum Beispiel eine Dezimalzahl auf der Tastatur eingetippt wird, erscheint die Zahl Stelle für Stelle. Nehmen wir die Zahl 217. Das Programm muß nun die 2 annehmen und zur Seite legen; wenn die 1 ankommt, muß die 2 mit zehn multipliziert werden, was zwanzig ergibt, und die 1 addiert werden; wenn die 7 eingetippt wird, muß die 21 mit zehn multipliziert werden, bevor die 7 addiert wird. Ergebnis: 217 in binärer Form. Wir müssen zuerst jedoch wissen, wie wir mit zehn multiplizieren können.

Um mit zehn zu multiplizieren, multiplizieren Sie zuerst mit zwei, dann noch einmal mit zwei. Jetzt haben wir das vierfache der Originalzahl. Addieren Sie nun dazu die Originalzahl und Sie erhalten das fünffache der Originalzahl. Multiplizieren Sie zuletzt mit zwei und Sie haben das Endergebnis. In Kapitel 7 werden wir dafür ein Beispiel sehen.

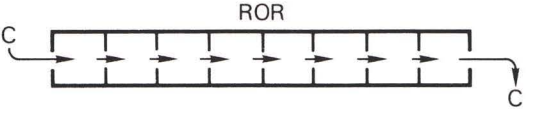
Rechtsschieben und Rotieren: Dividieren durch zwei

Wenn wir durch Schieben (und Rotieren) nach links mit zwei multiplizieren können, dann können wir auch dividieren, indem wir die Bits in der anderen Richtung bewegen. Wenn wir eine Mehrbyte-Zahl haben, müssen wir am oberen Ende beginnen.

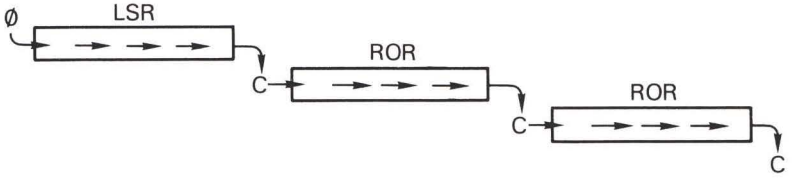
Der Befehl LSR (logical shift right) setzt eine Null in das linke (höherwertige) Bit, bewegt alle Bits nach rechts und schiebt das überfällige Bit in die Carry Flag. ROR (rotate right) schiebt dagegen das Carry Bit in das linke Bit, bewegt alles nach rechts und schiebt das überfällige Bit wiederum in die Carry Flag.



Bei LSR bewegt sich eine Null in das obere Bit und alle Bits bewegen sich eine Position nach rechts. Das unterste Bit wird zum Carry.



Bei ROR bewegt sich Carry in das obere Bit und alle Bits bewegen sich eine Position nach rechts. Das unterste Bit wird zum neuen Carry.



Um eine Dreibyte-Zahl durch zwei zu dividieren, schieben wir das höherwertige Byte mit LSR, dann benutzen wir ROR, um mit der C Flag die übrigen Bytes zu „koppeln".

Abbildung 4.5

Am Ende der Folge von Rechtsverschiebungen kann man das letzte Carry Bit als Rest nach Division durch zwei ansehen.

Kommentare zu Schieben und Rotieren

Wie man von Rechenoperationen erwarten kann, beziehen sich die Schiebe- und Rotierbefehle normalerweise auf das A Register. Sie bieten jedoch eine weitere Anwendungsmöglichkeit: diese Befehle können auch direkt auf den Speicher wirken. Mit anderen Worten, der

Computer kann direkt zu einer Speicheradresse gehen und die Bits dieser Adresse verschieben, ohne vorher die Daten in ein Register zu laden.

Zu diesem Zweck werden Sie oft die Befehle in einer Codierung vorfinden, die den Adreßteil des Befehls, der sich auf das A Register bezieht, mit einer Adresse versieht. Wir würden dann statt LSR A schreiben LSR $1234, um den Inhalt des Speichers zu verschieben.

Wenn ein Rotiere- oder Schiebebefehl direkt auf eine Speicherstelle angewandt wird, werden die Z, N und C Flags dem Speicherinhalt entsprechend beeinflußt. Z wird gesetzt, wenn der Speicherinhalt zu Null wird, N, wenn das obere Bit gesetzt ist, und C spielt seine bekannte Rolle, indem es die überfälligen Bit auffängt.

Einige Programmierer wundern sich vielleicht über die Begriffe logisch (logical) und arithmetisch (arithmetic), die als Teil der Definitionen benutzt werden. Die Unterscheidung hängt damit zusammen, wie Zahlen mit Vorzeichen behandelt werden. „Logisch" bedeutet, daß das Vorzeichen einer Zahl möglicherweise verloren geht, wenn die Zahl als Zahl mit Vorzeichen angesehen wurde. „Arithmetisch" bedeutet, daß das Vorzeichen möglicherweise erhalten bleibt. Das ist eine reine Frage der Terminologie: die Bits selbst bewegen sich exakt gleich.

Unterroutinen

Wir haben bisher Programme geschrieben, bei denen es sich um Unterroutinen handelte, die von BASIC aufgerufen wurden. Wir haben Aufrufe von Unterroutinen geschrieben, bei denen es sich um eingebaute Operationen wie zum Beispiel bei $FFD2 oder $FFE4 handelte. Können wir auch unsere eigenen Unterroutinen schreiben und diese aufrufen?

Natürlich können wir das. RTS (return from subroutine) bedeutet ja nicht „kehre ins BASIC zurück". Vielmehr bedeutet es „kehre dorthin zurück, von wo diese Routine aufgerufen wurde". Wenn BASIC die Maschinenspracheroutine aufrief, bringt Sie RTS zu BASIC zurück. Wenn ein anderes Maschinenspracheprogramm die Unterroutine aufrief, wird RTS Sie zu dessen Aufrufpunkt zurückbringen.

Im letzten Kapitel haben wir eine Unterroutine geschrieben, deren Aufgabe darin bestand, nur Zahlentasten zu akzeptieren, diese auf dem Bildschirm darzustellen und die ASCII Werte in binäre umzuwandeln. Wir wollen diese Unterroutine jetzt benutzen, um ein leistungsfähigeres Programm zu schreiben. Hier ist es. Vergewissern Sie sich, daß es in Ihren Computer eingegeben wurde.

```
.A 033C JSR $FFE1
.A 033F BEQ $0351
.A 0341 JSR $FFE4
.A 0344 CMP #$30
.A 0346 BCC $033C
.A 0348 CMP #$3A
.A 034A BCS $033C
.A 034C JSR $FFD2
.A 034F AND #$0F
.A 0351 RTS
```

Das Vorhaben

Das ist unser Auftrag: Unter Benutzung der obigen Unterroutine wollen wir ein einfaches Additionsprogramm aufbauen. Es soll folgendermaßen arbeiten. Der Benutzer tippt eine Zahlentaste, z. B. „3". Sofort erscheint „3+" auf dem Bildschirm. Nun tippt der Benutzer eine andere Taste, sagen wir „4" und das Programm führt die Addition so aus, daß hinterher auf dem Bildschirm erscheint „3+4=7". Wir gehen davon aus, daß die Summe im Bereich zwischen 0 und 9 liegt, sodaß wir uns nicht um die Ausgabe eines zweistelligen Resultats kümmern müssen - versuchen Sie nicht 5 + 5 einzugeben, sonst erhalten Sie eine falsche Antwort.

Auf geht's. Wir müssen unser Programm bei Adresse $0352 beginnen, um unsere Unterroutine nicht zu zerstören. Wir können es mit SYS 850 aufrufen.

```
.A 0352 JSR $033C
```

Wir rufen damit unsere vorher geschriebene Unterroutine auf. Diese wartet auf eine Zahlentaste, zeigt sie auf dem Bildschirm an, und wandelt diesen Wert in einen Binärwert, der sich im A Register befindet.

Unsere nächste Aktion besteht darin, das Pluszeichen zu drucken. Nachdem wir den ASCII Code für dieses Zeichen nachgeschaut haben, wissen wir, wie wir das bewerkstelligen. Anhang D sagt uns $2B. Wir brauchen also die Befehle LDA #$2B und JSR $FFD2. Einen Augenblick noch! Unser Binärwert befindet sich ja noch im A Register und wir wollen ihn nicht verlieren. Wir speichern diesen Wert also irgendwo ab.

```
.A 0355 STA $03C0
.A 0358 LDA #$2B
.A 035A JSR $FFD2
.A 035D JSR $033C
```

Wir haben $03C0 gewählt, da niemand diese Adresse zu benutzen scheint, und die Binärzahl dort sicher abgelegt. Jetzt drucken wir das Pluszeichen und kehren zurück, um nach einer weiteren Zahl zu fragen.

Wenn die Unterroutine zurückkehrt, hat sie einen neuen Binärwert im A Register. Die Zahl wurde auf dem Bildschirm direkt hinter das Pluszeichen gedruckt. Jetzt müssen wir das Gleichheitszeichen drucken. Aber nicht so schnell, zuerst müssen wir wieder unsere Binärzahl sichern.

Wir könnten den Wert im Speicher ablegen - vielleicht in $03C1 -, es gibt jedoch noch eine andere Möglichkeit. Im Augenblick scheinen wir das Register X oder Y für nichts zu gebrauchen, deshalb wollen wir den Wert in eines der beiden herüberschieben. Uns stehen vier „Transfer"-Befehle zur Verfügung, die Information zwischen A und einem der Indexregister bewegen.

TAX - Transferiere A nach X TAY - Transferiere A nach Y
TXA - Transferiere X nach A TYA - Transferiere Y nach A

Wie bei der Serie von Ladebefehlen stellen diese Instruktionen eine Kopie der Information her. Welche Information auch immer sich in A befand, nach TAX ist sie auch in X. Wie bei den

Ladekommandos werden die Z und N Flags durch die transferierte Information beeinflußt. Ob wir nun X oder Y benutzen, ist gleichwertig, wir wollen X nehmen:

```
.A 0360 TAX
.A 0361 LDA #$3D
.A 0363 JSR $FFD2
```

Wir haben unseren zweiten Wert in X abgelegt und das Gleichheitszeichen ($3D) gedruckt. Jetzt können wir den Wert zurückholen und unsere Addition durchführen. Die beiden folgenden Befehle können in beliebiger Reihenfolge stehen:

```
.A 0366 TXA
.A 0367 CLC
.A 0368 ADC $03C0
```

Die Summe befindet sich im A Register und ist zum Ausdruck fertig, aber sie liegt binär vor und wir brauchen einen ASCII Wert.

Wenn wir annehmen, daß sich die Summe im Bereich zwischen 0 und 9 bewegt, können wir sie direkt mit dem Befehl ORA in eine einzelne ASCII Zahl umwandeln. (Wenn sie größer als 9 ist, mogeln Sie und die Antwort ergibt keinen Sinn.)

```
.A 036B ORA #$30
.A 036D JSR $FFD2
```

Sind Sie ein sehr ordentlicher Mensch? Dann möchten Sie wahrscheinlich ein RETURN drukken, um eine neue Zeile zu beginnen:

```
.A 0370 LDA #$0D
.A 0372 JSR $FFD2
.A 0375 RTS
```

Prüfen Sie alles durch Disassemblieren! Wenn Sie beim Disassemblieren mit der Unterroutine beginnen, werden Sie mehr als einen Bildschirm voller Instruktionen brauchen, um alle zu sehen. Kein Problem. Wenn der Cursor am unteren Bildschirmrand blinkt, tippen Sie den Buchstaben D und RETURN und die Fortsetzung des Listings erscheint.

Zurück in BASIC: jetzt geben wir nicht SYS 828 ein - das ist die Unterroutine und wir wollen die Hauptroutine, nicht wahr?

Geben Sie den Befehl SYS 850 ein. Tippen Sie eine Reihe von Zahlen, deren Summe weniger oder gleich neun ergibt. Die Resultate erscheinen augenblicklich auf dem Bildschirm. Sie können auch eine BASIC Schleife schreiben und die Routine mehrmals aufrufen.

Vorhaben für Enthusiasten: Sie konnten nicht widerstehen. Oder? Sie mußten zwei Zahlen eintippen, deren Resultat über 9 lag und erhielten ein unsinniges Ergebnis. Gut, Ihre Aufgabe besteht darin, das obige Programm dahingehend zu erweitern, daß es auch zweistellige Ergebnisse erlaubt. Das ist nicht allzu schwierig, da die höchstmögliche Summe 9 + 9 oder 18 ist. Wenn also zwei Stellen auftauchen, muß es sich bei der ersten um eine 1 handeln. Sie brauchen nur das Resultat mit binär 9 zu vergleichen, um dann gegebenenfalls die 1 zu drukken und, wenn notwendig, 10 abzuziehen. Das hört sich kinderleicht an.

Was wir gelernt haben

- Wir können entscheiden, ob wir eine Zahl als einen Wert mit Vorzeichen behandeln. In diesem Fall wird das obere Bit der Zahl eine 0 sein, wenn die Zahl positiv ist, und eine 1, wenn sie negativ ist. Es ist unsere Entscheidung, für den Computer bleibt es in jedem Fall nur ein Bit.
- Wenn eine Zahl einen Wert annimmt, der nicht in ein Byte von acht-Bit Länge paßt, können wir mehr als ein Byte benutzen, um diesen Wert zu speichern. Wir haben das bereits getan, als wir Adressen in zwei Bytes speicherten: es gibt ein oberes Byte für den oberen Wert und ein unteres Byte für den unteren Wert.
- Mit dem Befehl ADC können wir zwei Zahlen im A Register addieren. Bevor wir mit der Addition beginnen, sollten wir immer die Carry Flag auf Null setzen. Die Carry Flag kümmert sich für uns um Mehrbyte-Zahlen, wenn wir die Addition richtig am unteren Ende beginnen.
- Mit dem Befehl SBC können wir zwei Zahlen im A Register subtrahieren. Bevor wir mit der Subtraktion beginnen, sollten wir immer die Carry Flag auf Eins setzen. Die Carry Flag kümmert sich für uns um Mehrbyte-Zahlen, wenn wir die Subtraktion richtig am unteren Ende beginnen.
- Bei Zahlen ohne Vorzeichen sollte Carry das gleiche Ergebnis wie zu Beginn enthalten (null bei Addition, eins bei Subtraktion), andernfalls ergab das Resultat einen Überlauf. Bei Zahlen mit Vorzeichen spielt die Carry Flag keine Rolle, die V Flag wird nach einem Überlauf gesetzt.
- Mit dem Befehl ASL (arithmetic shift left) können wir ein Byte mit zwei multiplizieren. Bei einer Mehrbyte-Zahl können wir die Multiplikation zu anderen Bytes fortführen, indem wir den Befehl ROL (rotate left) benutzen und dabei mit dem unteren Byte der Zahl beginnen.
- Mit dem Befehl LSR (logical shift right) können wir ein Byte durch zwei dividieren. Bei einer Mehrbyte-Zahl können wir die Division zu anderen Bytes forttragen, indem wir den Befehl ROR (rotate right) benutzen und dabei mit dem oberen Byte der Zahl beginnen.
- Schiebe- und Rotierbefehle können entweder auf den Inhalt des A Registers oder direkt auf den Speicher angewandt werden. Die N und Z Flags werden beeinflußt, auch die C Flag spielt beim Schieben und Rotieren eine wichtige Rolle.
- Wenn wir mit einem anderen Wert als zwei multiplizieren wollen, ist dies mit größerem Aufwand durchführbar.
- Wie zu erwarten können wir Unterroutinen in Maschinensprache schreiben und diese aus der Maschinensprache heraus aufrufen. Damit können wir ein Programm besser strukturieren.

Fragen und Aufgaben

Schreiben Sie ein Programm, das ähnlich dem der vorigen Übung zwei einstellige Zahlen subtrahiert. Sie können dabei die Unterroutine des vorhergehenden Kapitels weiterverwenden.

Schreiben Sie ein Programm zur Eingabe einer einstelligen Zahl. Ist die Zahl kleiner als fünf,

verdoppeln Sie diese und drucken Sie das Ergebnis aus. Ist die Zahl fünf oder größer, teilen Sie sie durch zwei (unter Vernachlässigung des Restes) und schreiben das Ergebnis auf den Bildschirm. Versuchen Sie, eine saubere Darstellung zu erreichen.

Schreiben Sie ein Programm zur Eingabe einer einstelligen Zahl. Schreiben Sie das Wort GERADE oder UNGERADE hinter die Zahl, je nachdem, ob diese gerade oder ungerade ist. Benutzen Sie den Befehl LSR gefolgt von einem BCC oder BCS Test, um auf gerade oder ungerade zu prüfen.

Wenn Sie der Logik bis hierher gefolgt sind, haben Sie schon einige Fähigkeiten in Maschinensprache entwickelt. Sie beherrschen die Eingabe ebenso wie die Ausgabe und können dazwischen sogar rechnen.

Bis hierher sollten Sie auch schon einige Geschicklichkeit mit dem Maschinensprachemonitor erworben haben und sich bei seiner Benutzung sicherer fühlen. Diese Fähigkeiten sind, wenn auch schnell erlernt, für den Anfänger wichtig. Ohne sie kommen Sie niemals auf angenehme Weise zum Kern der Sache: wie man in Maschinensprache selbst programmiert.

5. Arten der Adressierung

Dieses Kapitel behandelt:

- Implizierte, direkte und Register-Adressierung
- Absolute und Nullseiten-Adressierung
- Indizieren
- Die Relativadresse für Verzweigungen
- Indirekte Adressierung
- Indirekt, indiziert

Arten der Adressierung

Computerbefehle sind in zwei Teile aufgeteilt: der Befehl selbst, auch Opcode genannt, und die Adresse, der Operand. Der Begriff „Adresse" ist ein wenig irreführend, da sich der Operand manchmal auf keine Speicheradresse bezieht.

Der Begriff Adressierungsart bezieht sich auf die Art, in der der Befehl seine Information erhält. In Abhängigkeit davon, wie man diese zählt, gibt es bis zu dreizehn Adressierungsarten, die vom 650x Mikroprozessor benutzt werden können. Man kann sie folgendermaßen zusammenfassen:

1. Keine Speicheradresse: Impliziert, Akkumulator
2. Keine Adresse, sondern ein Wert: Unmittelbar.
3. Eine Adresse, die eine einzelne Speicherstelle bezeichnet: Absolut; Null-Seite.
4. Eine indizierte Adresse, die einen Bereich von 256 Stellen bezeichnet: Absolut, x; Absolut, y; Null-Seite, x; Null-Seite, y.
5. Eine Stelle, an der die wirkliche (Zweibyte) Sprungadresse gefunden werden kann: Indirekt.
6. Ein Offset Wert (z.B. vorwärts 9, rückwärts 17), der für Verzweigungsbefehle benutzt wird: Relativ.
7. Eine Kombination aus indirekten und indizierten Adressen, nützlich um Daten irgendwo im Speicher zu erreichen: Indirekt, indiziert; indiziert, indirekt.

Keine Adresse: Implizierte Adressierung

Befehle, wie zum Beispiel INX (increment X), BRK (break) und TAY (transfer A to Y) benötigen keine Adresse. Sie sprechen keinen Speicher an und sind in sich vollständig. Solche Befehle belegen im Speicher ein Byte.

Wir können auch sagen, daß solche Befehle „keine Adresse" haben. Der präzise Begriff lautet implizierte Adresse. Das bedeutet, es gibt tatsächlich eine Adresse, wir müssen sie jedoch nicht angeben.

Das Wort „impliziert" versteht man vielleicht besser auf folgende Weise: ein Befehl wie INX beinhaltet (impliziert) die Benutzung des Adreßregisters und ein Befehl wie BRK beinhaltet die Adresse des Maschinensprachemonitor. Wenn das so ist, gibt es einen Befehl, der diese Definition nicht erfüllt: NOP.

Der Befehl „Tunichts": NOP

NOP (no operation) ist ein Befehl, der nichts tut. Er beeinflußt weder Datenregister noch Flags. Wenn der Befehl NOP gegeben wird, passiert gar nichts und der Prozessor geht zum nächsten Befehl über. Mir erscheint es unangemessen, davon zu sprechen, daß NOP eine Adresse impliziert. Er tut überhaupt nichts, er hat überhaupt keine Adresse. Andererseits, vermute ich, sagen Logiker: „Ja, aber er hat auf das X Register keine Wirkung".

Der Befehl NOP, dessen Opcode $EA lautet, ist erstaunlich nützlich. Das hat nichts damit zu tun, daß, wenn Sie als angestellter Programmierer nach der Anzahl Ihrer Bytes bezahlt werden, Sie versucht sein sollten, eine große Zahl NOP Befehle in Ihr Programm zu schreiben. NOP kann zwei wichtige Funktionen zum Testen von Programmen erfüllen: Herausnahme unerwünschter Befehle oder Platz schaffen für Extrabefehle.

Ein Maschinenspracheprogramm läßt sich nicht so leicht wie ein BASIC Programm verändern. Wie Sie gesehen haben, werden die Befehle auf spezifischen Stellen abgelegt. Wenn wir einen Befehl herausnehmen wollen, müssen wir entweder alle folgenden Befehl zurückschieben oder die Stelle mit einem NOP Befehl füllen. Wenn wir die Befehle verschieben, müssen wir möglicherweise einige Adressen verändern.

Untersuchen Sie das folgende Programm:

```
0350 LDA #$00
0352 STA $1234
0355 ORA $3456
```

Wenn wir uns entschließen, den Befehl bei 0352 (STA $1234) herauszunehmen, müssen wir alle drei Bytes entfernen. Dazu setzen wir den Code $EA an die Stellen 0352, 0353 und 0354.

Angenommen, wir testen ein ziemlich langes Programm. Die meisten Programme lassen sich in einzelne Module zerlegen, von denen jedes eine spezielle Arbeit ausführt. Ein Modul könnte einen Teil des Speichers auf 0 setzen, ein anderes eine Berechnung durchführen usw. Wenn wir dieses Programm austesten, kann es sich als klug erweisen, den Lauf jedes einzelnen Moduls zu untersuchen.

In diesem Fall können wir einfach ein BRK (break) zwischen jedes Programmodul einfügen. Das Programm läuft an und wird dann zum Maschinensprachemonitor abbrechen. Mit dem Monitor können wir nun den Speicher untersuchen und uns vergewissern, daß dieses Modul seine Arbeit wie geplant ausgeführt hat. Wenn wir zufrieden sind, können wir das nächste Modul mit dem Befehl .G starten. Auf diese Weise läßt sich unser Programm sicher austesten.

Das ist alles schön und gut. Wenn wir unseren Test aber beendet haben und feststellen konnten, daß das Programm zufriedenstellend läuft, brauchen wir unsere BRK Anweisung

nicht mehr. Das läßt sich leicht beheben. Wir ersetzen die BRK Codes ($00) durch NOP ($EA) und das Programm läuft bis zum Ende durch.

Wenn wir hingegen ein Programm schreiben und vermuten, daß wir ein oder zwei weitere Befehle innerhalb eines bestimmten Programmbereichs werden einfügen müssen, setzen wir eine Zahl von NOP Befehlen dort ein. Der Raum bleibt zur Benutzung verfügbar, wenn wir ihn brauchen.

Keine Adresse: Akkumulator-Adressierung

Wir haben gesehen, daß die Schiebe- und Rotierbefehle, ASL, ROL, LSR und ROR eine Datenmanipulation entweder im A Register oder direkt im Speicher ermöglichen. Wollen wir das A Register, den Akkumulator, benutzen, sollten wir diese Tatsache beim Programmieren berücksichtigen. Sie würden zum Beispiel schreiben ASL A.

Wenn man die Akkumulator-Adressierung als Adressierungsart benutzt, ergeben sich die gleichen Eigenschaften, wie bei der implizierten Adressierung: der ganze Befehl paßt in ein Byte.

Wenn sich der Schiebe/Rotier Befehl jedoch auf eine Speicherstelle bezieht, wird man selbstverständlich eine Adresse angeben müssen. Diese Adressierungsart wird später beschrieben.

Neben den Schiebe- und Rotierbefehlen gibt es einen weiteren Satz von Befehlen, die den Speicher direkt manipulieren. Sie werden sich erinnern, daß INX, INY, DEX und DEY ein Indexregister inkrementieren oder dekrementieren.

INC (increment memory) addiert eine 1 zu einer Speicherstelle. DEC (decrement memory) subtrahiert eine 1 von einer Speicherstelle. Beide Befehle beeinflussen die Z und N Flag.

Wenn ein Befehl den Speicher verändert, handelt es sich bei der Adressierungsart weder um implizierte noch um Akkumulator-Adressierung. Die speicherbezogene Adressierung wird später noch diskutiert.

Nicht ganz eine Adresse: Unmittelbare Adressierung

Ein Befehl wie LDA #$34 spricht keine Speicheradresse an. Stattdessen wird ein bestimmter Wert (hier $34) bezeichnet. Ein Befehl mit unmittelbarer Adressierungsart braucht zwei Bytes Speicherplatz: eines für den Opcode und eines für den unmittelbaren Wert.

Wir haben die unmittelbare Adressierungsart einige Male benutzt. Sie wirkt natürlich, ist schnell und bequem. Diese birgt aber auch eine mögliche Gefahr: sie ist so einfach zu benutzen, daß man sie leicht mißbraucht. Jedesmal wenn Sie eine unmittelbare Adressierung wählen, bedenken Sie, ob sich dieser unmittelbare Wert jemals ändern kann. Wenn Sie nämlich einen Wert anstelle einer Variablen in das Programm schreiben, legen Sie diesen Wert für immer fest.

Ein Beispiel: ein Programm wurde für einen VIC-20 geschrieben, der 22 Bildschirmspalten hat. An verschiedenen Stellen dieses Programms werden Werte mit 22 (hex 16) verglichen oder 22 wird zu unterschiedlichen Bildschirmadressen addiert oder davon subtrahiert. In jedem Fall würde die unmittelbare Adressierung verwandt, um den Wert 22 vorzusehen. Der Programmierer entscheidet sich nach einiger Zeit, auf den Commodore 64 umzusteigen, dessen Bildschirm 40 Spalten umfaßt. Der Programmierer muß nun jede unmittelbare Adressierung von 22 auf 40 (hex 28) verändern.

Wäre der Wert 22 in einer Speicherstelle als Variable gespeichert worden, wäre die ganze Umcodierung unnötig. Die Erfahrung zeigt: Übertriebener Gebrauch der unmittelbaren Adressierung kann später zu einem höheren Programmieraufwand führen.

Es gibt bestimmte Befehle, bei denen eine unmittelbare Adressierung nicht möglich ist. Der Befehl LDA #$00 ist möglich: lade den Wert null, nicht den Inhalt der Adresse. Dagegen ist der Befehl STA nicht unmittelbar adressierbar - wir müssen ja die Information irgendwo im Speicher ablegen.

Eine einzelne Adresse: Absolute Adressierung

Ein Befehl kann jede Adresse innerhalb des Speicher wählen - von $0000 bis $FFFF - und eine Information aus dieser Adresse handhaben. Die Angabe der vollen Adresse bezeichnet man als absolute Adressierung. Wenn man so will, man kann absolut mit jeder Information im Speicher umgehen.

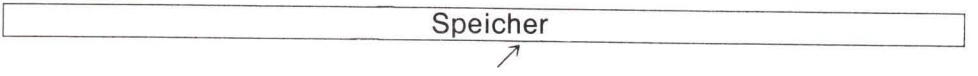


Abbildung 5.1: Die absolute Adressierung bezeichnet eine Adresse irgendwo im Speicher

Wir haben auch absolute Adressen mehrere Male benutzt. Als wir den Inhalt der Speicherstellen $0380 und $0381 ausgetauscht haben, gaben wir diese Adressen an. Als wir einen Wert von der Tastatur speicherten, nannten wir die Stelle $03C0. Wir haben sogar schon absolute Adressen zur Programmkontrolle benutzt: Unterroutinen bei $FFD2 und $33C wurden einfach durch Angabe ihrer Adresse aufgerufen.

Mit der absoluten Adresse ruft ein Befehl JSR (jump subroutine) eine Unterroutine irgendwo im Speicher auf. Es gibt auch einen JMP (jump) Befehl, der die Programmausführung an irgendeine Stelle im Speicher lenken kann. Dieser ähnelt dem BASIC Befehl GOTO. JMP kann mit der absoluten Adressierung benutzt werden - man kann damit irgendwohin springen.

Die absolute Adressierung kennt aber auch eine Einschränkung: Wenn der Befehl geschrieben ist, kann nur die angegebene Adresse erreicht werden. Man kann keinen Adressenbereich erreichen, nur eine einzige Adresse.

Die Adressierung einer einzigen Speicherstelle kann für die unterschiedlichsten Aufgaben nützlich sein. Auf dem PET/CBM können wir durch Manipulation der Adresse 59468 (hex E84C) zwischen Text- und Graphikmodus umschalten. Beim VIC-20 können wir dadurch, daß

wir einen Wert in die Stelle 36878 (hex 900E) setzen, den Lautstärkepegel des Tongenerators verändern. Beim Commodore 64 können wir die Hintergrundfarbe des Bildschirms durch Manipulation der Adresse 53281 (hex D021) verändern. In jedem Fall sprechen wir nur eine spezifische Adresse an. Wir erreichen das durch absolute Adressierung. Wir werden auch die absolute Adressierung einsetzen, um unterschiedliche Speicherstellen des RAM anzusprechen, die wir für unsere eigenen Programmvariablen ausgewählt haben.

Null-Seite Adressierung

Eine Hexadezimaladresse wie $0381 ist sechzehn Bits lang und benötigt zwei Bytes des Speichers. Wir nennen das obere Byte (hier $03) die „Speicherseite" der Adresse. Wir können sagen (tun das gewöhnlich aber nicht), daß diese Adresse auf Seite 3 in Position $81 steht.

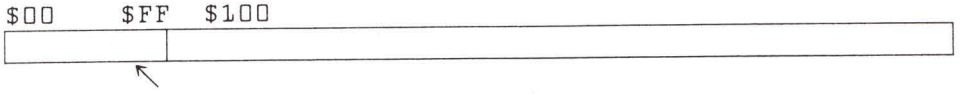


Abbildung 5.2: Die Null-Seite Adressierung bezeichnet eine einzelne Adresse von $00 bis $FF.

Adressen wie $004C und $00F7 liegen auf der Seite 0 (zero-page). Die Null-Seite besteht aus allen Adressen von $0000 bis $00FF. Speicherstellen auf der Seite 0 sind sehr beliebt und werden häufig benutzt. Es gibt eine Adressierungsart, die dazu entworfen wurde, schnell zu diesen Speicherstellen zu gelangen: Null-Seiten Adressierung. Wir können uns dabei eine kurze Adresse vorstellen, wenn wir die ersten beiden Stellen weglassen. Anstatt des Befehls LDA $0090 können wir LDA $90 schreiben. Der neue Code benötigt weniger Platz und läuft etwas schneller ab.

Die Speicherstellen der Seite 0 sind so beliebt, daß wir dort nur schwerlich freie Plätze für unsere eigenen Programme finden können. Auf Commodore Maschinen wollen wir deshalb Speicherstellen der Seite 0 für andere Aufgaben freihalten. Die wenigen, die uns zur Verfügung stehen, werden wir für eine spezielle Adressierungsart einsetzen: Indirekte, indizierte Adressierung. Wir kommen später darauf zurück.

Es gibt viele lesenswerte Stellen auf der Seite 0. Beispielsweise kann man die BASIC Systemvariable ST, die für die Bearbeitung der Eingabe/Ausgabe wichtig ist, dort untersuchen ($96 beim PET/CBM, $90 beim VIC-20 und Commodore 64). Wenn Sie herausfinden möchten, ob ein Benutzer eine Taste betätigt, gibt es auf der Seite 0 eine Adresse, die Ihnen das verrät ($97 beim PET/CBM, $CB beim VIC und 64).

Wie die absolute Adressierung spricht auch die Adressierung der Seite 0 nur eine einzige Speicherstelle an. Das reicht für einen spezifischen Wert aus, für einen ganzen Bereich von Werten brauchen wir etwas mehr.

Ein Bereich von 256 Adressen: Absolute, indizierte Adressierung

Wir haben in Kapitel 2 die Indizierung schon benutzt. Wir geben eine Absolutadresse an und bestimmen dann, daß der Inhalt von X oder Y zu dieser Adresse addiert werden soll, um eine Effektivadresse zu bilden.

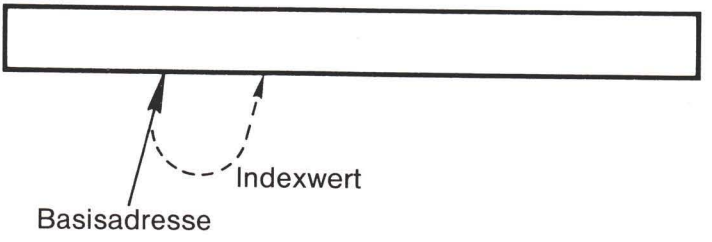


Abbildung 5.3

Die Indizierung wird nur bei der Handhabung von Daten benutzt: sie steht bei Operationen wie Laden und Speichern zur Verfügung, nicht aber bei Verzweigungen und Sprüngen. Bei vielen Befehlen können Sie zwischen dem Indexregister X oder Y wählen. Einige sind auf das X bzw. Y Register beschränkt. Befehle, die X und Y vergleichen oder speichern (CPX, CPY, STX und STY), erlauben keine absolute, indizierte Adressierung. Das trifft auch für den Befehl BIT zu.

Ein Befehl kann mit Hilfe der absoluten, indizierten Adressierung bis zu 256 Speicherstellen erreichen. Die Register X und Y können Werte zwischen 0 und 255 speichern, sodaß die Effektivadresse zwischen der gegebenen Adresse und einer Adresse, die 255 Stellen höher sitzt, liegen kann. Durch Indizierung wird die Adresse immer erhöht. Im Zusammenhang mit einer absoluten Adresse ist ein negativer Index nicht möglich. Liegt die gegebene Adresse oberhalb $FF00, kann ein hoher Indexwert dazu führen, daß die Adresse einmal „umläuft" und eine Effektivadresse im Bereich $0000 erzeugt. In allen anderen Fällen ist die Effektivadresse niemals kleiner als die Befehlsadresse.

Wir haben den Vorteil der Indizierung bereits kennengelernt. Ein Befehl kann sich auf eine bestimmte Adresse beziehen, dann aber, wenn das Programm eine Schleife durchläuft oder wenn der Informationsbedarf sich verändert, kann derselbe Befehl sich auf den Inhalt einer anderen Adresse beziehen. Die maximale Reichweite von 256 Speicherstellen stellt eine wichtige Einschränkung dar.

Die Reichweite eines absolut, indizierten Befehls ermöglicht es ihm, Information in Puffern zu handhaben (Eingabepuffer, Tastaturpuffer, Kassettenpuffer), ebenso in Tabellen (z.B. Tabelle der aktiven Files). Ebenfalls sind kurze Botschaften zugänglich (z.B. HALLO oder Fehlermeldungen). Sie ist jedoch nicht groß genug, um alle Teile des Bildschirmspeichers, alle Teile eines BASIC Programms, geschweige denn das gesamte RAM zu erreichen. Zu diesem Zweck werden wir die indirekte, indizierte Adressierung benutzen, auf die wir noch zu sprechen kommen.

Alles über Seite 0: Null-Seite, indiziert

Die Adressierungsart „Null-Seite, indiziert" scheint auf den ersten Blick dem absolut, indizierten Modus ähnlich zu sein. Zur angegebenen Adresse (in diesem Fall liegt sie auf Seite 0) muß der Inhalt des gewählten Indexregisters addiert werden. Der einzige Unterschied besteht jedoch darin, daß die Effektivadresse die Seite 0 niemals verlassen kann.

Diese Adressierung benutzt gewöhnlich das X Register, zwei Befehle, LDX und STX, benutzen jedoch das Y Register für die Null-Seite, indizierte Adressierung. In jedem Fall wird der Index zur Adresse der Seite 0 addiert. Wenn die Summe die Seite 0 überschreitet, läuft die Adresse einmal um. Ein Beispiel: wenn der Befehl `LDA $E0,X` lautet und das X Register zur Zeit der Ausführung den Wert 50 enthält, wird die effektive Adresse zu $0030. Die Summe ($E0 + $50 oder $130) wird auf die Seite 0 zurückgestutzt.

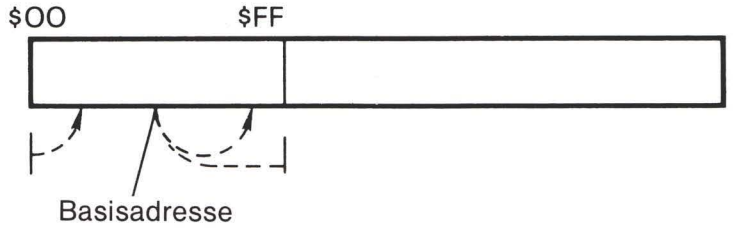


Abbildung 5.4

Jede Adresse der Seite 0 kann demnach durch Indizierung jeden anderen Platz der Seite 0 erreichen. Die Reichweite von 256 Stellen repräsentiert die gesamte Seite 0. Hierdurch eröffnet sich eine neue Möglichkeit: durch Null-Seiten, indizierte Adressierung können wir eine negative Indizierung herbeiführen. Nur für diese Adressierungsart können wir eine Indizierung in Abwärtsrichtung dadurch bekommen, daß wir Indexregisterwerte wie $FF für −1 benutzen, $FE für −2 usw.

Wie schon gesagt, ist die Seite 0 auf Commodore Maschinen äußerst dicht besetzt, sodaß zur Benutzung der Adressierungsart „Null-Seite, indiziert" nur begrenzte Möglichkeiten offen stehen.

Verzweigen: Relative Adressierungsart

Wir haben bereits einige Verzweigungsbefehle geschrieben. Dabei erlaubte uns der Assembler, die aktuelle Adresse, zu der wir verzweigen wollten, einzugeben. Der Assembler übersetzt diese in eine andere Form - die relative Adresse.

Relative Adresse bedeutet, „verzweige von diesem Punkt aus um eine bestimmte Anzahl Bytes vorwärts oder rückwärts". Die relative Adresse ist ein Byte lang, der gesamte Befehl damit zwei Bytes lang. Ihr Wert wird wie eine Zahl mit Vorzeichen behandelt.

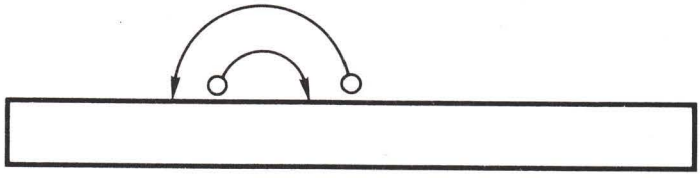


Abbildung 5.5

Ein Verzweigungsbefehl mit einer relativen Adresse von $05 würde bedeuten, „falls die Verzweigung genommen wird, überspringe die nächsten 5 Bytes". Ein Verzweigungsbefehl mit der relativen Adresse $F7 bedeutet hingegen, „wenn die Verzweigung benutzt wird, springe um 9 Bytes rückwärts von der Stelle ab, an der du sonst wärest". Als Zahl mit Vorzeichen ist $F7 gleich dem Wert –9.

Wir können mit hexadezimaler Subtraktion eine Verzweigung berechnen. Die Zieladresse wird von der Befehlszähleradresse subtrahiert. Wenn wir bei $0341 eine Verzweigung nach $033C hätten, würden wir folgendermaßen vorgehen: $033C (das Ziel) minus $0343 (die Stelle, die dem Verzweigungsbefehl folgt) ergibt $F9 oder –7. Diese Rechnung ist mühselig und führt oft zu Fehlern. Fehler bei der Berechnung von Verzweigungsadressen sind für den Programmablauf meist fatal. Wir tun gut daran, diese Berechnung einem Assembler zu überlassen.

Die weitest möglichen Verzweigungen sind: $F7 oder 127 Stellen vorwärts; $80 oder 128 Stellen rückwärts. Das stellt für kurze Programme, wie wir sie hier schreiben, keine Einschränkung dar. Bei langen Programmen könnte die Verzweigung jedoch nicht weit genug reichen. Als Lösung bietet sich an, einen JMP Befehl (jump) in die Nähe zu stellen, der an jeden Platz des Speichers springen kann. JMP benutzt die absolute Adressierung. Der entsprechende Verzweigungsbefehl führt zu JMP, der seinerseits das Programm an die gewünschte Stelle bringt.

Vertreter eines guten Programmierstils bringen folgende Argumente vor. Alle Programme sollten in kleine Einheiten aufgeteilt sein. Logische Blocks sollten in Unterroutinen zerlegt werden, die Unterroutinen ihrerseits in noch kleinere Unterroutinen. Auf diese Weise ist alles überschaubar und leicht zu prüfen. Falls Sie auf eine Verzweigung stoßen, die nicht weit genug reicht, fragen Sie sich, ob es nicht an der Zeit ist, Ihr Programm in kleinere Bausteine aufzuteilen, bevor die Logik zu unübersichtlich wird. Durch großzügigen Gebrauch von Unterroutinen können Sie Ihr Programm so anordnen, daß alle Verzweigungen kurz und damit innerhalb der Reichweite bleiben. Wenn Sie die Programmstruktur unterteilen, reichen die Verzweigungen immer weit genug. Es liegt bei Ihnen, Ihren eigenen Programmierstil zu wählen, Sie sollten diese Anregungen jedoch überdenken.

Ein interessanter Aspekt der relativen Adressierung besteht darin, daß ein Programm, das Verzweigungen enthält, verschiebbar (relocatable) wird. Ein Programmstück, das eine Verzweigung um sechs Stellen vorwärts enthält, arbeitet auch dann einwandfrei, wenn der gesamte Code in einen anderen Speicherbereich verschoben wird. Das trifft jedoch nicht für Sprünge und Aufrufe von Unterroutinen zu oder für jede Form von Code, der die absolute

Adressierung benutzt. Wenn der Speicherbereich sich ändert, muß auch die jeweilige Adresse verändert werden.

Die ROM Verbindung - Sprünge in indirekter Adressierung

Wir haben den Befehl JMP erwähnt, der das Programm zu jeder beliebigen Adresse bringt. JMP besitzt eine weitere Adressierungsart: die indirekte Adressierung.

Indirekte Adressierung wird dadurch angezeigt, daß die Adresse in Klammern gesetzt wird. Sie funktioniert folgendermaßen. Es wird eine Adresse geliefert, es handelt sich dabei jedoch nicht um die, die wir möglicherweise benutzen wollen. Wir nehmen diese Adresse und werden an der Stelle, die diese bezeichnet, die effektive oder indirekte Adresse finden. Diese indirekte Adresse hat natürlich eine Länge von zwei Bytes und wird auf die bekannte 650x-Art gespeichert, das niedrige Byte zuerst.

Ein Beispiel soll das näher erläutern. Angenommen, wir finden bei Adresse $033C den Befehl JMP ($1234). Die Klammern bedeuten, daß es sich um indirekte Adressierung handelt. Der Maschinencode lautet hex 6C 34 12. Wie immer ist die Adresse „umgedreht". Wir gehen nun davon aus, daß in den Adressen $1234 und $1235 die Werte $24 und $68 gespeichert sind. Der Sprungbefehl würde folgendes bewirken: er würde nach $1234 und $1235 gehen, den dortigen Inhalt nehmen und das Programm würde zur Adresse $6824 übergehen.

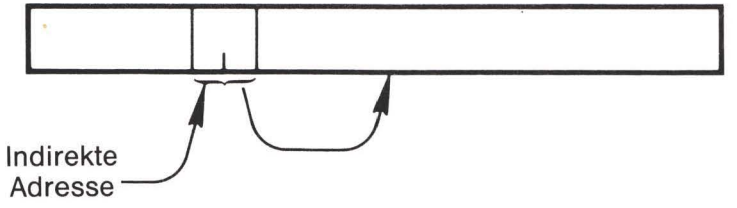


Abbildung 5.6

Der indirekte Sprungbefehl findet eine etwas spezialisierte Anwendung. Wenn wir im Normalfall die Kontrolle auf irgendeine Speicherstelle übertragen wollen, springen wir einfach dorthin. Wir brauchen keine indirekten Schritte. Es gibt hingegen einen ziemlich wichtigen Fall, bei dem indirekte Sprünge eine bedeutende Rolle spielen.

Innerhalb des ROM gibt es eine große Zahl permanenter Befehle, die der Computer zur Durchführung seiner Aufgaben braucht. Da diese im ROM liegen, können wir sie niemals verändern. Würden die verschiedenen Programme ausschließlich durch die Befehle JMP und JSR miteinander verbunden, so könnten auch diese niemals verändert werden, und wir wären nicht in der Lage, die Reaktion der Maschine zu modifizieren.

In das ROM Programm sind eine ganze Reihe sorgfältig geplanter indirekter Sprünge eingebaut. Dadurch bewegt sich das Programm nicht mit direkten Sprüngen nur im ROM, sondern es springt indirekt unter Einschaltung einer Adresse über den RAM Bereich. Den Inhalt des

RAM können wir verändern, und wenn wir eine solche im RAM gespeicherte Adresse verändern, können wir das Verhalten des gesamten Systems beeinflussen. Die bekannteste indirekte Adresse ist diejenige, die mit der Unterbrechungssequenz verbunden ist: sie liegt bei $0090 im PET/CBM und bei $0314 beim VIC, 64 und beim PLUS/4.

Sie werden vielleicht nicht viele indirekte Sprünge programmieren, aber froh sein, daß es diese im ROM gibt.

Daten von überall: Indirekte, indizierte Adressierung

Die Einschränkung der indizierten Adressierung haben wir kennengelernt: die Reichweite von nur 256 Bytes beeinträchtigt die Anwendung dieser Methode.

Die indirekte Adressierung scheint hier die vollkommene Lösung anzubieten. Wir können einen Befehl schreiben, der auf eine indirekte Adresse weist. Da wir die indirekte Adresse nach Belieben verändern oder zu ihr addieren oder von ihr subtrahieren können, können wir mit unserer Anweisung Daten überall im Speicher handhaben.

In Wahrheit ist das wieder mit einer Einschränkung, aber auch mit einer zusätzlichen Möglichkeit verbunden. Zunächst die Einschränkung: bei indirekt, indizierten Befehlen muß die indirekte Adresse auf der Seite 0 stehen - zwei Bytes natürlich, wie immer, das niedere Byte zuerst. Jetzt die zusätzliche Möglichkeit: nach Annahme der indirekten Adresse wird diese mit dem Y Register indiziert und bildet dann die endgültige effektive Adresse.

Wir wollen das einmal Schritt für Schritt verfolgen und die Funktion untersuchen. Angenommen, ich schreibe LDA ($C0),Y mit den Werten $11 in Adresse $00C0 und $22 in Adresse $00C1. Wenn das Y Register den Wert 3 enthält, durchläuft der Befehl folgende Schritte: die Adresse in $00C0-1 wird aufgenommen, wir erhalten $2211. Dann wird der Inhalt von Y addiert und ergibt eine Effektivadresse von $2214. Wenn der Inhalt von Y sich verändert, verändert sich auch die Effektivadresse ein wenig. Wenn dagegen die indirekte Adresse bei $C0 und $C1 verändert wird, verändert sich die Effektivadresse stark.

Die Kombination von indirekt und indiziert kommt einem Overkill gleich. Wenn man jede Speicherstelle mit einer indirekten Adresse ansprechen kann, warum sich dann mit Indizierung herumschlagen? Man kann auch sagen, überall plus eins ist immer noch überall.

Die indirekte Adressierung plus Indizierung scheint eine ideale Kombination zur Bearbeitung von Daten darzustellen. Alle Daten kann man als logische Blöcke irgendeiner Art betrachten: Verzeichnisse, Tabelleneinträge, Bildschirmzeilen, Worte usw. Nun zur Technik. Wir positionieren zunächst die indirekte Adresse auf den Anfang eines gegebenen logischen Datenblocks und benutzen das Y Register, um die Information abzusuchen. Wollen wir zum nächsten Gegenstand übergehen, verschieben wir die indirekte Adresse und wiederholen den gleichen Suchvorgang durch Veränderung des Y Registers.

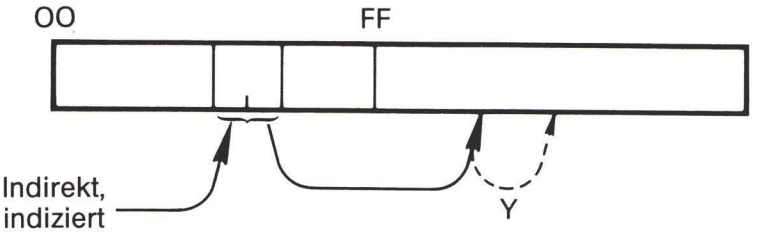


Abbildung 5.7

Wir können diesen Vorgang mit dem Angeln vergleichen: wir ankern unser Boot an einem bestimmten Punkt (dargestellt durch die indirekte Adresse) und benutzen die Angelleine (das Y Register), um die benötigten Daten zu fischen. Wenn wir den nächsten Gegenstand haben wollen, lichten wir den Anker und fahren zu einem neuen Platz.

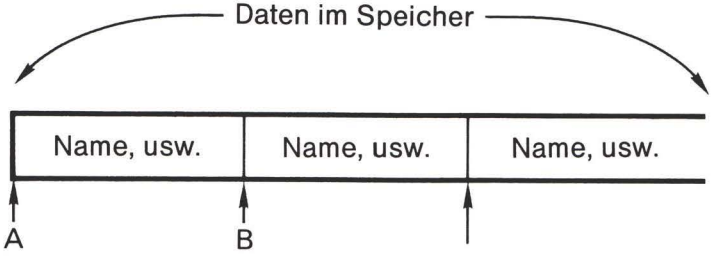
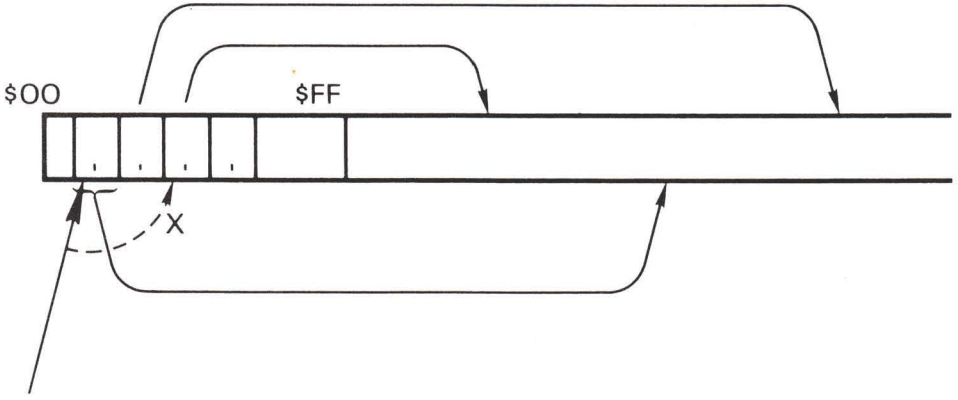


Abbildung 5.8

Wir wollen uns durch ein kompliziertes Beispiel durcharbeiten, das die indirekte, indizierte Adressierung benutzt, um den Bildschirm zu manipulieren. Zunächst jedoch eine kurze Bemerkung.

Eine Seltenheit: Indizierte, indirekte Adressierung

Es gibt eine weitere Adressierungsart, die bei Commodore Computern selten benutzt wird: indiziert, indirekt. Diese benutzt anstelle des Y das X Register und wird wie im folgenden Beispiel codiert: `LDA ($C0,X)`. In diesem Fall findet die Indizierung zuerst statt. Der Inhalt von X wird zur indirekten Adresse addiert (zu $C0), um eine effektive indirekte Adresse zu bilden. Wäre X in unserem Beispiel gleich 4, dann würde die effektive indirekte Adresse $00C4 sein, und der Inhalt von $00C4 und $00C5 würde als effektive Adresse für die Daten benutzt.



Indiziert, indirekt erlaubt mit dem X Indexregister die Auswahl verschiedener indirekter Adressen

Abbildung 5.9

Diese Adressierungsart erweist sich bei bestimmten Arten der Prozeßkontrolle als äußerst nützlich. X soll eine gerade Zahl enthalten. Da jede indirekte Adresse zwei Bytes lang ist, benötigen wir, um von einer zur anderen zu springen, gleichzeitig zwei Bytes.

Nehmen wir ein hypothetisches Kommunikationssystem an, das an vier Telefonleitungen angeschlossen ist, und betrachten, wie die indizierte, indirekte Adressierung benutzt werden kann. Die Zeichen werden von den vier Leitungen immer gleichzeitig empfangen. Jedesmal, wenn ein Zeichen einläuft, muß es in einen Speicherpuffer, der der jeweiligen Leitung zugeordnet ist, abgelegt werden. Auf diese Weise würde der Empfang der verschiedenen Quellen nicht vermischt. Die Seite 0 soll vier indirekte Adressen enthalten, eine für jede Leitung. Jede indirekte Adresse zeigt auf einen Eingabebereich für eine Leitung. Angenommen, ein Zeichen von einer der Leitungen würde im A Register empfangen. Die Leitungsnummer (mal zwei) befindet sich im X Register. Wir könnten nun das Zeichen mit dem Befehl `STA ($60,X)` ablegen. Wäre Leitung 0 beteiligt, würde ihre indirekte Adresse bei Adresse `$60/61` benutzt. Für Leitung 1 wäre es Adresse `$62/63` usw. Nachdem wir das betreffende Zeichen abgespeichert haben, müßten wir den indirekten Zeiger verändern, damit das nächste Zeichen in eine neue Position gebracht wird: wir erreichen das durch `INC $60,X`.

Das obige Beispiel zeigt eine ziemlich spezialisierte Anwendung der indizierten, indirekten Adressierungsart. Sie werden sie vielleicht nie benutzen. Tatsächlich führen die meisten Programmierer ein erfülltes Leben, ohne jemals ein Programm mit indizierter, indirekter Adressierung zu schreiben.

Die große Jagd auf der Null-Seite

Indirekte, indizierte Adressen sind sehr wichtig. Sie stellen die Brücke dar, über die man jeden Teil des Speichers mit einem einzigen Befehl erreichen kann. Sie müssen jedoch zwei Bytes auf der Seite 0 für jede indirekte Adresse, die Sie benutzen wollen, zur Verfügung haben.

Das Commodore ROM System hilft sich selbst durch freizügigen Gebrauch des Speichers auf der Seite 0. Ihnen bleibt dabei nicht viel Platz übrig. Wie können Sie den Platz für diese indirekten Zeiger finden?

Schauen Sie sich zunächst nach unbenutzten Speicherstellen um. Es gibt davon nur wenige: auf dem VIC und Commodore 64 finden Sie diese zwischen $00FC und $00FF. Das reicht für zwei indirekte Adressen.

Wenn Sie mehr benötigen, untersuchen Sie die Speicheraufteilung nach Stellen, die „Arbeitsbereichen" oder „Zeigern auf Benutzerprogramme" zugewiesen sind. Diese können für vorübergehende Aufgaben herangezogen werden.

Schließlich können Sie Arbeitsbereiche der Seite 0 benutzen, nachdem Sie diese zuvor in andere Bereiche des Speichers kopiert haben. Bevor Sie ins BASIC zurückkehren, müssen Sie den Originalinhalt dieses Speicherbereichs sorgfältig zurückspeichern. Versuchen Sie das aber nicht mit irgendwelchen Werten, die von den Interrupt Routinen benutzt werden (solche, die sich mit dem Bildschirm, der Tastatur oder RS-232 beschäftigen). Die Unterbrechung kann und wird ansprechen, während Ihr Maschinenspracheprogramm läuft. Wenn dann das Unterbrechungsprogramm diese Werte auf Seite 0 verändert, ist es um Ihr Programm böse bestellt.

Vorhaben: Bildschirmmanipulationen

Dieses Vorhaben soll die effektive Anwendung der indirekten, indizierten Adressierung zeigen. Wir wollen irgendetwas auf dem Bildschirm verändern, zumindest so viel, daß wir mehr als 256 Adressen erreichen. Gewöhnliche Indizierung würde diese Aufgabe nicht erfüllen können.

Wir wollen eine Anzahl Bildschirmzeilen aussuchen. Innerhalb jeder Zeile wollen wir eine bestimmte Gruppe von Zeichen verändern. Mit anderen Worten, wir wollen ein Programm schreiben, mit dem wir ein „Fenster" auf dem Bildschirm manipulieren.

Um das auszuführen, müssen wir zwei Schritte codieren: Angabe des Beginns der Bildschirmzeile und später Bewegung zur nächsten Zeile, wenn notwendig. Innerhalb jeder Zeile wollen wir uns durch den Bereich der von uns ausgewählten Bildschirmspalten hindurcharbeiten. Dabei handelt es sich um eine große Schleife (für die Zeilen), die eine kleine Schleife (für die Spalten innerhalb dieser Zeile) enthält. Wir wollen die indirekte Adressierung benutzen, um auf den Anfang jeder Zeile zu zeigen, und die Indizierung (das Y Register), um den Teil dieser Zeile, den wir verändern wollen, auszuwählen.

Auf Grund der Vielzahl von Commodore Maschinen müssen wir einige Probleme lösen. Alle Commodore Bildschirme sind speicherorientiert (memory mapped). Das bedeutet, daß die Information, die auf dem Bildschirm erscheint, direkt aus einem Teil des Speichers kopiert wird. Wir können den Bildschirm durch Veränderung des entsprechenden Speicherbereichs verändern. Verschiedene Maschinen benutzen jedoch unterschiedliche Speicheradressen. Darüber hinaus kann beim VIC und beim Commodore 64 der Bildschirmbereich verschoben werden. Weiterhin müssen wir bedenken, daß die Zeilenlänge bei unterschiedlichen Maschinen variiert - 22, 40 oder 80 Spalten kommen vor.

Kein Problem. Wenn Sie eine 40-Spalten Maschine haben, 40 ist gleich $28, codieren Sie

```
.A 033C LDA #$28
```

Für eine 22-Spalten Maschine ändern Sie auf LDA #$16 und für einen 80-Spalten PET codieren Sie LDA #$50.

Haben Sie den richtigen Wert geschrieben? Kommen wir zur nächsten Entscheidung. Beim PET/CBM beginnt der Bildschirmspeicher bei Adresse $8000, beim VIC oder Commodore 64 beginnt er bei der Adresse, die in der Adresse $0288 gespeichert ist. Wir wollen folgendermaßen codieren:

```
PET/CBM:            .A 033E LDX #$80
                    .A 0340 NOP
VIC/Commodore 64:   .A 033E LDX $0288
```

Der Befehl NOP bewirkt nichts, er bringt die Codierung aber auf die gleiche Länge, sodaß wir in jedem Fall mit Adresse $0341 fortfahren können. Das A Register nennt uns unsere Zeilenlänge und das X Register sagt uns die Seitenzahl, auf der der Bildschirm beginnt. Wir wollen diese beiseite legen. Die Zeilenlänge wird später für die Addition gebraucht, weshalb wir sie irgendwo abspeichern können. Die Bildschirmadresse wird Teil einer indirekten Adresse sein. Deshalb gehört sie auf die Seite 0.

Es ist schwierig, eine Adresse auf Seite 0 zu finden, die man bei allen Commodore Maschinen benutzen kann. Wir wählen $00BB und $00BC. $BB enthält natürlich das niederwertige Byte der Adresse. Wir codieren

```
.A 0341 STA $03A0
.A 0344 STX $BC
```

Wie Sie sehen, benutzen wir die Null-Seite Adressierung für den Befehl in Adresse $0344. Damit bringen wir das obere Byte der Adresse an seinen Platz. Jetzt setzen wir das untere Byte auf Null:

```
.A 0346 LDA #$00
.A 0348 STA $BB
```

Unsere indirekte Adresse zeigt jetzt auf den Anfang des Bildschirmspeichers. Wir wollen genauer besprechen, was wir mit dem Bildschirm anstellen möchten. Eine Anzahl von Zeilen soll auf dem Bildschirm verändert werden, sagen wir 14. Wir wollen durch Addition zu der indirekten Adresse unsere indirekte Adresse schrittweise verändern: sei es mit 22, 40 oder 80, je nachdem, was sich in Adresse 03A0 befindet. Es soll nicht die ganze Zeile verändert werden. Beginnen wir bei Spalte 5 und gehen bis Spalte 18. Wir wollen die Zeilen im X Register zählen und beginnen mit X gleich 0:

```
.A 034A LDX #$00
```

Jetzt sind wir für die Bildschirmzeile bereit. Später werden wir die indirekte Adresse anpassen, hierher zurückkehren und eine weitere Zeile bearbeiten. Wir sollten uns merken: „kehre nach $034C für die nächste Bildschirmzeile zurück".

Die indirekte Adresse zeigt auf den Beginn der Zeile. Wir wollen jedoch bei Spalte 5 begin-

nen, was bedeutet, daß Y mit einem Offset von 4 beginnen sollte (Anfang der Zeile plus 4). Also:

```
.A 034C LDY #$04
```

Wir gehen um den Betrag von Y vorwärts und kehren für das nächste Zeichen auf der Zeile in einer Schleife zu diesem Punkt zurück. Merken wir uns: „kehre für das nächste Zeichen nach $034E zurück".

Jetzt sind wir für den nächsten Schritt bereit. Holen wir nun das Zeichen, das im Augenblick dort auf dem Bildschirm steht:

```
.A 034E LDA ($BB),Y
```

Das ist einen Rückblick wert. Die Speicherstellen $BB und $BC enthalten die Adresse des Bildschirm-Speicheranfangs. Auf dem PET/CBM z.B. wäre das $8000. Dazu addieren wir den Inhalt von Y (Wert 4), um die Effektivadresse $8004 zu erzeugen. Aus der Adresse $8004 entnehmen wir das Zeichen aus dem Bildschirm.

Wir entscheiden uns, daß wir Leerstellen unangetastet lassen. Das Zeichen für Leerstelle wird auf dem Bildschirm mit einem Wert von dezimal 32, hex 20, dargestellt. Wir wollen die nächste Operation überspringen, wenn es sich um eine Leerstelle handelt.

```
.A 0350 CMP #$20
.A 0352 BEQ $0356
```

Die Adresse, zu der wir springen wollen, müssen wir schätzen, da wir dort noch nicht angekommen sind. Machen Sie sich eine Notiz: „diese Adresse muß möglicherweise korrigiert werden".

```
.A 0354 EOR #$80
```

Hier manipulieren wir das Zeichen. Das EOR ist ein Umschaltbefehl. Wir schalten das obere Bit des Bildschirmwertes um. Sie können den Bildschirmcode nachschlagen, um herauszufinden, was das bedeutet. An dieser Stelle finden wir den Anschluß zu dem Befehl in $0352. Wir haben wieder Glück gehabt: die Adresse ist genau richtig, um die Verbindung zu $0356 herzustellen. Wäre das nicht der Fall gewesen, wüßten Sie, wie Sie das korrigieren müßten? Verlassen Sie den Assembler, gehen Sie zurück und übertippen Sie.

Wir bringen jetzt das veränderte Zeichen auf den Bildschirm zurück:

```
.A 0356 STA ($BB),Y
```

Wir haben ein Zeichen bearbeitet. Nun wollen wir auf der Zeile zum nächsten Zeichen weiterrücken und sollten, wenn wir die Spalte 18 überschritten haben (Y = 17), abbrechen und zur nächsten Zeile übergehen.

```
.A 0358 INY
.A 0359 CPY #$12
.A 035B BCC $034E
```

Y bewegt sich zur nächsten Zeichenposition weiter: fünf, beim nächsten Mal sechs usw. Solange Y kleiner als 18 (hex 12) ist, kehren wir zurück, da BCC „verzweige, wenn kleiner" bedeu-

tet. Wenn wir diesen Punkt überschreiten, haben wir die Zeile vollendet und müssen zur nächsten überwechseln.

Bewegen wir uns auf die nächste Zeile, indem wir zur indirekten Adresse addieren. Wir müssen 22 oder 40 oder 80 addieren. Der Wert befindet sich in Adresse $03A0 (Sie werden sich erinnern, daß wir ihn mit dem Befehl bei $0341 abgespeichert haben). Wir müssen nun daran denken, daß wir vor dem Additionsbeginn die Carry Flag zurücksetzen und bei der Addition mit dem niederwertigen Byte der Adresse (bei $BB) beginnen.

```
.A 035D CLC
.A 035E LDA $BB
.A 0360 ADC $03A0
.A 0363 STA $BB
.A 0365 LDA $BC
.A 0367 ADC #$00
.A 0369 STA $BC
```

Die letzten drei Befehle scheinen überflüssig. Warum sollen wir zu dem Inhalt von $BC addieren? Sicher, das ändert überhaupt nichts. Mit ein wenig Nachdenken ist uns die Antwort klar: von der vorhergehenden Addition könnte Carry noch gesetzt sein.

Jetzt sind wir zum Zählen der Zeilen bereit. Wir haben X als Zähler gewählt. Zu X wollen wir nun 1 addieren und anschließend prüfen, ob wir 14 Zeilen bearbeitet haben:

```
.A 036B INX
.A 036C CPX #$0E
.A 036E BNE $034C
```

Wir haben die erforderliche Zeilenzahl bearbeitet und müssen nun nichts anderes mehr tun, als ins BASIC zurückkehren:

```
.A 0370 RTS
```

Disassemblieren und überprüfen Sie das Programm. Sie werden wieder feststellen, daß es mehr als eine volle Bildschirmseite belegt. Kehren Sie zum BASIC zurück.

Diesmal wollen wir ein kleines BASIC Programm schreiben, um das Maschinenspracheprogramm auszuprobieren. Tippen Sie NEW, um irgendein altes BASIC Programm zu löschen und geben Sie folgendes ein:

```
100 FOR J=1 TO 10
110 SYS 828
120 FOR K=1 TO 200
130 NEXT K,J
```

Die Zusatzschleife dient der Verzögerung. Die Maschinensprache läuft so schnell, daß der Effekt bei vollem Tempo nicht gut sichtbar würde.

Vorhaben für Enthusiasten: Können Sie das Programm für eine andere Zahl von Spalten verändern? Können Sie es modifizieren, daß nur der Buchstabe „S" verändert würde, wo immer er auch auf dem Bildschirm erscheinen mag?

Kommentar zum VIC-20 und Commodore 64

Diese Übung wird wie vorgesehen ablaufen. Andere Formen der Bildschirmbearbeitung können es notwendig machen, die Werte der Farbnybble des Bildschirmspeicher erst zu setzen, bevor Sie erfolgreich mit dem Bildschirmspeicher arbeiten können. Die Regeln für Maschinensprache unterscheiden sich nicht von denen für BASIC: falls Sie mit POKE den Bildschirm beeinflussen möchten, müssen Sie auch die Farbnybble Bereiche berücksichtigen.

Was wir gelernt haben

- Bei drei Adressierungsarten handelt es sich eigentlich nicht um Adressen. Implizite Adressierung bedeutet, es gibt keine Adresse. Akkumulator Adressierung benutzt das A Register und bedeutet das gleiche. Unmittelbare Adressierung benutzt einen Wert und keine Adresse.
- Absolute Adressen bezeichnen nur eine Adresse irgendwo im Speicher. Null-Seite Adressen bezeichnen eine einzelne Adresse im Bereich $0000 bis $00FF - das obere Byte der Adresse (00) ist die Speicherseite. Diese Adressierungsarten werden für bestimmte Stellen benutzt, die Arbeitswerte oder Systemschnittstellen enthalten.
- Absolut, indiziert und Null-Seite, indiziert erlauben, eine benannte Adresse mit dem Inhalt eines Indexregisters (X oder Y) zu justieren. Diese Befehle können einen Bereich von bis zu 256 Adressen ansprechen. Sie werden im allgemeinen für Datentabellen oder vorübergehende Speicherbereiche benutzt.
- Relative Adressen werden ausschließlich im Zusammenhang mit Verzweigungsbefehlen benutzt. Sie haben eine begrenzte Reichweite von ungefähr 127 Stellen sowohl vorwärts als auch rückwärts. Zur Festlegung der richtigen Werte benötigt man einige Rechenkünste, der Computer führt diese gewöhnlich für uns aus.
- Die indirekte Adressierung wird nur für Sprünge verwandt. Meist erlaubt sie einem festgelegten ROM Programm, einen variablen Sprung durchzuführen. Der durchschnittliche Programmierer in Maschinensprache wird diese selten benutzen, dennoch ist sie es wert, gelernt zu werden.
- Die indirekte, indizierte Adressierung stellt die wichtigste Handhabungsmöglichkeit von Daten überall im Speicher dar. Wir können durch Setzen der indirekten Adresse jede Stelle erreichen und dann diese Adresse durch Indizierung mit dem Inhalt von Y „feinjustieren".
- Indirekte, indizierte Adressierung verlangt, daß die indirekte Adresse auf der Seite 0 liegt. Vor ihrer Benutzung müssen wir die Speicherstellen auf der Seite 0 woanders sicher ablegen.
- Die Adressierungsart mit Namen indiziert, indirekt wird bei der Programmierung von Commodore Computern selten benutzt, steht aber auf Wunsch zur Verfügung.

Fragen und Aufgaben

Schreiben Sie ein Programm, um den Bildschirm Ihres Computers zu löschen - schlagen Sie die Speicherstellen des Bildschirmspeichers, wenn Sie diese vergessen haben sollten, im Anhang C nach. Drucken Sie nicht einfach das Zeichen zum Löschen des Bildschirms ($93), versuchen Sie es auf andere Weise. Können Sie das gesamte Programm schreiben, ohne die indirekte, indizierte Adressierung zu verwenden?

Schreiben Sie das Programm erneut, wobei Sie indirekte, indizierte Adressierung benutzen. Das Programm sollte etwas kürzer sein. Können Sie sich andere Vorteile dieser Schreibweise vorstellen?

Ein Benutzer möchte eine Textzeile auf der Tastatur schreiben, die mit RETURN endet. Er möchte dann, daß das Programm die Zeile zehnmal auf dem Bildschirm wiederholt. Welche Adressierungsart würden Sie benutzen, um den Benutzertext zu handhaben? Warum?

Nehmen Sie eine der vorhergehenden Übungen und versuchen Sie diese ohne direkte Adressierung zu schreiben. Das ist schwierig? Kennen Sie einen Grund, ohne jede direkte Adressierung auszukommen?

6. Kopplung von BASIC und Maschinensprache

Dieses Kapitel behandelt:

- Speicherbereiche für Maschinenspracheprogramme
- Anlage des BASIC Speichers
- Ladevorgang und SOV Zeiger
- BASIC Variable: Ganzzahlig, Fließkomma und Zeichenkette
- Datenaustausch mit BASIC

Ein Platz für das Programm

Bis jetzt haben wir alle Programme im Kassettenpuffer gespeichert. Für kurze Testprogramme ist dieser Platz gut geeignet, wir müssen jedoch nach attraktiveren Alternativen Ausschau halten.

Anlage des BASIC Speichers

Das BASIC RAM ist wie in Abbildung 6.1 organisiert. Die folgenden Bereiche sind für uns von besonderem Interesse:

1. Unterhalb des BASIC Bereichs finden wir den Bereich des Kassettenpuffer. Dieser steht uns dann zur Verfügung, wenn wir ihn nicht für Eingabe oder Ausgabe benötigen.

2. Der Beginn von BASIC (SOB = start of BASIC) befindet sich in der Maschine gewöhnlich auf einer festen Adresse. Beim PET/CBM liegt er bei $0401 (dezimal 1025), beim Commodore 64 ist es $0801 (dezimal 2049) und bei der PLUS/4 Serie liegt er bei $1001 (dezimal 4097). Beim VIC-20 kann er sich auf unterschiedlichen Positionen befinden: $0401, $1001 oder $1201. Ein Zeiger weist auf diese Adresse. Dieser Zeiger befindet sich beim PET/CBM in $28/$29 (dezimal 40 und 41) und beim VIC-20, Commodore 64 und PLUS/4 in $2B/$2C.
   Sie sollten den Zeiger untersuchen und sich vergewissern, daß er die entsprechende Adresse enthält. Wie Sie feststellen werden, läßt sich das mit dem Maschinensprachemonitor viel leichter bewerkstelligen, da die Adresse auf zwei Bytes verteilt ist (das niedrige Byte zuerst, wie immer).

3. Das Ende des BASIC Programms wird durch drei Null-Bytes irgendwo hinter SOB bezeichnet. Wenn Sie in BASIC den Befehl NEW eingeben, werden Sie diese drei Bytes direkt am Anfang von BASIC finden. Es gibt kein Programm, weshalb Anfang und Ende zusammenfallen. Es gibt für das Ende von BASIC nur diese drei Nullen, aber keinen Zeiger. Die nächste Speicherstelle hingegen (SOV) liegt meist direkt hinter dem Ende von BASIC.

Das BASIC Programm, das Sie eintippen, wird den Speicherplatz zwischen diesem Anfang von BASIC und Ende von BASIC belegen. Wenn Sie einem Programm Zeilen hinzufügen, wird, da durch Ihr Programm weiterer Speicher beansprucht wird, das Ende von BASIC nach oben rutschen. Wenn Sie Zeilen herausnehmen, wird sich das Ende von BASIC nach unten verschieben.

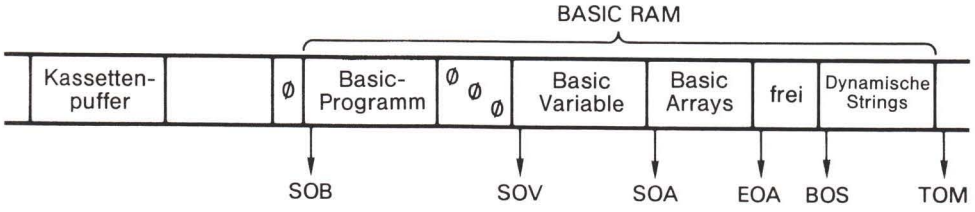


Abbildung 6.1

4. Der Variablenanfang (SOV = start of variables) beginnt oftmals direkt hinter dem Ende von BASIC. Während eines BASIC Programmlaufs werden die Variablen direkt von diesem Punkt an beginnend in den Speicher geschrieben. Jede Variable ist genau sieben Bytes lang. Ein Zeiger markiert diese Speicherstelle. Er liegt beim PET/CBM bei $2A/$2B (dezimal 42 und 43) und beim VIC-20, Commodore 64 und PLUS/4 bei $2D/$2E (dezimal 45 und 46).
   Der SOV Zeiger ist beim Laden und Speichern in BASIC äußerst wichtig. Wenn wir den Befehl SAVE im Direktmodus eingeben, speichert der Computer automatisch die Speicherinhalte zwischen SOB und der Stelle vor SOV. Er speichert also das ganze BASIC Programm einschließlich der drei Null-Bytes, die das Ende von BASIC markieren, aber keine einzige Variable. Wenn wir den Befehl LOAD im Direktmodus eingeben, lädt der Computer das Programm und stellt den SOV Zeiger genau hinter das letzte geladene Byte. Auf diese Weise werden Variable nie über ein BASIC Programm gespeichert, sie werden oberhalb des Endes von BASIC geschrieben. Mehr darüber später.
   Wenn das BASIC Programm verändert wird, bewegt sich SOV nach Bedarf aufwärts oder abwärts.

5. Der Arrayanfang (SOA = start of arrays) stellt ebenfalls eine Speicherstelle jenseits des Endes der BASIC Variablen dar und könnte deshalb auch „Ende der Variablen" genannt werden. Arrays, die von einem BASIC Programm erzeugt werden, sei es durch den Befehl DIM oder durch die bereits vorgegebene Dimensionierung, nehmen von diesem Punkt beginnend den Speicher ein. Er liegt beim PET/CBM bei $2C/$2D (dezimal 44 und 45) und beim VIC-20, Commodore 64 und PLUS/4 bei $2F/$30 (dezimal 47 und 48). Wenn das BASIC Programm verändert wird, wird der SOA Zeiger auf SOV umgesetzt. Das bedeutet, nach jeder Programmänderung sind alle BASIC Variablen gelöscht.

6. Das Arrayende (EOA = end of arrays) steht eine Stelle jenseits von der letzten Array Stelle im BASIC. Oberhalb dieses Punktes scheint der Speicher frei zu sein - wie wir bald sehen werden, ist er nicht wirklich frei. Ein Zeiger markiert diese Speicherstelle. Er liegt beim PET/CBM bei $2E/$2F (dezimal 46 und 47) und bei VIC-20, Commodore 64 und PLUS/4 bei $31/$32 (dezimal 49 und 50).

Bei Veränderung des BASIC Programms wird der EOA Zeiger auf SOA und SOV umgesetzt. Das bedeutet, daß nach jeder Programmänderung alle BASIC Arrays gelöscht sind.

Wir wollen unsere Betrachtung nun umkehren und uns durch den BASIC Speicher von oben nach unten durcharbeiten.

7. Die Obergrenze des Speichers (TOM = top of memory) befindet sich eine Speicherstelle jenseits des letzten für BASIC verfügbaren Byte. Beim PET/CBM und VIC-20 hängt sie davon ab, mit wieviel Speicher diese ausgerüstet sind. Ein 32 K PET würde TOM auf $8000 setzen. Beim Commodore 64 ist TOM normalerweise auf $A000 gestellt. Ein Zeiger markiert diese Speicherstelle. Er befindet sich beim PET/CBM bei $34/$35 (dezimal 52 und 53) und beim VIC-20, Commodore 64 und PLUS/4 bei $37/$38 (dezimal 55 und 56).
   Bei Untersuchung des TOM Zeigers werden Sie vielleicht feststellen, daß er nicht auf die erwartete Position weist. Das kann mit dem Maschinensprachemonitor zusammenhängen, der sich am oberen Speicherbereich niedergelassen und damit etwas vom Speicher weggenommen hat.
8. Der Anfang der Stringvariablen (BOS = bottom of string) wird auf den letzten „dynamischen" string, der erzeugt wurde, gesetzt. Wenn es keine BASIC strings gibt, wird der BOM auf dieselbe Adresse wie TOM gesetzt. Wenn neue Zeichenketten gebildet werden, bewegt sich dieser Zeiger von der Obergrenze des Speichers abwärts, der EOA Adresse zu. Ein Zeiger markiert diese Stelle. Er befindet sich beim PET/CBM bei $30/$31 (dezimal 48 und 49) und beim VIC-20, Commodore 64 und PLUS/4 bei $33/$34 (dezimal 51 und 52).

Eine dynamische Zeichenkette kann nicht direkt von dem Programm, von dem sie definiert wurde, benutzt werden. Sie könnten sie sich vielleicht als eine handgemachte Zeichenkette vorstellen. Wenn ich innerhalb eines BASIC Programms tippe: `100 X$="GUTES NEUES JAHR"`, braucht der BASIC Interpreter die Zeichenkette nicht im oberen Speicherbereich abzuspeichern. Er benutzt die Zeichenkette direkt von der Stelle, an der sie im Programm liegt. Wenn ich andererseits eine Zeichenkette etwa mit den Befehlen `R$=R$+"*"` oder `INPUT N$` definiere, müssen diese Zeichenketten im reservierten Speicherbereich abgespeichert werden. Hierbei kommt der BOS Zeiger ins Spiel: die bearbeitete Zeichenkette wird im oberen Speicherbereich abgelegt und der BOS bewegt sich nach unten, um den nächsten freien Platz zu markieren.

Wenn das BASIC Programm verändert wird, wird der BOS Zeiger auf den TOM umgesetzt. Bei Änderung eines BASIC Programms werden deshalb alle Zeichenketten gelöscht.

Der freie Speicher: ein gefährlicher Bereich

Anfänger haben den Eindruck, daß oberhalb vom Arrayende und unterhalb des Zeichenkettenanfangs eine Menge freien Speichers zur Verfügung steht, und daß das ein idealer Platz für Maschinenspracheprogramme sei. Das ist ein großer Irrtum: gewöhnlich funktioniert das nicht.

Die Gefahr liegt darin, daß die Speicherstelle für den Zeichenkettenanfang sich nach unten bewegt, je mehr dynamische Zeichenketten gebildet werden. Selbst wenn Strings nicht mehr gebraucht werden, werden sie zwar aufgegeben, bleiben aber im Speicher liegen und verbrauchen Platz.

Der BOS bewegt sich weiter nach unten. Erst wenn er den EOA berührt, wird der Speicher von den toten Strings gereinigt. Die guten werden zusammengepackt, was man auch als „Müllabfuhr" (garbage collection) bezeichnet. Es ist für BASIC Programmierer wichtig, über die garbage collection Bescheid zu wissen: mit Ausnahme von BASIC 4.0 und Commodore PLUS/4 Systemen kann diese zu einer ernsthaften Verlangsamung des Programms führen.

Daß der Bereich zwischen EOA und BOS nicht sicher ist, leuchtet ein. Wenn Sie ein Programm dort plazieren, könnte es durch die Strings möglicherweise zerstört werden. Wir müssen uns nach etwas anderem umschauen.

Ein Platz für Ihr ML Programm

Zuallererst sollten Sie Ihr Programm in den Kassettenpuffer legen. Unter der Voraussetzung, daß Sie keine Eingabe/Ausgabe vornehmen, ist Ihr Programm sicher. Der Platz ist hier jedoch auf etwa 190 Zeichen begrenzt.

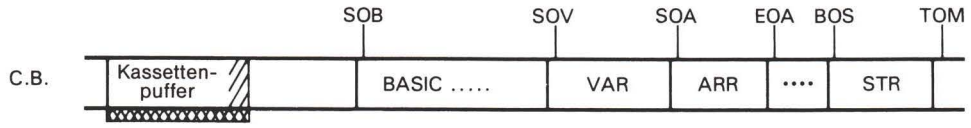


Abbildung 6.2

Als nächstes können Sie den Zeiger für das obere Ende des Speichers nach unten bewegen und das Programm in den Bereich legen, der dadurch freigestellt wurde. Dadurch ist der Platz unbegrenzt. Programme, die hier abgelegt wurden, bleiben bis zum Ausschalten der Netzspannung permanent erhalten. Viele Monitore, wie etwa Supermon, leben hier.

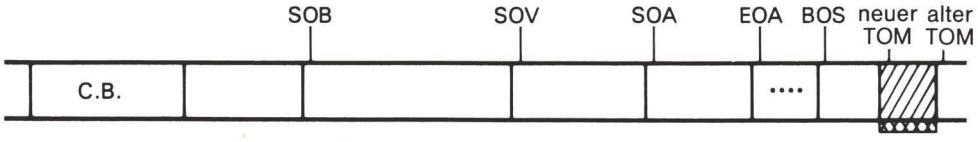


Abbildung 6.3

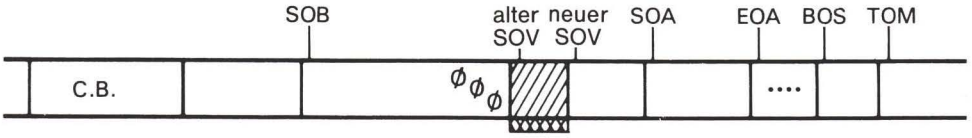


Abbildung 6.4

Als dritte Möglichkeit bietet sich an, den Zeiger für den Variablenbeginn nach oben zu verschieben und das Programm zwischen das BASIC Ende und den neuen Variablenanfang zu legen. Auch hier ist der Platz unbegrenzt. Programme, die dorthin gelegt wurden, verknüpfen sich gewissermaßen mit dem BASIC Programm: beide werden gemeinsam gespeichert und geladen.

Nachdem man die Zeiger wie bei den letzten beiden Methoden verändert hat, sollte man ins BASIC zurückkehren und mit dem Befehl CLR alle anderen Variablenzeiger mit denen in Übereinstimmung bringen, die bewegt wurden.

Diese drei eben genannten Bereiche wollen wir gleich genauer behandeln. Zunächst wollen wir uns jedoch ein oder zwei weiteren Speicherstellen zuwenden, die auf dem VIC-20 und Commodore 64 zur Verfügung stehen.

Extras für VIC und Commodore 64

Der Commodore 64 besitzt einen freien Block von Speicherstellen zwischen $C000 bis $CFFF (dezimal 49152 bis 53247). Das sind unbenutzte 4K RAM. Dorthin kann man seine Programme schreiben. Vorher sollten Sie überprüfen, ob dieser Speicherbereich nicht durch irgendwelche anderen Programme benutzt wird. Beim Commodore 64 handelt es sich um einen beliebten Platz, der von vielen Benutzer- und käuflichen Programmen bevölkert wird.

Falls Sie vorhaben, Maschinenspracheprogramme ohne irgendeine BASIC Routine zu schreiben, können Sie beim Commodore 64 System das BASIC vollständig entfernen und den Platz als verfügbares RAM deklarieren. Damit ist der gesamte Block von $0801 bis $CFFF für Programme und Daten frei - immerhin 94K - wenn nötig, könnte man sogar noch mehr freistellen. BASIC kann man beim Commodore 64 auch mit dem POKE 1,54 (LDA #$36, STA $01) entsprechenden Befehl löschen. Mit dem POKE 1,55 (LDA #$37, STA $01) entsprechenden Befehl läßt es sich wiederherstellen. Seien Sie aber sehr vorsichtig damit. Wenn das BASIC erst verschwunden ist, ist der Computer nicht einmal in der Lage, READY zu sagen.

Bei allen Commodore Maschinen ist es möglich, den Zeiger für den BASIC Anfang nach oben zu verschieben und den darunter liegenden Platz zu nutzen. Um das zu erreichen, ist es unabdingbar, einen Wert Null an die Stelle, die unmittelbar vor dem neuen BASIC Anfang liegt, zu speichern. Dann muß man alle anderen Zeiger neu justieren, gewöhnlich, indem man ins BASIC zurückkehrt und den Befehl NEW eingibt.

Das funktioniert und stellt soviel Platz wie nötig zur Verfügung. Beim Laden werden BASIC Programme automatisch verschoben. Da aber der Computer vor dem Laden des Hauptprogramms neu eingestellt werden muß und nach dem Programmlauf in den ursprünglichen Zustand versetzt werden sollte, ist diese Methode bei den meisten Commodore Maschinen nicht sehr beliebt, wird dagegen beim VIC-20 relativ oft benutzt.

Der Video Chip des VIC-20 kann einen RAM Speicher nur im Bereich $0000 bis $1FFF (dezimal 0 bis 8191) „erkennen". Jegliche Variableninformation, die auf dem Bildschirm erscheint, muß aus diesem Speicherbereich geholt werden. Der VIC-20 kann theoretisch auch Information

aus dem Bereich $8000 bis $9FFF erlangen. Da er jedoch in diesem Bereich nicht über ein RAM verfügt, können wir es auch nicht beeinflussen.

Um besondere Bildeffekte beim VIC-20 zu erzeugen, müssen wir die Daten zwischen $0000 und $1FFF manipulieren. Was steht uns dabei zur Verfügung? $0000 bis $03FF wird vom System benutzt. Im Gegensatz zum Kassettenpuffer sollten wir diesen Bereich nicht antasten. $0400 bis $0FFF steht uns ohne eine 3K RAM-Erweiterung nicht zur Verfügung. $1000 bis $1DFF enthält das BASIC Programm und $1E00 bis $1FFF ist der Bildschirmspeicher. Auch wenn die Einzelheiten variieren, kommt es auf dasselbe hinaus: für Bildeffekte bleibt kein Platz.

Eine beliebte Lösung für den VIC-20, besonders wenn RAM-Erweiterungen von 8K oder mehr hinzugefügt wurden, besteht darin, den Zeiger für den BASIC Anfang zu erhöhen, um im unteren Speicherbereich Raum zu schaffen. Dieser kann dann für Bildeffekte und Maschinenspracheprogramme benutzt werden. Diese Vorgehensweise ist beim VIC-20 notwendig, aber ziemlich schwerfällig.

Der vertrackte SOV

Der Zeiger auf den Variablenanfang kann, wenn man ihn nicht verstanden hat, häufig Schwierigkeiten bereiten. Bei seiner Benutzung sollte man folgende Regeln beachten:

1. Variable werden vom Beginn des SOV ab geschrieben.
2. Der Befehl SAVE in BASIC speichert den Bereich vom Anfang des BASIC an bis zum SOV.
3. Der direkte BASIC Befehl LOAD bringt ein Programm in den Speicher, verschiebt es, falls angebracht, und setzt anschließend den SOV Zeiger auf die Stelle, die dem letzten geladenen Byte folgt.
4. Veränderungen eines BASIC Programms bewirken, daß der Speicher von dem Punkt an, an dem eine Veränderung vorgenommen wurde, bis zum SOV nach oben oder unten verschoben wird. Der SOV ist damit ebenfalls um den entsprechenden Abstand auf- oder abwärts verschoben.

Es scheint sich um unbedeutende Regeln zu handeln. Regel 1 definiert die Rolle des SOV. Regel 2 besagt, wie der SOV den Befehl SAVE kontrolliert, damit nur das gesamte BASIC Programm, nicht aber die Variablen gespeichert werden. Regel 3 sorgt dafür, daß kurze Programme einen großen Speicherbereich für Variable zur Verfügung haben. Lange Programme haben entsprechend weniger. Regel 4 ist dafür verantwortlich, daß Veränderungen im BASIC für zusätzlichen Speicher sorgen oder diesen beanspruchen.

Sollte dem SOV eine falsche Adresse zugeordnet werden, geraten wir in Schwierigkeiten. Die Regeln arbeiten dann gegen uns. Variablen könnten in Bereiche geschrieben werden, in denen sie eine Gefahr darstellen. Durch ein SAVE könnte zuviel oder zu wenig gespeichert werden. Durch LOAD könnten wir festgelegt werden, da durch den Ladevorgang der SOV verändert wird. Bei dem Versuch, ein Programm mit einem schlechten SOV zu verändern, könnte zu wenig oder viel zu viel Speicherplatz bewegt werden. Wir müssen den SOV richtig bestimmen.

Auf welche Weise kann der SOV uns Schwierigkeiten bereiten? Wir wollen das an drei Beispielen erläutern, die den drei wichtigen Plätzen entsprechen, an die wir Maschinenspracheprogramme setzen könnten:

Wir haben ein Programm im Kassettenpuffer zusammen mit einem begleitenden BASIC Programm. Wir laden das BASIC Programm oder geben es ein (der SOV ist bis jetzt in Ordnung), dann laden wir das Maschinenspracheprogramm. Der SOV landet dann irgendwo im Bereich des Kassettenpuffer, was einer Katastrophe gleichkommt.

Wir haben Schwierigkeiten, das Programm kann man anscheinend korrekt listen, es scheint aber krank zu sein. Wenn wir es mit RUN starten, werden die Variablen in den Bereich des Kassettenpuffer gesetzt. Die folgenden Variablen werden fortlaufend in höhere Speicherbereiche abgelegt. Möglicherweise geraten sie dabei in den Bereich des BASIC Programms und überschreiben dieses. Alles kommt zum Stillstand. Gibt der arme Programmierer nun den Befehl LIST ein, um nachzuschauen, was passiert ist, ist sein BASIC Programm verschwunden und der übriggebliebene Rest gleicht einem großen Durcheinander.

Wir können größere Schwierigkeiten bekommen. Der Programmierer entschließt sich nun, mit dem Befehl SAVE sein BASIC Programm zu speichern. Das BASIC beginnt den Speichervorgang am Anfang von BASIC und fährt mit der Speicherung fort und fort und fort.... Es würde bis zum Erreichen des SOV nicht damit aufhören, dieser befindet sich jedoch unterhalb der Stelle, an der der Speichervorgang begann. Wir würden erst dann dorthin gelangen, nachdem die Adresse einmal durch Null „umgelaufen" wäre. Der arme Programmierer - wenn er oder sie nur lange genug wartet - stellt fest, daß das niedliche fünf Zeilen BASIC Programm in über 250 Blöcke auf der Disk gespeichert wurde oder fünfzehn Minuten an Band verbraucht hat. Das gespeicherte Programm ist darüberhinaus nutzlos.

Noch größere Schwierigkeiten sind denkbar. Falls der Programmierer sein Programm auflistet und sich entscheidet, ein Zeichen aus einer BASIC Zeile zu löschen, verschiebt BASIC sofort vom Beginn des Veränderungspunktes ab den Speicherbereich. Es würde damit nicht aufhören, bis es SOV erreicht. Wieder liegt es dann unterhalb der Startposition. Alles, was verschoben werden könnte, wird verschoben. Der RAM Bereich würde verschoben, ohne dabei irgendetwas zu beschädigen. Dann würden die Speicherbereiche der IA Chips verschoben, was zu einem Durcheinander der Farbdarstellung führt oder die ganze Bildschirmdarstellung sinnlos macht. Dann würde versucht, den ROM Bereich zu verschieben, was gar nicht funktionieren kann, da das ROM nicht verändert werden kann. Anschließend findet ein Übergang zur Seite 0 statt, um dort alles zu verschieben, was für das System fatale Folgen hat. Vermutlich findet der Zusammenbruch vor Erreichen von SOV statt, da die eigenen Arbeitszeiger zerstört werden.

All das ist vermeidbar, wenn der Programmierer zuerst das Maschinenspracheprogramm und dann das BASIC Programm lädt. Der SOV würde ans Ende des BASIC Programms gestellt, an den Platz, an den er in diesem Fall gehört.

Kurzes Zwischenspiel

Ein Problem läßt sich leicht erkennen, wenn man den SOV und seine Bedeutung erst einmal verstanden hat. Wenn Sie hingegen den SOV nicht verstanden haben, kann das Ergebnis an Ihrem Selbstvertrauen rütteln. Viele Programmierer haben die Maschinensprache aufgegeben, weil sie mit dem SOV schlechte Erfahrungen gemacht haben.

Das läuft folgendermaßen ab. Ein Programmierer schreibt ein perfektes Programm in den Kassettenpuffer und speichert es mit Hilfe des Maschinensprachemonitor. Später lädt er dieses Programm, nachdem sich ein BASIC Programm im Speicher befand, und verschiebt SOV an eine unmögliche Stelle. Wenn das BASIC läuft, werden die Variablen nacheinander hinter das Maschinenspracheprogramm, aber vor das BASIC Programm gespeichert. Wenn mehr und mehr Variablen vorkommen, kriechen sie unaufhaltsam in das BASIC Programm.

Mit einem perfekten Maschinenspracheprogramm und einem perfekten BASIC Programm entscheidet sich unser ungeduldiger Programmierer zu dem Befehl RUN. Das BASIC Programm läuft auch einige Zeit, um dann plötzlich auf Grund zu laufen, gewöhnlich mit einer verrückten Bildschirmdarstellung oder nach Anzeige eines Fehlers in einer nicht existierenden Zeile. Wir wissen natürlich, was geschehen ist: die Variablen haben sich über das BASIC Programm geschrieben. Unser unglücklicher Programmierer weiß nichts davon. Er gibt den Befehl LIST ein, worauf nur Unsinn erscheint.

Was geht dem Programmierer durch den Kopf? „Ich war so sicher, daß das Programm in Ordnung ist (das ist es auch). Eine Gemeinheit, daß es den Speicher durcheinander bringt! Maschinensprache scheint doch schwieriger zu sein, als ich angenommen habe."

Er gibt entnervt auf, ohne zu wissen, daß er mit ein klein wenig mehr Information in der Lage wäre, alle Probleme zu beseitigen. Das ist nur eine Möglichkeit, wie ein falsch gesetzter SOV Zeiger alles verderben kann. Allein der Versuch, ein BASIC Programm zu verändern oder zu speichern, kann zu einem Systemausfall führen.

Solche Erfahrungen nagen am Selbstbewußtsein. Sie sind verantwortlich für den Mythos, daß Maschinensprache äußerst schwierig ist und nur superclevere Programmierer damit umgehen können.

SAVE beim Maschinensprachemonitor

Nachdem wir die möglichen Fallstricke des SOV kennengelernt haben, können wir uns mit der Frage beschäftigen, wie wir ein Maschinenspracheprogramm speichern können. Wahrscheinlich verstehen Sie, warum ich die Behandlung dieses Befehls bis jetzt zurückgestellt habe. Der typische Speicherbefehl lautet im MLM

```
.S "PROGRAMM",01,033C,0361
```

So sähe das Format für das Band aus. Der Befehl lautet .S gefolgt vom Namen des Programms. Die Speichereinheit ist das Band, weshalb wir 01 schreiben - vergewissern Sie sich, daß Sie beide Ziffern angegeben haben. Als nächstes folgt die Anfangsadresse (in dem Beispiel 033C). Ihr folgt die Endadresse plus eins. In unserem Beispiel ist die letzte gespeicherte Adresse $0360. Falls wir auf Disk speichern, geben wir die Laufwerksnummer zusätzlich an:

```
.S "0:PROGRAMM",08,033C,0361
```

Die einmal gespeicherten Programme können direkt von BASIC geladen werden. Achten Sie jedoch sorgfältig auf den SOV. Der Befehl LOAD sollte beim VIC-20 und Commodore 64 eine zusätzliche Angabe enthalten, um einer Verschiebung des Programms entgegenzuwirken: LOAD "PROGRAMM",8,1 sorgt dafür, daß das Programm in denselben Speicherbereich zurückgeladen wird, aus dem es gespeichert wurde.

Mehr über den Befehl LOAD

Es gibt einen Maschinensprachebefehl .L, mit dem man ein Programm laden kann, ohne irgendwelche Zeiger (besonders den SOV) zu verändern. Man findet eine Zahl unterschiedlicher Maschinensprachemonitore, bei denen dieser Befehl teilweise anders ausgeführt wird. Sie sollten denjenigen, den Sie benutzen, daraufhin überprüfen. Im Idealfall sollte der Befehl (Format:.L "PROGRAMM",01) das Programm ohne Verschiebung in den Speicher zurückbringen.

Der Befehl .L ist von begrenztem Wert. Man sollte von einem Programmbenutzer nicht erwarten, daß er zunächst einen Maschinensprachemonitor laden muß, um dann mit einer Folge von .L Befehlen das eigentliche Programm zu laden. Das Programm selbst sollte sich für den Benutzer darum kümmern.

Wir haben ganz besonders darauf hingewiesen, daß der BASIC Befehl LOAD, wenn er als direktes Kommando eingegeben wird, den SOV verändert. Wenn der Befehl LOAD aus einem Programm heraus gegeben wird, wird SOV nicht verändert. Hier muß man jedoch auf etwas Neues achten.

LOAD von einem Programm ausgeführt, wurde besonders dazu entwickelt, eine Funktion auszuführen, die man als Aneinanderketten (chaining) bezeichnet. Dabei handelt es sich um eine BASIC Technik, die wir in diesem Buch nicht behandeln. Das Aneinanderketten zeichnet sich jedoch durch zwei wichtige Eigenschaften aus:

1. Es werden keine Zeiger beeinflußt. Das Programm verliert keine Variablen, wenn es den Befehl LOAD ausführt. Das ist gut so: wir wollen schließlich unsere Ergebnisse nicht verlieren.
2. Ist der Vorgang LOAD beendet, nimmt das BASIC Programm seine Arbeit beim ersten Befehl wieder auf. Es wird nicht an der Stelle fortfahren, an der der Befehl LOAD steht, sondern zum Anfang zurückkehren. Das ist für unsere Anwendung schlecht: wir scheinen damit unsere Orientierung im BASIC Programm zu verlieren.

Wenn wir das Problem, das unter Punkt 2 auftaucht, verstehen, können wir es leicht durch Anwendung des Punkt 1 umgehen. Das folgende Beispiel soll das Problem illustrieren: wir haben ein Programm auf der Disk, das für den Kassettenpuffer geschrieben ist und "ML" genannt wird. Wir wollen es mit einem BASIC Programm laden. Wir könnten als erste Befehlszeile schreiben: 100 LOAD "ML",8 - das wird uns aber in Schwierigkeiten bringen. Zuerst würde das Programm ML geladen, dann würde es an den Anfang zurückkehren und ML abermals laden. Es würde danach immer wieder an den Anfang zurückkehren. Das ist unbefriedigend. Wir wollen Regel 1 benutzen, um alles in Ordnung zu bringen:

```
100 IF A=1 GOTO 130
110 A=1
120 LOAD "ML",8,1
130 ...Fortsetzung
```

Wenn wir RUN eingeben, wird die erste Zeile ausgeführt. A ist ungleich 1, wir fahren mit Zeile 110 fort. A wird gleich 1 gesetzt und Zeile 120 bewirkt, daß das gewünschte Programm geladen wird. BASIC kehrt an den Anfang zurück, alle Variablen blieben jedoch erhalten, sodaß A gleich 1 geblieben ist. Zeile 100 testet A und geht nach Zeile 130, zu dem auf LOAD folgenden Befehl. Alles arbeitet wunschgemäß. Sollten mehrere LOAD Befehle vorkommen, könnten wir, wenn nötig, Zeile 100 verändern: 100 ON A GOTO 130,150,170....

Vorsicht: wir haben den programmierten Befehl LOAD nur im Zusammenhang mit dem Laden von Maschinensprachemodulen behandelt. Wenn Sie einen Ladevorgang in ein anderes BASIC Programm vornehmen möchten (verketten oder laden), gelten die obigen Regeln immer noch, müssen aber unter Umständen unterschiedlich benutzt werden.

Andere Verwechslungsmöglichkeiten beim SOV

Wir haben über die schlimmen Resultate gesprochen, die dann auftreten, wenn man ein Maschinenspracheprogramm in den Kassettenpuffer lädt (unter Benutzung des direkten Befehls), nachdem BASIC geladen wurde. Wir sollten nun gelernt haben, wie wir diesen Fehler vermeiden. Wie steht es mit Programmen, die in andere Bereiche gespeichert wurden, beispielsweise in den oberen Speicher oder direkt hinter das BASIC?

Angenommen, wir wollen ein Programm in den oberen Speicherbereich bringen, indem wir entweder den Zeiger für das Speicherende nach unten bewegen, um Platz zu schaffen, oder indem wir den freien RAM Bereich zwischen $C000 bis $CFFF beim Commodore 64 benutzen. Wir haben ebenfalls ein BASIC Programm zu laden. Wird der SOV verändert, wenn wir in der falschen Reihenfolge laden?

Die Antwort lautet ja, das Problem ist jedoch nicht so schwerwiegend. Wir stellen fest, daß, nachdem wir ein Programm durch einen direkten Befehl in den oberen Speicherbereich geladen haben, der SOV direkt darüber gesetzt wird. Damit steht er aber zu hoch, es gibt keinen Platz mehr für Variable. Was wir auch immer anstellen, wir erhalten die Fehlermeldung OUT OF MEMORY.

Offensichtlich können wir den SOV nicht in der oberen Stratosphäre belassen. Wir müssen zuerst den oberen Speicher und dann das BASIC Programm laden. Der zweite Ladevorgang wird den SOV Zeiger zurechtbiegen. Wenn Sie es so versuchen, werden Sie sehen, daß Sie den Zeiger für das obere Speicherende neu setzen müssen, indem Sie zwischen den beiden Ladevorgängen den Befehl NEW eingeben. Steht Ihnen andernfalls gar kein Speicherbereich mehr zur Verfügung, können Sie nicht einmal den nächsten LOAD Befehl ausführen.

Überblick: Zeiger setzen

Untersuchen Sie im Zweifelsfall die Zeiger, indem Sie diese mit dem Befehl .M anzeigen. Für den VIC/64/PLUS/4 lautet der Befehl .M 002B 003A, beim PET/CBM hieße er .M 0028 0037. Vergewissern Sie sich immer, daß der Zeiger für den Variablenanfang auf einen sinnvollen Wert gesetzt ist.

Wie immer können Sie einen falschen Speicherwert - diesmal einen falschen Zeiger - dadurch verändern, daß Sie den Cursor zurückbewegen, den zu verändernden Wert überschreiben und dann RETURN drücken.

Nach dem Ende von BASIC - Harmonie

Angenommen, wir setzen das Maschinenspracheprogramm an das Ende von BASIC - das sind die drei Nullen im Speicher - und bewegen den SOV nach oben, damit die Variablen dieses Programm nicht zerstören. Wird alles zufriedenstellend ablaufen?

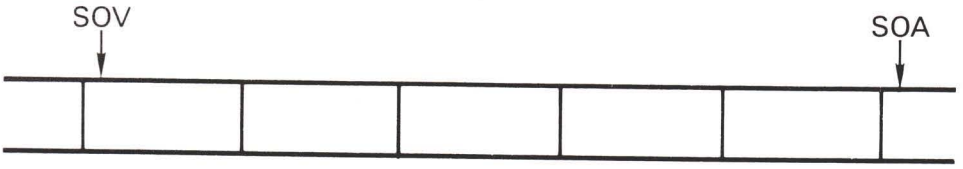
Das wird es in der Tat. Diesmal müssen wir unser BASIC Programm zuerst laden. Der SOV weist direkt hinter das BASIC. Dann können wir unser Maschinenspracheprogramm laden und der SOV verschiebt sich direkt dahinter. Der SOV weist exakt auf die richtige Stelle, wenn wir in der richtigen Reihenfolge laden, sonst werden die Variablen unser Maschinenspracheprogramm zerstören.

Sind unsere beiden Programme erst einmal vereinigt, werden sie gemeinsam (das BASIC und die Maschinensprache) mit dem Befehl SAVE als Kombination abgespeichert. Eine kurze Überlegung ergibt, daß der Speicherbereich vom BASIC Anfang ab bis genau vor den Variablenanfang alles enthält, was wir brauchen. Ein nachfolgender Ladevorgang bringt alles zurück und positioniert den SOV auf die richtige Stelle. Wir haben nun eine Programmeinheit - BASIC und Maschinensprache arbeiten zusammen, sie werden wie ein Programm geladen und gespeichert.

Bei dieser Anordnung verbleibt nur ein kleines Problem. Nachdem BASIC und Maschinenspracheprogramm miteinander verbunden sind, dürfen wir das BASIC Programm nicht ändern. Falls wir diesem Programm etwas hinzufügen oder wegnehmen, bewegt sich das Maschinenspracheprogramm nach oben oder unten. Die Verschiebung des Speichers überträgt sich auf den SOV. Das Programm wäre auf dem neuen Platz selbstverständlich nicht lauffähig und unsere SYS Befehle wären falsch.

BASIC Variable

Vier Arten von Einträgen können in die BASIC Variablentabelle vorgenommen werden. Unabhängig von der Art belegen alle Variablen sieben Bytes. Die ersten zwei Bytes bezeichnen den Namen und die übrigen fünf Bytes (nicht immer vollständig genutzt) enthalten den Wert oder die Definition. Der Variablentyp wird als Teil des Namen angezeigt: die oberen Bits von einem oder beiden Buchstaben des Namen bezeichnen einen speziellen Typ.



Jede Variable ist genau sieben Byte lang.
Variable erscheinen in der Reihenfolge ihrer Benutzung.

Abbildung 6.5

Wenn beispielsweise eine Fließkommavariable den Namen AB hat, würde dieser in den beiden Bytes als $41, $42 - dem ASCII Code für A und B - gespeichert. Genauso würde es aussehen, wenn wir die Variable ABACUS genannt hätten, da nur die ersten zwei Buchstaben des Namens benutzt werden. Wäre die Variable dagegen AB% genannt worden, um sie als ganzzahlige Variable zu deklarieren, würde der Name als $C1, $C2 gespeichert. Die ASCII Codes sind die gleichen, das obere Bit wurde nur darüber gesetzt. Um das Bild zu vervollständigen, eine Zeichenkettenvariable mit dem Namen AB$ würde mit $41, $C2 codiert - das obere Bit ist nur über dem zweiten Zeichen gesetzt.

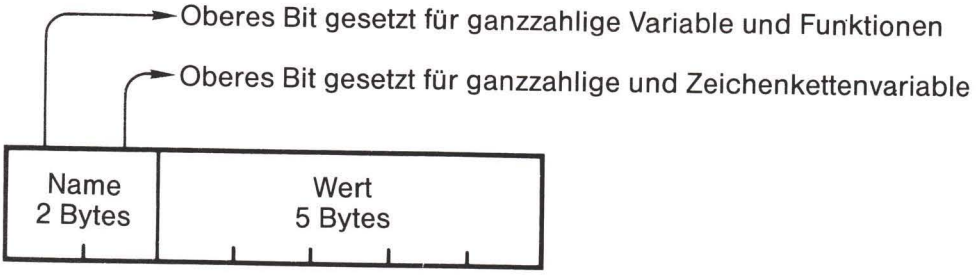


Abbildung 6.6

Es gibt eine vierte Art von Eintrag in die Variablentabelle. Dabei handelt es sich jedoch nicht um eine Variable, sondern um eine Funktionsdefinition. Wenn wir den Variablenbefehl DEF FNA (...) geben, erfolgt ein Eintrag in diese Tabelle. Dieser wird dadurch unterschieden, daß das obere Bit nur über dem ersten Zeichen gesetzt ist.

Zeichenkettenvariable benutzen nur drei der fünf zur Verfügung stehenden Bytes. Das erste Byte bezeichnet die Länge der Zeichenkette und die nächsten beiden Bytes geben die Adresse der Zeichenkette an. Diese Gruppe von drei Bytes wird als „Descriptor" bezeichnet, sie beschreiben die Eigenschaft der Zeichenkette.

Es gibt zwei Arten von numerischen Variablen: Fließkomma und ganzzahlige. Fließkommavariable nutzen alle fünf Bytes, ganzzahlige Variable benutzen nur die ersten zwei Bytes. Man kann den Wert einer Fließkommavariablen ermitteln, um damit zu arbeiten. Das ist jedoch nicht einfach. Eine Beschreibung, wie man dabei vorzugehen hat, befindet sich in Anhang F. Im Gegensatz dazu ist die Bestimmung des Wertes einer ganzzahligen Variablen und deren Benutzung einfach.

Wir wollen das in einem Beispiel versuchen. Tippen Sie NEW, gefolgt von A=5:B%=5. Damit erzeugt man zwei unterschiedliche Variable: A und B%. Gehen Sie jetzt zum Maschinensprachemonitor. Die Variablen sollten sich in der Nähe des BASIC Anfangs finden. Sie können ihre exakte Adresse auch durch Untersuchung des SOV Zeigers ermitteln ($2A/$2B beim PET/CBM oder $2D/$2E beim VIC, Commodore 64 oder PLUS/4). Beim Commodore 64 würden wir feststellen, daß die Variablen bei $0803 beginnen. Um beide anzuzeigen, tippen wir .M 0803 0810. Wir sehen die Fließkommavariable A:

```
41 00 83 20 00 00 00
```

Die ersten zwei Bytes stellen den Namen dar. 41 bedeutet in ASCII A und die Null besagt, daß kein zweiter Buchstabe vorhanden ist. Wo ist aber die 5? Eingebettet in 83 20 00 00 00. Es erfordert ein gehöriges Stück Arbeit, um die 5 für eine weitere Verarbeitung herauszufischen.

Hinter dieser Variablen sehen wir die ganzzahlige Variable B:

```
C2 80 00 05 00 00 00
```

Hex C2 ist in ASCII der Buchstabe B (42) mit dem oberen Bit gesetzt. 80 ist null mit dem oberen Bit gesetzt - es gibt wieder keinen zweiten Buchstaben. Der Wert selbst findet sich in den nächsten beiden Bytes und ist leicht zu lesen. Die letzten drei Bytes werden nicht benutzt.

Was läßt sich leichter mit Maschinensprache verbinden? Offensichtlich die ganzzahlige Variable. Sie eignet sich oft für die Programmarbeit zum Zählen von Zeichen, Setzen von Zeigern und ähnlichen Aufgaben.

Datenaustausch: BASIC und Maschinensprache

Um zwischen BASIC und Maschinensprache Daten hin und her zu schieben, gibt es mehrere Ansätze. Der einfachste besteht darin, mit POKE im BASIC den Wert in eine gegebene Stelle zu setzen. Die Maschinensprache lädt diese bei Bedarf. Für den umgekehrten Weg speichert die Maschinensprache den Wert und BASIC kann ihn mit PEEK übernehmen.

Eine andere Methode ist etwas komplizierter. BASIC Variable sind im Speicher abgelegt: warum kann ein Maschinenspracheprogramm nicht genau dorthin gehen, und ihren Wert ermitteln oder verändern? Die Idee hört sich verlockend an.

Bis jetzt wissen wir, wie wir in Maschinensprache nach einer speziellen BASIC Variable suchen müssen. Unter Angabe des Namens können wir die Adresse der ersten Variablen durch den SOV Zeiger erhalten und diese als indirekte Adresse speichern. Wenn wir die indirekte, indizierte Adressierung benutzen und das Y Register von 0 nach 1 fortschreiten lassen, können wir nachschauen, ob der Name übereinstimmt. Falls nicht, addieren wir zu der indirekten Adresse 7, um zur nächsten Variablen zu gelangen. Wenn diese übereinstimmt, ist unsere indirekte Adresse auf den Anfang dieser Variable gesetzt, wir können nun Y auf 2, 3, 4, 5 und 6 setzen und den gesamten Wert ermitteln. Handelt es sich bei der Variablen um eine ganzzahlige, brauchen wir lediglich die ersten zwei Bytes (Y=2 und 3) zu ermitteln. Befindet

sich die Variable nicht in der Variablentabelle, dann gelangen wir mit der indirekten Adresse auf den Arrayanfang und wissen damit, daß wir die Variable nicht gefunden haben.

Bei einer kleinen Zahl von Variablen gibt es einen schnelleren Weg. Variable werden in der Tabelle in der Reihenfolge, in der sie definiert wurden, abgelegt. Welche Variable auch immer als erste im BASIC Programm definiert wurde, sie wird am Anfang der Tabelle stehen. Wenn wir es so einrichten, daß unsere Variablen in einer bestimmten Reihenfolge definiert werden, können wir unsere Suche in Maschinensprache dergestalt vereinfachen, daß wir nach der ersten Variable, der zweiten Variable und so fort suchen, ohne die Namen beachten zu müssen.

Diese Vorgehensweise wollen wir einmal betrachten. Wenn wir die erste Variable benutzen wollen, ist das einzige, was wir brauchen, die Adresse der ersten Variable irgendwo auf der Seite 0, damit wir sie als indirekte Adresse benutzen können. Diese Adresse haben wir schon - es ist der SOV Zeiger, der Zeiger auf den Variablenanfang, der uns auf die erste Variable verweist. Durch entsprechende Erhöhung des Wertes von Y gelangen wir nach der ersten zur zweiten oder nach Bedarf zur dritten oder sechsunddreißigsten Variablen.

Vorhaben: Wir wollen das Maschinenspracheprogramm hinter das Ende von BASIC setzen. Das sieht in Abhängigkeit von der benutzten Maschine unterschiedlich aus. Das folgende Programm zeigt die korrekten Adressen für den Commodore 64. Die Werte für andere Maschinen kann man aus Anhang E entnehmen.

Zunächst schreiben wir unser BASIC Programm, um dessen Umfang zu bestimmen. Wir benötigen die Speicherstelle für das Ende von BASIC, um unser Maschinenspracheprogramm dort abzulegen. Dieses Programm soll die Maschinensprache veranlassen, nach einem Wert von V% zu fragen und diesen mit 10 zu multiplizieren. Denken Sie daran, NEW einzugeben. Wir schreiben das BASIC Programm folgendermaßen:

```
100 V%=0
110 FOR J=1 TO 5
120 INPUT "WERT";V%
130 SYS ++++
140 PRINT "MAL ZEHN =";V%
150 NEXT J
```

Wahrscheinlich benötigt unser BASIC Programm weniger als 127 Bytes. Wir können das später ermitteln, es scheint aber sicher, wenn wir den Beginn unseres Maschinenspracheprogramms bei ungefähr 2049 + 127 oder 2176 (hexadezimal 880) ansetzen. Auf dieser Grundlage können wir Zeile 130 zu SYS 2176 verändern. Versuchen Sie auf keinen Fall das Programm schon zu starten.

Wir können jetzt das BASIC Programm auf Band oder Disk speichern und das Maschinenspracheprogramm entwickeln. Das erlaubt uns, jeden der beiden Programmteile unabhängig voneinander zu verfeinern. Um es kurz zu machen und weil es sich bei unserem Beispiel um eine einfache Aufgabe handelt, wollen wir den Maschinencode direkt in den Speicher schreiben.

Schalten Sie auf den Maschinensprachemonitor um. Assemblieren Sie den folgenden Code:

```
.A 0880 LDY #$02
.A 0882 LDA ($2D),Y
.A 0884 STA $033C
.A 0887 STA $033E
.A 088A LDY #$03
.A 088C LDA ($2D),Y
.A 088E STA $033D
.A 0891 STA $033F
```

Wir haben jetzt zwei Bytes aus der ersten Variable V% ermittelt. Das obere Byte wurde sowohl nach $033C als auch nach $033E gespeichert. Warum, werden wir gleich sehen. Das untere Byte des Wertes wurde nach $033D und $033F gebracht.

Vorhaben für Enthusiasten: Durch effektivere Nutzung der Indizierung sollten Sie in der Lage sein, das obige Programm kompakter zu schreiben.

```
.A 0894 ASL $033D
.A 0897 ROL $033C
.A 089A ASL $033D
.A 089D ROL $033C
```

Wir haben den Inhalt von $033D/$033C mit zwei multipliziert und dann abermals mit zwei multipliziert. Damit enthalten diese Stellen den Originalwert mulitpliziert mit vier. Beachten Sie, daß wir das untere Byte mit ASL und das obere mit ROL bearbeitet haben. Wir sollten vielleicht auf Überlauf prüfen, wollen aber darauf vertrauen, daß die Zahlen im Bereich bleiben.

Da wir die Originalzahlen mal vier in $033D/$033C haben, können wir sie zu den Originalzahlen in $033F/$033E addieren, um die Originalzahl mal fünf zu erhalten:

```
.A 08A0 CLC
.A 08A1 LDA $033D
.A 08A4 ADC $033F
.A 08A7 STA $033D
.A 08AA LDA $033C
.A 08AD ADC $033E
.A 08B0 STA $033C
```

Jetzt enthalten die Stellen $033C/$033D die Originalzahl mal fünf. Wenn wir zuletzt die Zahl verdoppeln, erhalten wir den Wert mal zehn:

```
.A 08B3 ASL $033D
.A 08B6 ROL $033C
```

Wir haben die Zahl mit zehn multipliziert. Jetzt wollen wir sie in die Variable zurückgeben.

```
.A 08B9 LDY #$02
.A 08BB LDA $033C
.A 08BE STA ($2D),Y
```

```
.A 08C0 LDY #$03
.A 08C2 LDA $033D
.A 08C5 STA ($2D),Y
.A 08C7 RTS
```

Die Zahlen werden auf die gleiche Art, auf die wir sie geholt haben, zurückgespeichert. Wir müssen genau darauf achten, daß wir das obere und untere Byte richtig behandeln. Ganzzahlige Variable haben das hohe Byte zuerst, gefolgt von dem niedrigen Byte. Das ist genau umgekehrt, wie bei der Behandlung von 650x Adressen.

Wir müssen, bevor wir das Programm endgültig fertigstellen, folgendes machen. Wir müssen den Variablenanfangszeiger verändern, damit er auf eine Stelle oberhalb des Maschinenspracheprogramms weist. Das würde $08C8 sein. Wir zeigen den SOV mit `.M 002D 002E` an und verändern den Zeiger auf

```
.:002D C8 08 .. .. .. .. .. ..
```

Es folgt Überprüfen, Disassemblieren und dann zurück zu BASIC. Listen Sie, und Sie werden Ihr BASIC Programm wiederfinden, ohne irgendein Anzeichen von Maschinenspracheprogramm, aber mit SAVE wird alles zusammen gespeichert.

Starten Sie das BASIC Programm mit RUN, geben Sie Zahlen wie verlangt ein und vergewissern Sie sich, daß diese mit zehn multipliziert wurden.

Sie werden sich daran erinnern, daß unser Maschinenspracheprogramm nicht auf Überlauf prüft. Starten Sie das Programm abermals und versuchen Sie die höchste Zahl herauszufinden, die fehlerfrei mit zehn multipliziert werden kann. Was passiert beim Überlauf? Hätten Sie das erwartet?

Vorhaben für Enthusiasten: Können Sie dem obigen Programm einen Test auf Überlauf hinzufügen? Sie sollten entscheiden, was im Fall eines Überlaufs passieren soll: drucken Sie eine Warnung aus, setzen Sie den Wert zu Null oder was immer Sie wollen. Sie sollten jedoch das Programm nicht anhalten oder zum Monitor verzweigen, solche Dinge würden den Programmbenutzer verwirren. Ihr Programm wird länger werden. Vergessen Sie deshalb nicht, den SOV Zeiger bei $2D/$2E zu verändern, damit Ihr Programm vor den Variablen sicher ist.

Was wir gelernt haben

- Kurze Maschinenspracheprogramme können bequem in den Kassettenpuffer geschrieben und dort ausgetestet werden. Wir haben das während der Übungen gemacht. Dieser Bereich ist für lange Programme oder für Programme, die wir auf Band speichern wollen, nicht zufriedenstellend.
- Programme können in der Nähe des oberen BASIC Speichers eine halbpermanente Stelle finden. Der Zeiger für das Speicherende muß zu ihrem Schutz nach unten verschoben werden. Diese Programme müssen meist durch einen eigenen Ladevorgang dorthin gebracht werden.

- Programme können hinter das BASIC Ende gesetzt werden. Diese Stelle ist durch drei aufeinanderfolgende Null-Bytes im Speicher markiert. Der Zeiger für den Variablenanfang muß erhöht werden, damit die Variablen das Programm nicht überschreiben können. Man muß darauf achten, daß das BASIC Programm danach nicht mehr verändert werden darf.
- Beim VIC-20 ist der Anfang des BASIC häufig nach oben verschoben, um Platz für Bildschirminformationen im unteren Speicher zu schaffen. Durch eine weitere Verschiebung mit diesem Zeiger können wir Platz für Maschinenspracheprogramme gewinnen.
- Der Commodore 64 besitzt einen unbenutzten RAM Bereich zwischen Adresse $C000 und $CFFF. Prüfen Sie, ob keine anderen Programme diesen Bereich benutzen.
- Der Zeiger für den Variablenanfang ist mit den BASIC Befehlen SAVE und LOAD eng verknüpft. Es ist äußerst wichtig, daß man sich vergewissert, daß jeder Ladevorgang diesen Zeiger an einer sicheren Stelle beläßt, damit die Variablen das Programm nicht überschreiben und auf diese Weise zerstören können.
- Die Maschinensprache-Monitorbefehle .S (save) und .L (load) können dazu benutzt werden, Programme in verschiedenen Teilen des Speichers ablaufen zu lassen. Wiederum sollte man größte Sorgfalt darauf verwenden, daß die Zeiger nach Benutzung solcher Instruktionen sinnvolle Werte enthalten.
- Ein BASIC Programm kann einen LOAD Befehl enthalten, der etwa folgendes laden kann: ein anderes BASIC Programm, ein Maschinenspracheprogramm oder Daten. Wiederum ist eine vorsichtige Handhabung notwendig.
- Es gibt drei Hauptarten von BASIC Variablen: ganzzahlige, Fließkomma- und Zeichenkettenvariablen. Maschinenspracheprogramme sind in der Lage, jede dieser Variablen zu lesen und zu benutzen. Besonders eignen sich dazu die ganzzahligen Variablen.
- Wenn wir wollen, können wir die Suche nach BASIC Variablen dadurch erleichtern, daß wir sie in einer bestimmten Reihenfolge erzeugen.

Fragen und Vorhaben

Schreiben Sie ein einfaches BASIC- sowie Maschinenspracheprogramm. Über BASIC soll eine Zahl kleiner als 256 eingebbar sein. Bringen Sie diese mit POKE irgendwo in den Speicher. Rufen Sie Maschinensprache auf, um die Zahl durch 2 zu teilen. Holen Sie diese mit PEEK zurück und geben Sie sie aus.

Ein Programm, das andere Programme in den Speicher lädt, wird „bootstrap"-Programm genannt (Einfädelprogramm). Schreiben Sie ein einfaches BASIC „boot"-Programm, mit dem Sie ein früheres Übungsprogramm, das im Kassettenpuffer abläuft, laden können und rufen Sie es mit dem Befehl SYS auf (das Programm aus Kapitel 2, das HALLO ausdruckt, wäre dazu geeignet).

Einfädelprogramme sind besonders beim VIC-20, Commodore 64 und PLUS/4 beliebt, um große Datenmengen wie zum Beispiel Sprites, neue Zeichensätze oder die Information für ganze Bildschirmdarstellungen zu laden. Sie könnten Spaß daran finden, ein solches System zu erstellen.

Versuchen Sie es mit folgendem BASIC Programm

```
100 X=5
110 SYS ...
120 PRINT A
```

Schreiben Sie das Maschinenspracheprogramm, das von SYS aufgerufen wird, um den Namen der Variablen X in A umzuwandeln. Vorsicht: das mag spaßig sein, aber in richtigen Programmen ist es gefährlich, da Sie auf diese Weise zwei Variablen mit demselben Namen erzeugen könnten.

7. Stapel, USR, Unterbrechung und „Keil"

Dieses Kapitel behandelt:

- Der Stapel als temporärer Speicher
- USR: eine Alternative zu SYS
- Unterbrechungen: IRQ, NMI und BRK
- Die IA Bausteine: PIA und VIA
- BASIC infiltrieren: Der „Keil"

Eine kurze Bemerkung

Wenn Sie bis hierhin gefolgt sind und die verschiedenen Vorhaben ausgeführt haben, sollten Sie schon einiges von den Grundlagen der Maschinensprache verstehen. Sie sollten in der Lage sein, sich an eine Reihe kleiner Vorhaben heranzuwagen und Bereiche zu untersuchen, die von besonderem Interesse sein könnten.

Es ist an der Zeit, eine Pause einzulegen und eine Bestandsaufnahme durchzuführen. Die übrigen Kapitel stellen den Zuckerguß des Kuchens dar. Sie beschäftigen sich mit Details und besonderen Feinheiten der Maschinensprache. Falls Sie also irgendeine Unsicherheit bei den Dingen verspüren, die bisher behandelt worden sind, gehen Sie noch einmal zurück. Verankern Sie die Grundlagen zunächst ganz fest, bevor Sie fortfahren und sich in schwierigere Gefielde begeben.

Temporärer Speicher: der Stapel

Der Stapel (stack) ist ein für vorübergehende Informationen geeigneter Platz. Er funktioniert wie ein Aktenstapel: Sie können einen Gegenstand auf dem Stapel ablegen (push). Wenn Sie einen Gegenstand davon zurücknehmen (pull), nehmen Sie den, den Sie als letzten abgelegt haben. Formell bezeichnet man diesen Vorgang mit „zuletzt hinein, zuerst hinaus" (Last-In, First-Out = LIFO). Das ist natürlich und leicht zu verstehen.

Eine wichtige Regel für den Stapel sollte man sich immer merken: „Verlassen Sie ihn so sauber, wie Sie ihn vorgefunden haben". Mit anderen Worten, wenn Sie drei Dinge auf dem Stapel ablegen, vergewissern Sie sich, daß Sie diese drei Dinge auch wieder zurücknehmen. Verzweigen Sie niemals vorher und betrachten Sie den Stapel nicht als Abfallbehälter.

Der Stapel befindet sich im Speicher auf der Seite 1. Der Stapelzeiger (SP) wird zusammen mit den Registern angezeigt. Um die Informationen des Stapel zu betrachten, muß man den Wert $0100 zu diesem Wert addieren, um die nächst verfügbare Stapelposition zu erhalten. Ein Beispiel: wenn der SP einen Wert von $F8 zeigt, wird der nächste Gegenstand, der auf

dem Stapel abgelegt werden soll, in die Adresse $01F8 gelangen. In dem Moment, in dem wir einen Gegenstand auf dem Stapel ablegen, bewegt sich der Zeiger abwärts und nimmt den Wert $F7 an.

Beim Füllen des Stapels geht der Zeiger nach unten. Wenn die Gegenstände vom Stapel herunter genommen werden, geht der Zeiger nach oben. Ein niedriger Wert im Stapelzeiger zeigt einen vollen Stapel an: ein Wert unter $40 signalisiert Schwierigkeiten.

Der 650x Chip selbst läßt dem Stapel keine spezielle Behandlung zukommen. Wenn ein Maschinenspracheprogramm - möglicherweise auf Grund eines Programmierfehlers - eintausend Dinge auf dem Stapel ablegen möchte, ist das solange in Ordnung, wie der Mikroprozessor darin verwickelt ist. Der Stapel würde niemals die Seite 1 verlassen: wenn der Stapelzeiger den Wert null überschreitet, springt er um nach $FF und fährt fort. Sie würden natürlich niemals diese tausend unterschiedlichen Dinge zurückerhalten. Wenn auf ähnliche Weise ein Programm tausend Gegenstände vom Stapel herunternehmen möchte - ob sie nun vorher dort abgelegt wurden oder nicht - würde der Prozessor den Stapelzeiger munter Runde um Runde auf der Seite 1 herumbewegen und Bytes liefern. Natürlich würden nur 256 unterschiedliche Werte geliefert, der Prozessor kümmert sich nicht darum.

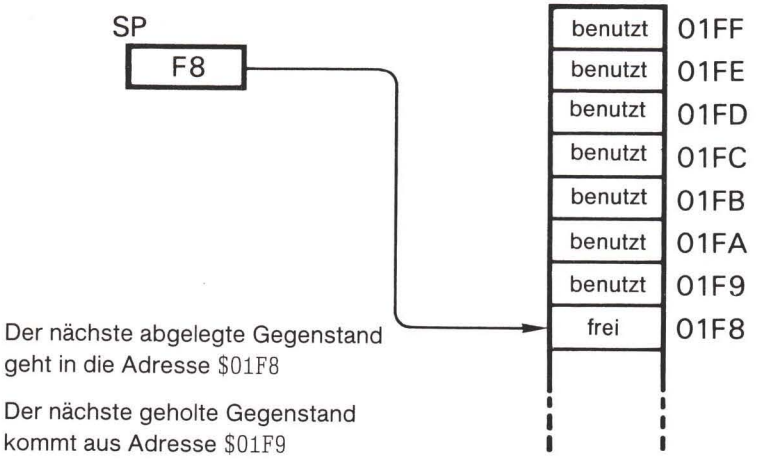


Abbildung 7.1

In einer BASIC Umgebung beginnt der Stapelzeiger bei $FA (der erste Gegenstand wird bei Adresse $01FA in den Stapel gelegt) und verläuft von hier an abwärts. Wenn der Stapelzeiger ungefähr den Wert $40 unterschreitet, signalisiert BASIC ein OUT OF MEMORY. Mehr als 160 Speicherstellen stehen im Stapel zur Verfügung, mehr als genug Platz für die meisten Anwendungen.

PHA (push A) und PLA (pull A)

Wie können wir den Stapelspeicher benutzen? Angenommen, wir haben einen Wert im Register A und wollen diesen einige Zeit später benutzen. Zuerst müssen wir etwas ausdrucken und das Zeichen, das wir ausdrucken wollen, muß in das A Register geladen werden. Wie können wir den Wert in A beiseite legen und später zurückbringen? Wir könnten ihn in ein anderes Register mit einem Transferbefehl (TAX oder TAY) schieben und später von dort zurückholen. Wir könnten ihn auch im Speicher ablegen und dann zurückladen. Alternativ könnten wir das A Register auf dem Stapel ablegen (PHA) und den Wert später davon zurückholen (PLA).

Wir wollen das wieder mit einem Beispiel erläutern. Angenommen, das A Register enthält den Wert 5 und der Stapelzeiger steht auf $F3. Wenn das Programm PHA befiehlt, wird der Wert 5 in die Adresse $01F3 gespeichert und der Stapelzeiger verändert sich auf $F2. Später treffen wir im Programm auf den Befehl PLA: der Stapelzeiger bewegt sich auf $F3 zurück und der Wert 5 wird aus der Adresse $01F3 gelesen und in das Register A gesetzt.

Eine einfache Art, um einen Wert aus dem A Register für eine kurze Zeit beiseite zu legen.

PHP (push processor status) und PLP

Wenn wir ein Programm schreiben, möchten wir manchmal auf einen bestimmten Zustand sofort prüfen, auf das Ergebnis dieses Tests aber später reagieren. Wir können das dadurch erreichen, daß wir die augenblicklichen Flag Werte beiseite legen und sie erst dann zurückholen, wenn wir sie überprüfen wollen. Um alle Flags auf dem Stapel abzulegen, benutzen wir den Befehl PHP (push the processor status word = lege die Prozessorzustandsinformation auf dem Stapel ab), um die Flags in das Statusregister (SR) zurückzuladen, benutzen wir PLP (pull the processor status word = hole die Prozessorzustandsinformation vom Stapel zurück).

Wozu das nützlich sein kann, soll wieder ein Beispiel illustrieren. Angenommen, wir lesen eine Datei von Kundeneinkäufen, und während wir einen Wert eingeben, stellen wir fest, daß es sich dabei um den letzten handelt - das Ende der Datei ist erreicht. Es bedeutet, daß wir die Datei abschließen und die Aktivitäten des Kunden zusammenfassen wollen. Das wollen wir nicht sofort tun. Zuerst müssen wir das, was wir eingegeben haben, bearbeiten. Wir können nun die Information, daß es sich um das Ende der Datei handelt, „stapeln", um den letzten Eintrag auf gleiche Weise zu bearbeiten wie frühere Einträge. Anschließend holen wir uns die Statusinformation zurück, um zu entscheiden, ob wir die Datei abschließen und die Ergebnisse ausdrucken. Wir wollen im nächsten Kapitel PHP und PLP für genau diese Art von Aufgabe verwenden.

PHA und PHP legen beide genau einen einzigen Gegenstand auf dem Stapel ab, PLA und PLP nehmen einen einzigen. Es gibt andere Befehle, die mehr als eine Stapelstelle bearbeiten.

JSR und RTS

Wir kennen diese Befehle schon. Was haben sie hier zu suchen?
Wenn der Befehl JSR ausgeführt wird, wird die Rückkehradresse auf dem Stapel abgelegt.

Wenn der Befehl RTS ausgeführt wird, wird die Rückkehradresse vom Stapel geholt und das Programm kehrt dorthin zurück.

Etwas genauer ausgedrückt, wenn ein JSR vorliegt, legt der Prozessor die Rückkehradresse minus eins als zwei Byte auf dem Stapel ab. Der höherwertige Teil der Adresse geht zuerst auf den Stapel. Wird der Befehl RTS erreicht, nimmt der Prozessor die beiden Bytes vom Stapel, addiert eine Eins und fährt mit dem Programm an der so ermittelten Adresse fort.

Beispiel: Wenn Adresse $0352 den Befehl JSR $033C enthält, laufen folgende Ereignisse ab. Die Rückkehradresse würde $0355 sein, der Befehl, der direkt hinter JSR steht. Es wird jedoch eine Adresse $0354 berechnet - die 03 geht zuerst auf den Stapel und die 54 darunter. Die Unterroutine bei $033C wird nun abgearbeitet. Möglicherweise trifft sie auf ein RTS. Die Werte 54 und 03 werden vom Stapel genommen, in die Adresse $0354 umgewandelt, Eins dazu addiert und der Prozessor nimmt seine Arbeit bei Adresse $0355 wieder auf.

Sie brauchen das überhaupt nicht zu wissen. Wir haben Unterroutinen ohne Kenntnis dieser Details schon einige Male benutzt. Manchmal ist es jedoch nützlich, den Stapel mit der Frage untersuchen zu können „Wer hat diese Unterroutine aufgerufen?" Hier findet sich die Antwort.

Unterbrechungen und RTI

Es gibt drei Arten von Unterbrechungen (interrupt): IRQ, NMI und den Befehl BRK. Bei IRQ (interrupt request = Unterbrechungsanforderung) und NMI (non-mascable interrupt = nicht maskierbare Unterbrechung) handelt es sich um Anschlüsse des 650x. Wenn ein geeignetes Signal an den entsprechenden Anschluß gelegt wird, wird der Prozessor zur Unterbrechung und dazu veranlaßt, eine Unterbrechungsroutine (interrupt routine) zu durchlaufen. Den Befehl BRK kann man sich als eine falsche Unterbrechung vorstellen - er verhält sich auf ähnliche Weise wie IRQ.

Taucht ein Unterbrechungssignal auf, vollendet der Prozessor den Befehl, den er gerade bearbeitet. Dann nimmt er den Inhalt des Registers PC (des Befehlszählers, der die Adresse des nächsten Befehls enthält) und legt ihn auf dem Stapel ab, das obere Byte zuerst. Zum Schluß legt er das Statusregister auf den Stapel. Dabei handelt es sich um insgesamt drei Bytes, die auf den Stapel wandern.

Der Prozessor übernimmt dann die Ausführungsadresse aus einer der folgenden Speicherstellen:

IRQ oder BRK - von $FFFE und $FFFF
NMI - von $FFFA und $FFFB

Welcher Wert auch immer in diesen Zeigern gefunden wird, er wird zur neuen Unterbrechungsausführungsadresse: der Prozessor nimmt seine Arbeit bei dieser Adresse auf. Möglicherweise trifft der Prozessor auf den Befehl RTI. Das Statusregister und die PC Adresse werden dann vom Stapel geholt und das Programm wird an der Stelle wieder aufgenommen, an der es verlassen wurde.

Man beachte, daß es sich bei der Adresse im Stapel hier um die wahre Rückkehradresse handelt, im Unterschied zu JSR/RTS, bei denen die Rückkehradressen minus Eins gespeichert wurde.

Bei allen Commodore Maschinen wird IRQ ungefähr sechzigmal pro Sekunde aufgerufen. Der Befehl NMI wird bei PET/CBM nicht benutzt (er ist jedoch verfügbar). Er steht in der 264 Serie nicht zur Verfügung. Beim VIC-20 und Commodore 64 wird er für die RESTORE Taste und für die RS-232 Kommunikation verwandt.

Der Befehl BRK kann vom IRQ Signal durch ein Bit im Statusregister unterschieden werden. Bit 4 ist die B oder „break" Flag. Wenn dieses Bit gesetzt ist, wurde die letzte Unterbrechung durch ein BRK und nicht durch ein IRQ verursacht.

Wir wollen später die Unterbrechungsroutinen behandeln und in unseren eigenen Programmen anwenden. In der Zwischenzeit können wir die Unterbrechung anders verwenden: mehrere zusätzliche Dinge werden im Stapel abgelegt: das A, X und Y Register. Das wird durch ein Programm im ROM, nicht durch den Prozessor ausgeführt. Wir können diese Register benutzen, da wir wissen, daß sie am Ende der Unterbrechung sicher zurückgespeichert werden, was sich als sehr praktisch erweist.

Vermischen und Vergleichen

Der Prozessor benutzt den Stapel mechanisch. Wenn wir den Stapel zu manipulieren verstehen, können wir ihn für überraschende Dinge einsetzen. Beispielsweise können wir den Befehl RTS geben, obwohl keine Unterroutine aufgerufen wurde. Das einzige, was wir tun müssen, ist, den Stapel mit der erwünschten Adresse vorzubereiten. Überlegen Sie einmal, was der folgende Code ausführt:

```
LDA #$24
PHA
LDA #$68
PHA
RTS
```

Diese Codierung entspricht einem JMP $2469. Wir haben eine „falsche Rückkehradresse" auf den Stapel gelegt und RTS hat sie aufgenommen und benutzt. Das scheint ziemlich unnütz zu sein, wir hätten auch direkt JMP $2469 schreiben können. Schauen Sie sich jedoch den folgenden Code an:

```
LDA TABLE1, X
PHA
LDA TABLE2, X
PHA
RTS
```

Das benutzte Prinzip ist dasselbe, aber jetzt können wir in Abhängigkeit von dem Wert X zu jeder der verschiedenen Adressen ausbrechen.

USR: ein Bruder von SYS

Wir haben den Befehl SYS schon einige Male benutzt. Er bedeutet „gehe zur angegebenen Adresse und führe den dort befindlichen Maschinencode als Unterroutine aus". USR ähnelt dem in vielerlei Hinsicht: es bedeutet „gehe zu einer festen Adresse und führe den dort befindlichen Maschinencode als Unterroutine aus". Die feste Adresse kann man durch POKE in den USR Zeiger schreiben. Bei den meisten Commodore Maschinen befindet er sich bei Adresse 1 und 2. Beim Commodore 64 liegt er bei Adresse 785 und 786 (hex 0311 und 0312).

Es gibt einen Unterschied, der zunächst wichtig zu sein scheint. Bei SYS handelt es sich um einen Befehl, bei USR um eine Funktion. Sie können nicht den Befehl USR (0) eingeben - Sie würden lediglich ein SYNTAX ERROR erhalten. Vielmehr müssen Sie etwa PRINT USR (0) oder X = USR (0) eingeben, wobei USR als Funktion benutzt wird. Es scheint, als ob SYS mit Aktionsprogrammen und USR mit Berechnungsprogrammen in Zusammenhang zu bringen seien. In Wirklichkeit ist der Unterschied bei deren Benutzung nicht so groß.

Welcher Wert auch in Klammern stehen mag (das Argument der USR Funktion), er wird bestimmt und in den verschiebbaren Akkumulator geladen, bevor die USR Funktion aufgerufen wird. Der verschiebbare Akkumulator befindet sich bei den meisten PET/CBM Computern bei $5E bis $63, bei $61 bis $66 beim VIC-20, Commodore 64 und PLUS/4. Wie wir im Kapitel 6 angedeutet haben, ist die Fließkommadarstellung sehr komplex. Die meisten Anfänger unter den Programmierern geben sich lieber nicht damit ab, sondern übergeben die Werte durch POKE Befehle oder ganzzahlige Variable in den Speicher.

Wenn die USR Funktion die Kontrolle an BASIC zurückgibt, findet man ihren Funktionswert im verschiebbaren Akkumulator wieder. Wenn wir ihn nicht verändert haben, ist er der gleiche Wert wie das Argument. In den meisten Fällen würde der Befehl PRINT USR (5) deshalb den Wert 5 ausdrucken.

Unterbrechungen: NMI, IRQ und BRK

Wir haben den mechanischen Aspekt der Unterbrechung bereits erwähnt. Jetzt wollen wir uns den Gebrauch der Unterbrechung bei einfachen Aufgaben einmal anschauen.

Der Befehl IRQ ist an einen Zeiger im RAM gebunden. Wenn wir die Adresse in diesem Zeiger verändern, verändern wir die Adresse, an die die Unterbrechung springt. Der Unterbrechungszeiger befindet sich an folgenden Stellen:

Die meisten PET/CBM:	0090 - 0091 (dezimal 144 - 145)
VIC/Commodore 64:	0314 - 0315 (dezimal 788 - 789)

Bevor wir diesen Zeiger verändern, sollten wir uns über etwas Wichtiges im Klaren werden: die Unterbrechung führt sechzigmal in der Sekunde viele Arbeiten durch. Sie bringt die Uhr auf den neuesten Stand, prüft die RUN/STOP Taste, bedient den Kassettenmotor, bringt den Cursor zum Blinken und bearbeitet Tastatureingaben. Wenn Sie den IRQ Zeiger gedankenlos verändern, wird die Bearbeitung all dieser Dinge beendet. Es ist schwierig, einen Computer

mit einer toten Tastatur zu handhaben. Sie könnten zwar alle diese Funktionen selbst programmieren, es gibt jedoch einen einfacheren Weg.

Angenommen, wir wollen den Zeiger verbiegen, um unsere eigenen Programme abzuarbeiten und am Ende unserer Programme erlaubten wir der Unterbrechung, mit dem fortzufahren, mit dem sie sich normalerweise beschäftigt. Auf diese Weise würde unser Programm sechzigmal in der Sekunde bearbeitet und die üblichen Unterbrechungsroutinen würden immer noch ausgeführt.

Das ist nicht schwierig, und wir können dadurch viele interessante Effekte erreichen. Denken Sie aber daran, daß die Unterbrechung jederzeit abläuft, sogar dann, wenn kein BASIC Programm ausgeführt wird. Durch Spielen mit der Unterbrechung können wir ein permanentes Computersystem verändern, das sogar dann abläuft, wenn kein Programm geladen ist.

Wir müssen bei der Veränderung der Unterbrechungszeiger sehr vorsichtig vorgehen. Angenommen, wir sind gerade dabei, eine zwei-Byte Adresse zu verändern. Wir haben das erste Byte verändert und plötzlich geschieht eine Unterbrechung. Diese benutzt eine Adresse, die weder Fisch noch Fleisch ist: die eine Hälfte besteht aus der alten, die andere aus der neuen Adresse. In solch einem Fall wird die Unterbrechungsroutine wahrscheinlich durcheinander gebracht und, wenn die Unterbrechung durcheinander ist, gerät der ganze Computer in Schwierigkeiten. Wir müssen also einen Weg finden, mit dem wir die Unterbrechung während der Veränderung des Zeigers verhindern.

Das können wir in Maschinensprache erreichen: bevor eine Routine den IRQ Zeiger verändert, können wir den Befehl SEI (set interrupt disable = setze Unterbrechungsperre) geben. Nach diesem Befehl kann IRQ uns nicht unterbrechen. Wir können nun den Zeiger setzen und dann die Unterbrechung mit dem Befehl CLI (clear interrupt disable = lösche Unterbrechungsperre) wieder zulassen. Vergewissern Sie sich, daß Sie den letzten Befehl geben, da die Unterbrechungsroutine viele für den Computer wichtige Funktionen ausführt. Wir können auch sagen, wir haben die Unterbrechung während des Zeitraums zwischen der Ausführung von SEI und CLI maskiert. Die Unterbrechung NMI (=nicht maskierbare Unterbrechung) ist nicht maskierbar. Das bedeutet, daß SEI auf diese keinen Einfluß hat.

Zum Sperren der Unterbrechung gibt es eine zweite Möglichkeit - wir können die Unterbrechungsquelle selbst ausschalten. Irgendetwas verursacht eine Unterbrechung, es könnte sich dabei um einen Zeitgeber, um ein äußeres Signal, oder sogar um ein Bildschirmereignis handeln. Was es auch sei, wir können die Quelle der Unterbrechung angehen und diese abschalten.

Letztendlich werden alle Unterbrechungssignale von einem IA (interface adaptor) Chip geliefert. Diese Schaltkreise erlauben ohne Unterschied eine zeitweise Blockierung der Unterbrechungssignale. Wir wollen die IA Chips später behandeln. Nur soviel, die normalen Unterbrechungssignale können durch folgende Vorgehensweisen blockiert werden:

Commodore 64: speichern Sie $7F in Adresse $DC0D (POKE 56333,127) zum Abschalten; speichern Sie $81 in die gleiche Adresse (POKE 56333,129) zum Anschalten der Unterbrechung.

VIC-20: speichern Sie $7F in Adresse $912E (POKE 37166,127) zum Abschalten; speichern Sie $C0 in die gleiche Adresse (POKE 37166,192) zum Anschalten der Unterbrechung.

PET/CBM: speichern Sie $3C in Adresse $E813 (POKE 59411,60) zum Abschalten; speichern Sie $3D in die gleiche Adresse (POKE 59411,61) zum Anschalten der Unterbrechung.

Es funktioniert, ohne daß man besonders betonen muß, daß die obigen POKE Befehle normalerweise nicht als direkte Befehle gegeben werden dürfen. Das erste POKE legt in jedem Fall die Tastatur lahm (neben anderen Dingen) und Sie wären nicht in der Lage, das rückspeichernde zweite POKE einzugeben.

Eine Warnung zu Unterbrechungsprogrammen: eine Veränderung des IRQ Zeigers führt zu Schwierigkeiten beim Laden und Speichern von Programmen. Bevor Sie eine dieser Aktivitäten durchführen, müssen Sie den Zeiger in seinen ursprünglichen Zustand zurückversetzen.

Ein Interrupt Vorhaben

Die folgenden Programme sind nur für den Commodore 64 geschrieben worden. Die entsprechenden Codes für den PET/CBM findet man im Anhang E.

Wir werden nun das Programm für die Unterbrechung selbst schreiben. Wir wollen den Inhalt der Adresse $91 sechzigmal in der Sekunde auf den oberen Teil des Bildschirms schreiben. Auf geht's:

```
.A 033C LDA $91
.A 033E STA $0400
.A 0341 JMP ($03A0)
```

Wozu der indirekte Sprung? Wir wollen die reguläre Unterbrechungsroutine abfangen, wissen aber noch nicht, wo sie sich befindet. Wenn wir die Adresse gefunden haben, werden wir sie in den Speicherplatz $03A0/$03A1 einfügen, damit der indirekte Sprung die entsprechende Verbindung herstellen kann.

Wir wollen nun die Routine für den obigen Unterbrechungscode aufschreiben. Dazu kopieren wir zuerst die Unterbrechungsadresse von $0314 in die indirekte Adresse bei $03A0:

```
.A 0344 LDA $0314
.A 0347 STA $03A0
.A 034A LDA $0315
.A 034D STA $03A1
```

Wir sind jetzt bereit, die Adresse für unsere eigene Unterbrechungsroutine (bei $033C) in den IRQ Zeiger zu setzen.

```
.A 0350 SEI
.A 0351 LDA #$3C
.A 0353 STA $0314
.A 0356 LDA #$03
.A 0358 STA $0315
.A 035B CLI
.A 035C RTS
```

Die neue Unterbrechungsprozedur wollen wir durch ein SYS auf $0344 einschalten, oben durch SYS 836. Bevor wir diesen Befehl geben, müssen wir noch einige Zeilen schreiben, um alles zurückzusetzen:

```
.A 035D SEI
.A 035E LDA $03A0
.A 0361 STA $0314
.A 0364 LDA $03A1
.A 0367 STA $0315
.A 036A CLI
.A 036B RTS
```

Wir setzen die Originaladresse, wie Sie sehen können, zurück, indem wir sie aus dem indirekten Adreßbereich, in dem sie gespeichert wurde, kopieren.

Nachdem das Programm eingegeben, disassembliert und überprüft wurde, kehren wir ins BASIC zurück. SYS 836 ruft den neuen Unterbrechungscode auf. SYS 861 schaltet ihn wieder ab. Beachten Sie, daß das Zeichen (eine Kopie des Inhalts von Adresse $91) in der linken oberen Ecke des Bildschirms erscheint. Man kann das Zeichen durch Tippen einiger Tasten beeinflussen. Können Sie feststellen, wieviele Tasten betroffen sind?

Bei einigen Modellen des Commodore 64 wird blau auf blau ausgedruckt, wenn man den Bildschirmspeicher durch POKE verändert, wie wir es hier tun. Sollte das der Fall sein, erscheint nicht immer ein Zeichen in der oberen linken Ecke. Vorhaben für Enthusiasten: Beseitigen Sie dieses Problem, indem Sie in die Farbnybble-Tabelle bei Adresse $D800 einen Wert speichern.

Die IA Chips: PIA, VIA und CIA

Die Interfaceadapter (IA) Bausteine sind sehr detailreich. Sie werden die Spezifikationen im einzelnen lesen müssen, um sie vollständig zu verstehen. Hier wollen wir uns mit ihren Hauptfunktionen beschäftigen.

PIA steht für peripheral interface adaptor, VIA für versatile interface adaptor (vielseitiger Interfaceadapter) und CIA für complex interface adaptor. Unter Commodore Besitzern kursieren Spekulationen darüber, ob der nächste Chip „FBI" genannt werden wird.

Durch den Interfaceadapter werden folgende Funktionen ausgeführt:

1. Ereignisspeicherung und Unterbrechungskontrolle. Wir haben festgestellt, daß diese Chips zur Blockierung des Unterbrechungssignals manipuliert werden können. In Wirklichkeit stellen sie mehr als eine „Tür" für ein Signal dar, dadurch, daß sie entweder den Durchgang zum Auslöser des Prozessor IRQ freihalten oder aber blockieren. Sie speichern oft auch ein Signal in einer Ereignisflag. Gelegentlich wird diese Unterbrechungsflag genannt.

   Diese Zwischenspeicherung ist wichtig. Ein auslösendes Ereignis könnte ja so kurz sein, daß das ursprüngliche Signal, das die Unterbrechung auslöste, bereits vorüber ist, bevor

der Prozessor Gelegenheit fand, es aufzunehmen. Eine IA Ereignisflag hält das Signal solange fest, bis ein Programm sie abschaltet.

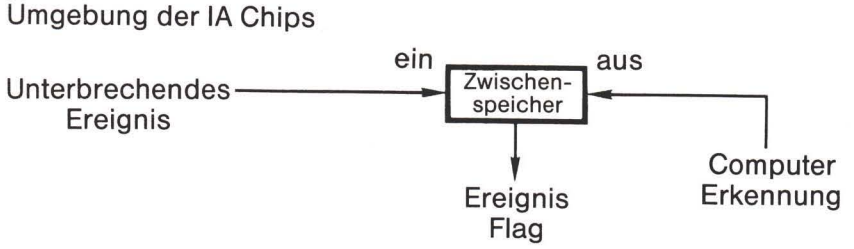


Abbildung 7.2

Wenn ein Ereignis zeitkritisch ist - d.h., wenn der zeitliche Ablauf eines Ereignisses genau gemessen werden muß oder wenn die Ereignisflag zur Feststellung eines neuen Ereignisses schnell gelöscht werden muß - können wir die Ereignisflag mit der Unterbrechungsleitung verbinden. In diesem Fall führt das Auftreten eines Ereignisses zu einer Unterbrechung des Prozessors. Wir müssen ein Programm schreiben, das an die Unterbrechungsroutinen angebunden ist, um dieses Ereignis festzustellen. Wir müssen dann die Flag löschen und den notwendigen Prozeßablauf durchführen. Die Verbindung zur Unterbrechungsleitung stellen wir mit Hilfe eines Registers her, welches meist als interrupt enable register (Unterbrechung möglich) bezeichnet wird.

Andererseits gibt es Ereignisse, die nicht besonders zeitkritisch sind. In diesem Fall genügt es, die entsprechende Ereignisflag von Zeit zu Zeit zu überprüfen. Tritt ein Ereignis ein, können Sie die Flag löschen und auf dieses Ereignis reagieren. Eine Unterbrechung ist nicht nötig. Selbst wenn eine Ereignisflag nicht mit einer Unterbrechung im Zusammenhang steht, kann sie als Unterbrechungsflag bezeichnet werden. Lassen Sie sich nicht durch die Terminologie durcheinander bringen.

Ob Sie diese Ereignisse nun durch Unterbrechungssequenzen behandeln oder nicht, wichtig ist, daß Sie dafür verantwortlich sind, die Ereignisflag zu löschen. Die Flag speichert das Signal solange, bis es ausgeschaltet wird, und normalerweise wird es nur durch ein Programm ausgeschaltet.

Die verschiedenen Flags werden durch Zeitgeber oder äußere Signale gesetzt. Sie können den Zustand einer Flag durch Überprüfung des interrupt flag register lesen. Verschiedene Flags werden in diesem Register zusammengepackt. Sie können, wie immer, logische Operatoren - AND, ORA oder EOR - dazu benutzen, um die jeweiligen, Sie interessierenden Flags herauszuschälen oder zu modifizieren. Sie können auch das IFR (interrupt flag register) zum Löschen der Flags benutzen.

2. Zeitabläufe. Bestimmte Adressen innerhalb des IA Chip sind häufig mit Zeitgeberfunktionen verknüpft. Diese Zeitgeber zählen rückwärts. Das bedeutet, wenn wir einen Wert von $97 in einen Zeitgeber setzen und sofort danach den Wert untersuchen, können wir zum Beispiel einen Wert von $93 vorfinden. Zeitgeber kommen in den unterschiedlichsten For-

men und Größen vor (zu Einzelheiten befragen Sie das Datenblatt des Bausteins). Die meisten von ihnen schalten eine Unterbrechungsflag um, wenn sie auf Null heruntergezählt haben. Wie wir schon sagten, können Sie darüber entscheiden, ob diese Flag wirklich ein Unterbrechungssignal auslösen soll oder nicht.

3. Eingabe/Ausgabe. Bestimmte Adressen innerhalb des IA Chip sind mit Ausgabekanälen (ports) verbunden, die den Computer mit der Außenwelt verbinden. Auf diese Weise kann der Computer äußere Ereignisse feststellen oder außenliegende Einheiten kontrollieren. Ausgabesignale sind von ihrer Natur her speichernde Signale, was bedeutet, daß ein Befehl „Speichere" folgendermaßen aufgefaßt werden kann: „Schalte Ausgabekanal 5 ein und laß ihn eingeschaltet".

Hinweise zu IA Chips

Viele Adressen innerhalb eines IA Chip haben in Abhängigkeit davon, ob in sie hinein geschrieben (gespeichert) oder ob aus ihnen heraus gelesen (geladen) wird, unterschiedliche Bedeutung. Achten Sie darauf besonders, wenn Sie die Spezifikationen der Bausteine durchlesen.

Der Vorgang, der eine Unterbrechungsflag ausschaltet, ist oft ungewohnt. Er gleicht seltsamerweise einem Einschaltmodus für eine Flag. Sie müssen daran denken, daß eine Flag ausschließlich durch die äußere Aktivität, an die sie gebunden ist, eingeschaltet werden kann. Auch wenn es ungewöhnlich aussieht, daß Sie durch Speicherung eines Wertes 1 in Bit 0 die Flag ausschalten (als ob Sie Bit 0 anschalten wollten), kümmern Sie sich nicht darum. Sie werden sich daran gewöhnen.

Das IER (interrupt enable register) bereitet oft Schwierigkeiten. In vielen Fällen hat das obere Bit eines Wertes, den wir speichern, eine besondere Bedeutung. Wenn dieses gesetzt ist, schalten die anderen Bits die zugehörige Unterbrechung an. Wenn dieses gelöscht ist, sorgen die übrigen Bits dafür, daß die entsprechenden Unterbrechungsverbindungen abgeschaltet sind. Sie werden sich vielleicht daran erinnern, daß wir beim Commodore 64 die Unterbrechung dadurch abgeschaltet haben, daß wir $7F in Adresse $DC0D speicherten. Das sieht ungewöhnlich aus: wir speichern einen Binärwert von $01111111. Damit scheinen wir Bits anzuschalten. In Wirklichkeit weist der Wert 0 im oberen Bit darauf hin, daß alle übrigen Bits Ausschaltsignale darstellen. Auf diese Weise bewirkt der Wert eine Blockierung aller Unterbrechungen.

BASIC Infiltrieren: der „Keil"

Auf der Seite 0 gibt es eine Unterroutine, die der BASIC Interpreter häufig dazu benutzt, Informationen von Ihren BASIC Programmen zu übernehmen. Diese Unterroutine wird dazu benutzt, ein Zeichen aus Ihrem BASIC Programm anzunehmen und auf dessen Typ zu überprüfen (numerisch, Befehlsende oder andere).

Diese Routine wird normalerweise von zwei Stellen aus durchlaufen: CHRGET, um das nächste Zeichen aus Ihrem BASIC Programm zu übernehmen und CHRGOT, um das letzte Zeichen

zu überprüfen. Die Unterroutine befindet sich bei $0070 bis $0087 bei den meisten PET/CBM Computern und bei $0073 bis $008A beim VIC-20 oder Commodore 64. Auf Wunsch können Sie sie dort disassemblieren. Der Programmteil wird unten beschrieben.

Da sich CHRGET je nach Maschine an unterschiedlichen Stellen befindet, wird der folgende Programmteil mit symbolischen Adressen dargestellt. Das bedeutet, statt die hex Adreßwerte zu zeigen, wird die Adresse mit einem Namen oder Symbol bezeichnet. Demnach steht CHRGOT für die Adresse $0079 und CHRGOT+1 bezeichnet die Adresse $007A und so weiter.

```
CHRGET INC CHRGOT+1
       BNE CHRGOT
       INC CHRGOT+2
CHRGOT LDA xxxx
```

Diese Unterroutine modifiziert sich selbst, womit gemeint ist, daß sie Teile von sich selbst während des Programmlaufs verändert. Hierbei handelt es sich nicht immer um eine gute Programmiertechnik, funktioniert in diesem Fall jedoch ausgezeichnet.

Der erste Teil der Unterroutine addiert eine Eins zu der Adresse, die von der Instruktion CHRGOT benutzt wird. Es handelt sich dabei um eine übliche Art, eine Adresseninkrementierung zu codieren: addiere Eins zum niederwertigen Byte der Adresse. Wenn wir den Wert Null dadurch erreichen, muß das niederwertige Byte von $FF nach $00 umgesprungen sein. In diesem Fall addiere Eins zu dem höherwertigen Byte.

Die Adresse, die durch CHRGOT geladen wurde, liegt innerhalb Ihres BASIC Programms oder im Bereich des Eingabepuffer, wenn Sie gerade einen direkten Befehl eingegeben haben. Bevor wir die folgenden Programmteile näher betrachten, wollen wir uns unser Vorhaben genauer anschauen:

1. Wenn wir ein Leerzeichen finden, kehren wir zurück und holen das nächste Zeichen.
2. Wenn wir eine Null finden (in BASIC ein Zeilenende) oder einen Doppelpunkt (hex $3A, in BASIC ein Befehlsende), möchten wir die Z Flag setzen und aus der Unterroutine herausgehen.
3. Wenn wir einen numerischen Wert finden, wollen wir die Z Flag löschen. Finden wir keinen numerischen Wert, wollen wir sie setzen.

```
CHRGOT LDA xxxx
       CMP #$3A
       BCS EXIT
```

Handelt es sich bei dem Zeichen um einen Doppelpunkt ($3A), verlassen wir die Unterroutine mit gesetzter Z Flag. Das ist eines unserer Vorhaben. Nun der Teil eines weiteren: ist das Zeichen $3A oder größer, kann es sich nicht um einen ASCII Zahlenwert handeln - Zahlenwerte liegen im Bereich zwischen $30 und $39.

```
CMP #$20
BEQ CHRGET
```

Handelt es sich bei dem Zeichen um ein Leerzeichen, kehren wir zurück und holen ein weiteres Zeichen.

Der folgende Code sieht ein wenig fremd aus, ist aber korrekt. Nach den beiden Subtraktionen enthält das A Register den gleichen Wert wie zu Beginn:

```
SEC
SBC #$30
SEC
SBC #$D0
```

Hiernach ist das A Register unverändert, dagegen ist die C Flag dann gesetzt, wenn die Zahl kleiner als $30 ist. Das bedeutet, es handelt sich nicht um einen ASCII Zahlenwert. Zusätzlich ist die Z Flag dann gesetzt, wenn A den binären Wert 0 enthält. Wir haben alle unsere Vorhaben erfüllt und können nun zurückkehren:

```
EXIT RTS
```

Einbruch in BASIC

Da BASIC häufig auf diese Unterroutine zugreift, können wir durch eine Veränderung dieser Unterroutine BASIC unterwandern. Eine zusätzliche Programmierung in diesem Bereich wird häufig ein „Keilprogramm" genannt (wedge program). Wir müssen dabei sehr sorgfältig vorgehen:

- Wir müssen A, X und Y unverändert lassen: entweder dürfen wir diese nicht benutzen oder wir müssen ihren Inhalt beiseite legen und später zurückladen.
- Wir dürfen nicht in die Flags eingreifen.
- Wir müssen darauf achten, daß wir BASIC selbst nicht zu sehr verlangsamen.

Das ist eine wichtige Regel. Die letzte Forderung läßt sich oft durch zwei Techniken erfüllen: benutzen Sie einen „Keil" nur, um weitere Befehle im direkten Modus einzubauen und machen Sie von einem speziellen Zeichen Gebrauch, um unsere besonderen Befehle zu identifizieren.

Beim PET/CBM können wir zur Veränderung dieser Unterroutine eine von zwei Stellen auswählen: entweder am Anfang in CHRGET oder nach dem LDA in CHRGOT. Jede Stelle hat ihre Vorteile. Im Bereich von CHRGET brauchen wir weder das A Register noch die Statusflags zu sichern, während CHRGOT diese Arbeit für uns übernimmt. Im Bereich, der CHRGOT folgt, finden wir das Zeichen, das wir untersuchen wollen, im A Register vor.

In jedem Fall handelt es sich um eine anspruchsvolle Aufgabe.

Der VIC-20 und der Commodore 64 erleichtern die Arbeit dadurch, daß sie einen Zeiger bei der Adresse $0308/$0309 zur Verfügung stellen. Dadurch gewinnen wir unmittelbar vor Ausführung eines jeden BASIC Befehls auf Wunsch eine zusätzliche Kontrollmöglichkeit. Wir müssen zwar weiterhin sorgfältig vorgehen, erlangen aber einen größeren Spielraum.

Die mit dem Befehl CHRGOT verbundene Adresse wird oft als TXTPTR (text pointer) Textzeiger bezeichnet. Diese Adresse zeigt immer zu dem BASIC Befehl, der im Augenblick ausgeführt wird. Wenn wir am Lesevorgang des BASIC teilhaben wollen, müssen wir mit TXTPTR umzugehen lernen, um die entsprechende Information zu erhalten. Das geschieht gewöhnlich durch indirekte, indizierte Adressierung. Anschließend muß diese Adresse auf eine geeignete Stelle weisen, wenn wir die Kontrolle an das normale BASIC Programm zurückgeben.

Vorhaben: Hinzufügen eines Befehls

Wir wollen zum VIC und Commodore 64 durch Benutzung des $0308 Zeigers einen einfachen Befehl hinzufügen. Das Zeichen & (ampersand) wird bei den meisten BASIC Programmen nicht benutzt. Wir wollen dessen Bedeutung folgendermaßen festlegen: bei Auftauchen des Code „&" drucke zehnmal das Zeichen * auf den Bildschirm, gefolgt von einem Wagenrücklauf (carriage return).

Genau wie bei unserem Unterbrechungsprogramm wollen wir den alten Adreßinhalt von $0308/$0309 in eine indirekte Adressenstelle kopieren, damit wir eine Verbindung mit den normalen Computerroutinen nach Bedarf herstellen können.

Ein wichtiger Punkt: Wir behalten nach Wunsch dadurch über alles die Kontrolle, daß der TXTPTR unmittelbar vor die nächste Instruktion weist. Wollen wir die Kontrolle an BASIC zurückgeben, müssen wir sicher sein, daß TXTPTR ähnlich positioniert ist.

Hier ist nun unser „Abfang" Befehl:

```
.A 003C LDY #$01
```

Wir benutzen hier die indirekte, indizierte Adressierung, um auf den nächsten Befehl „vorauszuschauen". Wir wollen, indem wir TXTPTR als eine indirekte Adresse benutzen, diese „Vorschau" vornehmen:

```
.A 033E LDA ($7A),Y
```

Da Y gleich Eins ist, schauen wir genau hinter die Adresse, auf die TXTPTR weist:

```
.A 0340 CMP #$26
.A 0342 BEQ $0347
.A 0344 JMP ($03A0)
```

Handelt es sich bei dem Zeichen um ein &, verzweigen wir vorwärts nach $0347. Andernfalls stellen wir durch den indirekten Zeiger die Verbindung zum regulären BASIC Interpretercode her:

```
.A 0347 JSR $0073
```

Wir können CHRGET aufrufen, um den Zeiger auszurichten. Jetzt weist TXTPTR genau auf das &-Zeichen. Wir sind bereit, zehn Sterne zu drucken:

```
.A 034A LDY #$00
.A 034C LDA #$2A
.A 034E JSR $FFD2
.A 0351 INY
.A 0352 CPY #$0A
.A 0354 BCC $034E
.A 0356 LDA #$0D
.A 0358 JSR $FFD2
.A 035B JMP $0344
```

Der obige Code druckt zehnmal einen Stern ($2A) gefolgt von einem RETURN ($0D). Dann kehrt er zum regulären BASIC Interpreter zurück, der hinter dem „&" Zeichen nach einem neuen BASIC Befehl sucht.

Wir müssen jetzt die Verbindung zu unserem Programm herstellen. Dazu schreiben wir den folgenden Code, den wir bei $035E beginnen, damit SYS 862 diesen neuen Befehl (&) ermöglicht:

```
.A 035E LDA $0308
.A 0361 STA $03A0
.A 0364 LDA $0309
.A 0367 STA $03A1
.A 036A LDA #$3C
.A 036C STA $0308
.A 036F LDA #$03
.A 0371 STA $0309
.A 0374 RTS
```

Wenn Sie den Code fertiggestellt und überprüft haben (denken Sie daran, er ist für den VIC und Commodore 64), kehren Sie ins BASIC zurück. Geben Sie NEW ein und schreiben das folgende Programm:

```
100 PRINT 34:&:PRINT 5+6
110 &
120 PRINT "DAS IST ALLES"
```

Wenn Sie RUN eintippen, erhalten Sie in Zeile 100 ein SYNTAX ERROR. Wir haben unseren „&"-Befehl noch nicht eingebaut. Tippen Sie den Befehl SYS 862 und lassen Sie das Programm mit RUN noch einmal laufen. Der „&"-Befehl druckt gehorsam jedesmal, wenn er aufgerufen wird, zehn Sterne.

BASIC zu unterwandern ist keine leichte Aufgabe, aber es ist möglich.

Was wir gelernt haben

- Der Stapel (stack) befindet sich auf Seite 0 von $01FF abwärts bis $0100. Er wird zum Speichern temporärer Informationen benutzt. Ein Programm kann Information auf dem Stapel ablegen und diese später wieder herunternehmen. Der letzte Gegenstand, der auf dem Stapel abgelegt wurde, ist der erste, der wieder heruntergenommen wird.
- Man muß sehr sorgfältig darauf achten, daß das Programm genau die gleiche Anzahl vom Stapel herunternimmt, die es abgelegt hat. Vergewissern Sie sich, daß nicht unbeabsichtigt durch eine Verzweigung oder einen Sprung eine notwendige Stapelaktivität ausgelassen wird. Ein schlecht bearbeiteter Stapel hat oft fatale Folgen für den Programmablauf.
- PHA legt den Inhalt von A auf den Stapel; PLA nimmt vom Stapel und legt im A Register ab. Diese beiden Befehle werden oft dazu benutzt, um den Inhalt von A zeitweise zu sichern. PHP legt den Inhalt des Statusregisters (SR) ab; PLA holt ihn zurück. Diese beiden Befehle werden oft zur Verzögerung von Entscheidungen benutzt.

- JSR legt eine Rückkehradresse (minus 1) auf den Stapel; RTS ruft diese Adresse zurück. Wir können JSR und RTS benutzen, ohne den Stapel und seine Funktion genau zu kennen, da diese beiden Befehle für sich selbst sorgen.
- Durch Unterbrechungen, einschließlich dem Befehl BRK, werden drei Dinge auf den Stapel gelegt; RTI bringt diese zurück, sodaß das unterbrochene Programm unverändert fortfahren kann.
- Bei USR handelt es sich um eine Funktion, im Gegensatz zu SYS, bei dem es sich um einen Befehl handelt. USR geht an eine vorgegebene Adresse, übergibt ein Argument in Form einer Zahl und kann einen Wert zurückgeben. In der Praxis werden USR und SYS auf ähnliche Weise benutzt.
- Im ROM System von Commodore befinden sich Programmteile für Unterbrechungsroutinen, die dafür sorgen, daß die Datenregister (A, X und Y) auf dem Stapel abgelegt werden und daß durch eine indirekte Adresse eine Verzweigung genommen wird, die durch den Benutzer verändert werden kann. Da Unterbrechungen zu jeder Zeit aktiv sind, können sie sogar zu Zeiten genutzt werden, zu denen kein BASIC Programm abläuft.
- Die verschiedenen IA Bausteine (PIA, VIA und CIA) führen viele unterschiedliche Funktionen aus: Aufzeichnung von Ereignissen in speichernden Flags und Kontrolle von Unterbrechungen; Zeitgeberfunktionen und Verbindung von Eingabe- Ausgabekanälen. Die einzelnen Datenblätter müssen zu den unterschiedlichen, teilweise recht komplexen Details zu Rate gezogen werden.
- Während ein BASIC Programm läuft, wird vom Interpreter eine Routine, CHRGET genannt, häufig aufgerufen. Indem wir diese Routine verändern oder zu ihr etwas hinzufügen, können wir das BASIC selbst verändern oder Befehle hinzufügen.

Fragen und Aufgaben

Wenn Sie den Unterbrechungszeiger auf Ihr eigenes Maschinenspracheprogramm weisen lassen, können Sie den gesamten Inhalt der Seit 0 auf den Bildschirm kopieren. Benutzen Sie die Indizierung, beginnen Sie mit X gleich 0 und gehen Sie durch die gesamte Seite 0. Laden Sie dabei den Speicherinhalt und speichern Sie ihn (wiederum indizierend natürlich) im Bildschirmspeicher. Vergessen Sie dabei nicht, Ihr Programm an die reguläre Eingangsadresse für die Unterbrechung anzubinden.

Sie werden einen faszinierenden Bildschirm erhalten. Zeitgeber werden dort ablaufen und, wenn Sie auf der Tastatur tippen, werden verschiedene Werte im Inneren sich verändern. Genießen Sie den Anblick.

Manchmal wird vorgeschlagen, eine brauchbare Art, Informationen an eine Unterroutine zu übergeben, bestehe darin, diese auf einem Stapel abzulegen und dann die Unterroutine aufzurufen. Die Unterroutine kann dann diese Information vom Stapel selbst zurückholen. Was ist an diesem Vorschlag falsch?

Man kann den obigen Vorschlag zwar einbauen, es kostet jedoch eine Menge Arbeit, den Stapel dabei in Ordnung zu halten. Vielleicht möchten Sie sich Stück für Stück durch die notwendige Logik durcharbeiten.

Es gibt einige Benutzerprogramme, die einmal in den Computer eingelesen einen Seitenwechsel (scrolling) beim Listing erlauben. Anders ausgedrückt, kann der Benutzer bei einem Listing eines BASIC Programms, das die Zeilen 250 bis 460 auf dem Bildschirm zeigt, durch Bewegung des Cursors auf die untere Bildschirmzeile und Betätigen der Taste „Pfeil abwärts" neue BASIC Zeilen erscheinen lassen, die auf die Zeile 460 folgen. Das zu programmieren ist nicht einfach, aber hier nun unsere Aufgabe: Glauben Sie, daß man diese Aufgabe mit einem SYS Befehl, einem „Keil" oder einer Unterbrechungstechnik erfüllen kann? Warum?

Der Befehl SYS, aus dem BASIC heraus ausgeführt, wirkt wie der Aufruf einer Unterroutine. Aus diesem Grund muß er eine Adresse auf dem Stapel ablegen, um RTS die Rückkehr ins BASIC zu ermöglichen. Schauen Sie auf dem Stapel nach und versuchen Sie festzustellen, welche Adresse bei Ihrer Maschine benutzt wird, um ins BASIC zurückzukehren.

8. Zeitablauf, Eingabe/Ausgabe und Zusammenfassung

Dieses Kapitel behandelt:

- Wie man die Geschwindigkeit eines Programms abschätzt
- Eingabe und Ausgabe von Band, Disk und Drucker
- Überblick über die Befehle
- Fehlerbeseitigung
- Symbolische Assembler
- Wie kann es weiter gehen

Zeitablauf

Maschinenspracheprogramme erwecken bei vielen Anwendungen den Eindruck, augenblicklich abzulaufen. Die Geschwindigkeit des 650x ist wesentlich größer als die anderer Einrichtungen, den menschlichen Benutzer eingeschlossen. Das Maschinenspracheprogramm befindet sich häufig in irgendeinem Wartezustand: es wartet auf die Tastatur, wartet auf den Drucker, wartet auf die Disk oder es wartet auf den Menschen, der Informationen, die auf dem Bildschirm dargestellt werden, lesen und darauf reagieren muß.

Es kann von Fall zu Fall jedoch wichtig sein, den präzisen Zeitablauf eines Maschinenspracheprogramms zu ermitteln. Dafür kann man sich folgende Faustregel merken:

- Sämtliche Abschätzungen für Zeitabläufe sind ungenau, so lange die Unterbrechungsroutinen noch aktiv sind. Der Einfluß von Unterbrechungen auf den Zeitablauf läßt sich grob dadurch berücksichtigen, daß man zu der Ablaufzeit zehn Prozent addiert.

- Berücksichtigen Sie auch Schleifen. Wird ein Befehl innerhalb einer Schleife zehnmal wiederholt, ist auch sein Zeitbedarf zehnmal höher anzusetzen.

- Die Zykluszeit der meisten Commodore Maschinen beträgt ungefähr eine Mikrosekunde - eine Millionstel Sekunde. Der genaue Wert variiert von Maschine zu Maschine und unterscheidet sich auch zwischen Nordamerika und anderen Regionen.

- Die meisten Befehle laufen mit der am schnellsten vorstellbaren Geschwindigkeit ab. Durch Abzählen der Zykluszeiten läßt sich die Dauer der Befehlsausführung ermitteln. Zum Beispiel benötigt `LDA #$0D` zwei Speicherzyklen, um den Befehl aufzunehmen. Damit kann man die Ausführungszeit bestimmen. `LDA $0500,X` benötigt gewöhnlich vier Speicherzyklen, drei um den Befehl aufzunehmen und einen, um die Daten von Seite 5 zu gewinnen. Ausnahmen: kein Befehl läuft in weniger als zwei Zyklen ab; die Befehle Schiebe/Rotiere, `INC/DEC` und `JSR/RTS` benötigen gewöhnlich eine längere Zeit, als man nach dieser Regel erwarten würde.

- Verzweigungen ergeben einen unterschiedlichen Zeitaufwand in Abhängigkeit davon, ob die Verzweigung eingeschlagen wird (drei Zyklen) oder nicht (zwei Zyklen).
- Bei Überschreitung einer Seitengrenze benötigt der Computer für Berechnungen einen zusätzlichen Zyklus. Verzweigt das Programm von $0FE4 nach $1023, gibt es einen Extrazyklus. Bei dem Befehl LDA $24E7,Y wird ebenfalls ein Extrazyklus eingeschaltet, wenn Y einen Wert größer als $19 enthält.

Genaue Zeitwerte kann man den meisten Befehlstabellen entnehmen.

Wir wollen den Zeitablauf einer einfachen Routine bestimmen. Das folgende Programm verknüpft durch ein logisches AND den Inhalt von 100 Speicherstellen von $17E0 bis $1844:

```
033C LDX #$00
033E LDA #$00
0340 AND $17E0,X
0343 INX
0345 CPX #$64
0347 BCC $0340
0349 RTS
```

Wir können den Zeitablauf folgendermaßen bestimmen:

LDX #$00	- einmal ausgeführt:	2
LDA #$00	- einmal ausgeführt:	2
AND $17E0,X	- 32 mal 4 Zyklen:	128
	68 mal 5 Zyklen (Seitenüberschreitung):	340
INX	- 100 mal 2 Zyklen:	200
CPX #$64	- 100 mal 2 Zyklen:	200
BCC	- 99 mal 3 Zyklen:	297
	1 mal 2 Zyklen (keine Verzweigung):	2
RTS	- 6 Zyklen:	6

Gesamtzeit: 1171 Zyklen oder etwas über eine Tausendstel Sekunde. Wir sollten, um den Einfluß von Unterbrechungen zu berücksichtigen, 10 Prozent addieren. Da es sich um eine Unterroutine handelt, sollten wir weitere 6 Zyklen addieren, die notwendig sind, um JSR auszuführen.

In Fällen, in denen der Zeitablauf kritisch ist, könnte man die Unterbrechung durch SEI ausschalten. Seien Sie vorsichtig: es ist selten notwendig und möglicherweise gefährlich.

Eingabe und Ausgabe

Wir wissen, daß durch Aufruf der Kernroutine CHROUT bei $FFD2 ein ASCII Zeichen auf den Bildschirm geschrieben wird. Wir können die Ausgabe auch zu jedem logischen File umlenken.

Wir haben auch gesehen, daß wir durch Aufruf der Kernroutine GETIN bei $FFE4 eine Eingabe

vom Tastaturpuffer in das A Register bewirken. Wir können auch die Eingabe umlenken, sodaß wir Information aus jedem logischen File beziehen.

Die gleichen Befehle - $FFD2 und $FFE4 - führen nach wie vor die Eingabe und Ausgabe durch. Wir schalten jedoch jeden von ihnen um, damit wir ihn mit einer ausgewählten Einheit oder genaugenommen einem ausgewählten logischen File verbinden. Der File muß eröffnet sein, wir können auf diesen File umschalten und dann nach Belieben zur normalen Eingabe/Ausgabe zurückschalten.

Umschalten der Ausgabe

Wir benutzen die Unterroutine CHKOUT bei Adresse $FFC9, um die Ausgabe auf einen logischen File zu lenken. Wollen wir die Ausgabe wieder auf den Bildschirm bringen, rufen wir Unterroutine CLRCHN bei $FFCC auf. Hierbei handelt es sich nicht um das gleiche wie bei OPEN und CLOSE , sondern wir schalten einfach die Verbindung zu einem File an oder aus. Das können wir sooft machen wie wir wollen.

Unterroutine: CHKOUT
Adresse: $FFC9
Wirkung: Schaltet den Ausgabepfad (benutzt von CHROUT, $FFD2) um, sodaß die Ausgabe zu dem logischen File umgeleitet wird, der durch das X Register bezeichnet ist. Der logische File muß zuvor eröffnet worden sein.

Bei dem Zeichen, das danach durch $FFD2 gesendet wird, handelt es sich gewöhnlich um ASCII (oder PET ASCII). Wird zum Drucker ausgegeben, werden Sonderzeichen - Text/Graphik, Breite - auf die übliche Weise bearbeitet. Genauso können Diskbefehle auf Wunsch über die Sekundäradresse 15 übermittelt werden. Hierbei muß zuvor ein logischer „Kommandokanal"-File eröffnet worden sein.

Register: die Register A und X werden durch den Aufruf von CHKOUT verändert. Sichern Sie vor Aufruf von CHKOUT sämtliche empfindlichen Daten, die sich in diesen Registern befinden.

Status: Statusflags können sich ändern. Beim VIC und Commodore 64 weist die C (carry) Flag darauf hin, daß ein bestimmter Problemfall bei der Anschaltung des Ausgabekanals aufgetreten ist.

Wenn wir die Ausgabe auf den logischen File 1 schalten, müssen wir folgende Schritte ausführen:

1. Lade den Wert 1 in X (LDX #$01).
2. Springe mit JSR zu Adresse $FFC9.

Nachdem die Ausgabe umgeschaltet ist, können wir unter Benutzung der Unterroutine

$FFD2 soviele Zeichen senden, wie wir wollen. Unter Umständen müssen wir die Verbindung zu diesem logischen File abbrechen und zu unserer normalen Ausgabe, dem Bildschirm, zurückkehren. Das geschieht durch die Unterroutine CLRCHN bei Adresse $FFCC.

Unterroutine: CLRCHN
Adresse: $FFCC
Wirkung: Schaltet Eingabe und Ausgabe von jedem logischen File ab und stellt die normale Verbindung zu den Eingabe- und Ausgabekanälen, Tastatur und Bildschirm her. Die logischen Files werden nicht geschlossen und können später angeschaltet werden.

Register: die Register A und X werden durch den Aufruf von CLRCHN verändert. Sichern Sie vor Aufruf von CLRCHN sämtliche empfindlichen Daten, die sich in diesen Registern befinden.

Status: Statusflags können sich ändern. Beim VIC und Commodore 64 weist die C (carry) Flag darauf hin, daß ein bestimmter Problemfall mit der Ausgabe aufgetreten ist.

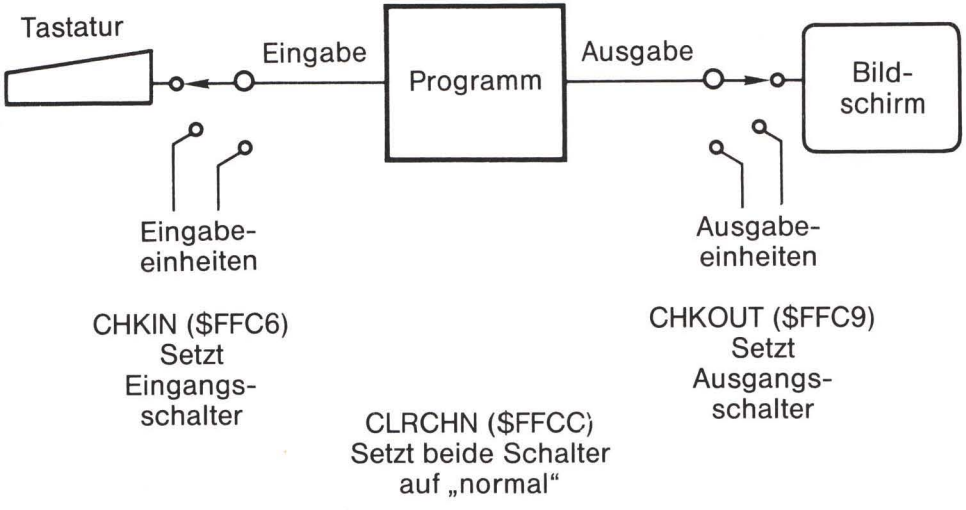


Abbildung 8.1

Das Konzept des „logischen Files“ ist wichtig. Man kann an jedes Ziel senden - Kassette, Drucker, Disk oder Bildschirm - ohne wissen zu müssen, welche Einheit betroffen ist. Ich schicke die Zeichen auf die Reise und das Betriebssystem sorgt dafür, daß sie an ihrem Bestimmungsort ankommen.

Dadurch wird die Arbeit des Maschinensprache-Programmierers vereinfacht. Zeichen zu einem logischen Kanal zu senden ist eine simple Aufgabe. Der Programmierer braucht sich nicht um eine spezielle Codierung zu bemühen, gleichgültig, welche Einheit betroffen ist.

Ein Beispiel für Ausgabe

Wenn wir die Buchstaben HI auf dem Drucker ausgeben wollen, können wir folgendermaßen vorgehen.

Zuerst eröffnen wir den Druckerkanal in BASIC. Wir benutzen dazu die logische Filenummer 1:

```
100 OPEN 1,4
110 SYS 828
120 CLOSE 1
```

Haben Sie keinen Drucker, können Sie den File auch zur Kassette (OPEN 1,1,2) oder zur Disk (OPEN 1,8,3,"O:DEMO,S,W") eröffnen. Dem Maschinenspracheprogramm ist das gleichgültig. Es sendet zur logischen Filenummer 1, egal, um was es sich handelt. Es kann sogar der Bildschirm sein (OPEN 1,3). Schreiben wir das Programm:

```
.A 033C LDX #$01
.A 033E JSR $FFC9
```

Die Ausgabe ist jetzt mit dem logichen File 1 verbunden. Wir wollen HI ausgeben:

```
.A 0341 LDA #$48
.A 0343 JSR $FFD2
.A 0346 LDA #$49
.A 0348 JSR $FFD2
.A 034B LDA #$0D
.A 034D JSR $FFD2
.A 0350 JSR $FFCC
.A 0353 RTS
```

Vergessen Sie nicht, RETURN zu senden - der Drucker braucht es. Nachdem das Maschinenspracheprogramm HI ausgegeben hat, kehrt es zu BASIC zurück und schließt den File. Beachten Sie, daß das Maschinenspracheprogramm sich nicht darum kümmert, wer das HI ausgibt, es sendet die Daten zum logischen File 1.

Umschalten der Eingabe

Wir benutzen die Unterroutine CHKIN bei Adresse $FFC6, um die Eingabe so umzuschalten, daß sie Daten von einem logischen File bezieht. Wollen wir die Eingabe auf die Tastatur zurückschalten, geschieht dies mit der Unterroutine CLRCHN bei $FFCC. Das ist wieder nicht dasselbe wie ein OPEN und CLOSE - wir stellen lediglich die Verbindung zu einem File her und brechen sie ab. Das Ganze kann so oft wir wollen durchgeführt werden.

Unterroutine: CHKIN
Adresse: $FFC6
Wirkung: Schaltet den Eingabepfad (benutzt von GET, $FFE4) um, sodaß die Eingabe von dem logischen File genommen wird, der durch das X Register bezeichnet ist. Der logische File muß zuvor eröffnet worden sein.

Bei dem Zeichen, das danach durch $FFE4 im A Register erhalten wurde, handelt es sich gewöhnlich um ASCII (oder PET ASCII). Wurde eine binäre Null von einem File empfangen, bedeutet das meist folgendes: ein Eingabezeichen, dessen Wert CHR$(0) ist. Damit unterscheidet es sich von einem GET der Tastatur, bei dem eine binäre Null bedeutet „keine Taste betätigt". Bei Zugriff auf einen File wird ST (Adresse $90 bei VIC und Commodore 64, $96 bei den meisten PET/CBM) in seiner gewohnten Funktion benutzt, d.h. es signalisiert ein Fileende oder einen Fehler. Genauso kann man Informationen über den Status der Disk über die Sekundäradresse 15 erhalten. Hierbei muß zuvor ein logischer „Kommandokanal"-File eröffnet worden sein.

Register: die Register A und X werden durch den Aufruf von CHKIN verändert. Sichern Sie vor Aufruf von CHKIN sämtliche empfindlichen Daten, die sich in diesen Registern befinden.

Status: Statusflags können sich ändern. Beim VIC und Commodore 64 weist die C (carry) Flag darauf hin, daß ein bestimmter Problemfall bei der Anschaltung des Eingabekanals aufgetreten ist.

Wenn wir die Eingabe auf den logischen File 1 schalten, müssen wir folgende Schritte ausführen:

1. Lade den Wert 1 in X (LDX #$01).
2. Springe mit JSR zu Adresse $FFC6.

Nachdem die Eingabe umgeschaltet ist, können wir unter Benutzung der Unterroutine $FFE4 soviele Zeichen empfangen, wie wir wollen. Unter Umständen müssen wir die Verbindung zu diesem logischen File abbrechen und zu unserer normalen Eingabe, der Tastatur, zurückkehren. Das geschieht durch die Unterroutine CLRCHN bei Adresse $FFCC. Dabei handelt es sich um die gleiche Unterroutine, die die Ausgabe von einem logischen File abschaltet.

Ein Beispiel für Eingabe

Wir können ein Programm schreiben, um einen Eingabefile von der Disk oder Kassette zu lesen. Zuerst wollen wir den File schreiben. Wir eröffnen den File je nach Typ folgendermaßen:

```
Disk:       OPEN 1,8,3,"O:DEMO,S,W"
Kassette:   OPEN 1,1,1
```

Das läßt sich mit einem direkten Befehl ausführen. Wir wollen nun einige Dinge in den File schreiben:

```
PRINT#1,"HALLO DAS IST EIN TEST"
PRINT#1,"DAS IST DIE LETZTE ZEILE"
CLOSE 1
```

Wenn wir die obigen Befehle richtig eingegeben haben, sollten wir einen vollständigen sequentiellen File auf die Kassette oder Disk geschrieben haben. Bevor wir das Eingabeprogramm in Maschinensprache schreiben, wollen wir uns ansehen, wie wir den File in BASIC zurücklesen können:

```
Disk:      100 OPEN 1,8,3,"DEMO"
Kassette:  100 OPEN 1
           110 INPUT#1,X$
           120 PRINT X$
           130 IF ST=0 GOTO 110
           140 CLOSE 1
```

Wir könnten die Zeilen 110 und 120 auch folgendermaßen schreiben:

```
110 GET#1,X$
120 PRINT X$;
```

Das ähnelt stärker dem logischen Ablauf unseres Maschinenspracheprogramms, da es die Zeichen einzeln übernimmt. Wenn Sie über die Rolle von ST im Ungewissen sind, lesen Sie darüber hinweg. Wir werden dieselbe Variable (bei ihrer Adresse $90 oder $96) für die gleiche Anwendung in Maschinensprache benutzen.

Geben Sie NEW ein und anschließend das folgende Programm:

```
Disk:      100 OPEN 1,8,3,"DEMO"
Kassette:  100 OPEN 1
           110 SYS 828
           120 CLOSE 1
```

Im folgenden wollen wir ausschließlich in Maschinensprache den File einlesen und auf den Bildschirm kopieren. Wir beginnen das Programm bei $033C

```
.A 033C LDX #$01
.A 033E JSR $FFC6
```

Die Eingabe ist jetzt mit dem logischen File 1 verbunden. Wir wollen daraus Informationen beziehen und auf dem Bildschirm ausgeben:

```
.A 0341 JSR $FFE4
.A 0344 JSR $FFD2
```

Wie in BASIC müssen wir ST prüfen. ST kann systemabhängig in einer der beiden Adressen stehen:

```
VIC, Commodore 64:   .A 0347 LDA $90
CBM/PET:             .A 0347 LDA $96
```

Ist ST 0, ist aus dem File noch mehr zu erwarten; wir sollten zurückkehren. Wenn ST nicht 0

ist, kann es sich um einen Fehler handeln oder wir haben das Ende des Files erreicht. In beiden Fällen wollen wir nicht mehr aus dem File lesen.

```
.A 0349 BEQ $0341
.A 034B JSR $FFCC
.A 034E RTS
```

Überprüfen und probieren Sie es aus. Der File wird in Windeseile auf dem Bildschirm dargestellt.

Ein File Transfer Programm

Wir wollen ein Programm schreiben, das Daten von einer beliebigen Einheit auf eine beliebige andere Einheit überträgt. BASIC soll entscheiden, welche Files gehandhabt werden sollen. Sind die Files erst eröffnet, soll das Maschinenspracheprogramm aus einem lesen und auf die gewünschte logische Einheit übertragen.

Sowohl Eingabe als auch Ausgabe gleichzeitig umzuschalten ist keine gute Idee. Gemeint ist damit, sowohl $FFC6 als auch $FFC9 gleichzeitig aufzurufen, ohne jede einzelne Unterroutine über $FFCC zurückzusetzen. Die Kernroutine wird davon nicht betroffen, dagegen werden die peripheren Einheiten durcheinander gebracht. Diese gehen davon aus, jeweils allein Zugang zum Computerbus zu haben. Aus diesem Grund wollen wir folgendem Muster folgen: anschalten, senden oder empfangen, abschalten und dann zur nächsten Einheit übergehen.

Noch etwas. ST meldet uns den Status der zuletzt behandelten Einheit. Bedenken Sie: Wenn wir ein Zeichen eingeben, dann ein Zeichen ausgeben und anschließend den Wert von ST überprüfen, haben wir folgende Schwierigkeit. ST kann uns nichts über die Eingabe aussagen, da die letzte behandelte Einheit die Ausgabe war. Wir wissen deshalb nicht, ob wir uns am Ende des Files befinden oder nicht. Sowohl in Maschinensprache als auch in BASIC müssen wir zur Lösung dieses Problems sorgfältig programmieren.

Behandeln wir zuerst BASIC:

```
100 PRINT "FILE TRANSFER"
110 INPUT "INPUT FROM (DISK, TAPE)";A$
120 IF LEFT$(A$,1)="T" THEN OPEN 1:GOTO 160
130 IF LEFT$(A$,1)<>"D" GOTO 110
140 INPUT "DISK FILE NAME";N$
150 OPEN 1,8,3,N$
160 INPUT "TO (DISK, TAPE, SCREEN)";B$
170 IF LEFT$(B$,1)="S" THEN OPEN 2,3:GOTO 240
180 IF LEFT$(B$,1)="D" GOTO 210
190 IF LEFT$(B$,1)<>"T" GOTO 160
200 IF LEFT$(A$,1)="T" GOTO 160
210 INPUT "OUTPUT FILE NAME";F$
220 IF LEFT$(B$,1)="D" THEN OPEN 2,8,4,"O:"+N$+",S,W"
230 IF LEFT$(B$,1)="T" THEN OPEN 2,1,1,N$
240 SYS xxxx
250 CLOSE 2:CLOSE 1
```

Wir wollen das Ganze für den Commodore 64 behandeln. Für PET/CBM oder VIC-20 können Sie es verändern. Das obige BASIC Programm sollte nicht mehr als 511 Bytes benötigen. Das bedeutet für einen normalen Commodore 64, wir haben ab Adresse $0A00 (dezimal 2560) genügend freien Platz zur Verfügung. Wir verschieben natürlich den Variablenanfang, damit unser Maschinenspracheprogramm von diesen nicht gestört wird.

Als wir die Zeile 240 zum ersten mal eintippten, wußten wir noch nicht, welche SYS Adresse wir benutzen werden. Nachdem das Programm eingetippt ist (mit SYS xxxx in Zeile 240), können wir uns leicht davon überzeugen, daß das Maschinenspracheprogramm bei $0A00 beginnen kann, indem wir den Zeiger für den Variablenanfang überprüfen. Wir verändern nun Zeile 240 in SYS 2560. Nun sind wir zur Eingabe des Maschinenspracheprogramms bereit:

```
.A 0A00 LDX #$01
.A 0A02 JSR $FFC6
.A 0A05 JSR $FFE4
```

Wir haben jetzt im A Register einen Wert aus der Eingabequelle. Auch in ST befindet sich ein Wert, der uns sagt, ob es sich um das letzte Zeichen handelt. Wir wollen diesen Wert in ST überprüfen: wir müssen ihn uns jetzt ansehen, weil sich ST nach der Ausgabe geändert haben wird. Aber wir wollen auf ST noch nicht sofort reagieren; zuerst müssen wir das Zeichen senden, das wir empfangen haben. Prüfen wir also ST und geben wir das Ergebnis auf den Stapel:

```
.A 0A08 LDX $90
.A 0A0A PHP
```

Wenn ST gleich Null ist, ist die Z Flag gesetzt. Wir sichern diese Flag zusammen mit den anderen, bis wir sie vom Stapel zurückrufen können. Wenn Sie dieses Programm für den PET/CBM anpassen wollen, vergessen Sie nicht, daß ST bei dieser Maschine bei der Adresse $96 liegt.

Als nächstes wollen wir die Eingabe durch Aufruf von $FFCC abkoppeln; aber das wird uns das A Register zerstören: wie kann man nun diesen Wert sichern? Durch Übertragen in ein anderes Register oder, indem man A auf den Stapel schreibt. Das wollen wir tun. Es sind jetzt zwei Dinge auf dem Stapel:

```
.A 0A0B PHA
```

Wir können uns nun vom Eingabekanal abkoppeln und an die Ausgabe ankoppeln:

```
.A 0A0C JSR $FFCC
.A 0A0F LDX #$02
.A 0A11 JSR $FFC9
.A 0A14 PLA
```

Das A Register erhält den letzten Wert, der auf dem Stapel gesichert war, zurück und natürlich ist das unser Eingabezeichen. Wir können es also an die Ausgabeeinheit weitergeben:

```
.A 0A15 JSR $FFD2
.A 0A18 JSR $FFCC
```

Jetzt nehmen wir die Bedingung von ST, die wir früher gestapelt haben. Hier kommen die Flags, die wir gespeichert haben:

```
.A 0A1B PLP
```

Wenn die Z Flag gesetzt ist, wollen wir zurückgehen und ein anderes Zeichen holen. Wenn nicht, sind wir fertig und können ins BASIC zurückkehren und BASIC erlauben, für uns die Files zu schließen:

```
.A 0A1C BEQ $0A00
.A 0A1E RTS
```

Wichtig: Bevor das Programm gestartet wird, muß man sicher sein, daß der Zeiger für den Variablenanfang ($002D/$002E) so gesetzt ist, daß er auf Adresse $0A1F weist. Andernfalls zerstören die BASIC Variablen dieses Programm.

Übersicht: Der Befehlssatz

Wir begannen mit Laden, Speichern und Vergleichen bei den drei Datenregistern:

```
LDA LDX LDY
STA STX STY
CMP CPX CPY
```

Die Befehle sind fast identisch in ihrer Ausführung, obgleich nur das A Register den indirekten, indizierten Adressierungsmodus verwendet. Wir fuhren mit den logischen und arithmetischen Routinen fort, die nur das A Register betreffen:

```
AND ORA EOR ADC SBC
```

Die Arithmetik beinhaltet auch die Schiebe- und Rotierbefehle, die auf das A Register oder direkt auf den Speicher wirken können:

```
ASL ROL LSR ROR
```

Man kann den Speicher durch die Befehle Inkrementiere und Dekrementiere direkt verändern. Darüberhinaus gibt es entsprechende Befehle, die auf das X bzw. Y Register wirken:

```
INC DEC
INX DEX
INY DEY
```

Wir können den Programmablauf auch durch Verzweigungen beeinflussen. Diese Verzweigungsbefehle sind alle an Bedingungen geknüpft:

```
BEQ BCS BMI BVS
BNE BCC BPL BVC
```

Mit den Verzweigungsbefehlen kann man nur kurze „Hüpfer" ausführen. Der Sprungbefehl ist dagegen an keine Bedingung geknüpft:

```
JMP
```

Unterroutinen werden mit einem „jump subroutine" aufgerufen. Aus ihnen kehrt man mit einem „return from subroutine" zurück. Man kann auch aus Unterbrechungen zurückkehren:

```
JSR RTS RTI
```

Wir können jede einzelne Flag mit dem entsprechenden Setz- oder Löschbefehl ändern. Manche der Flags kontrollieren interne Prozessorvorgänge; die I (interrupt disable) Flag zum Beispiel blockiert die Unterbrechung und die D (decimal mode) Flag beeinflußt die Art und Weise, in der ADC und SBC mit Zahlen arbeiten.

```
SEC     SEI SED
CLC CLV CLI CLD
```

Wir können Information zwischen dem A Register und X oder Y verschieben. Um die Lage des Stapels zu überprüfen oder festzulegen, können wir den Stapelzeiger nach X oder X zum Stapelzeiger bewegen. Der letztere ist ein leistungsfähiger Befehl. Benutzen Sie ihn also vorsichtig.

```
TAX TAY TSX
TXA TYA TXS
```

Wir können aus dem Stapel Information lesen oder in ihn Information schreiben:

```
PHA PHP
PLA PLP
```

Es gibt einen speziellen Befehl, der meist benutzt wird, um den IA Baustein zu überprüfen:

```
BIT
```

Der BIT Test wird nur für ganz bestimmte Speicherstellen benutzt: eine Indizierung ist nicht erlaubt. Das höhere Bit der Speicherstelle, die geprüft wird, wird direkt in die N Flag geschrieben. Das nächsthöhere Bit (Bit 6) geht unmittelbar in die V Flag. Schließlich wird die Z Flag dann entsprechend gesetzt, wenn die gesetzten Bits einer Speicherstelle mit den zugehörigen gesetzten Bits im A Register übereinstimmen. Auf diese Weise können wir eine Speicherstelle mit BIT $.... prüfen. Folgt diesem Befehl ein BMI, läßt sich damit das oberste Bit prüfen. Mit BVS kann man Bit 6 und mit BNE kann man irgendein bestimmtes Bit oder eine Gruppe von Bits prüfen. Es handelt sich um einen ziemlich spezialisierten Befehl, der aber zur Prüfung der Eingabe- und Ausgabekanäle sehr nützlich ist.

Schließlich der Befehl, der nichts tut, und der BRK Befehl, der eine „falsche Unterbrechung" hervorruft und uns zum Monitor zurückbringt:

```
NOP BRK
```

Das ist der vollständige Befehlsatz. Mit diesem können Sie Programme schreiben, die den Computer alle die Dinge ausführen lassen, die Sie sich ausgedacht haben.

Fehlerbeseitigung

Ist ein Programm geschrieben, folgt als nächster Schritt, nach möglichen Fehlern oder „Wanzen" (bugs) zu suchen. Der Vorgang des Suchens und systematischen Eliminierens von Fehlern wird „debugging" genannt.

Die meisten Programme bestehen aus Teilprogrammen, von denen jedes eine bestimmte Aufgabe zu erfüllen hat. Läuft ein Programm falsch, läßt sich ein Fehler dadurch leichter eingrenzen, daß man darauf achtet, welche Teile des Programms funktionieren und welche nicht. Im Zweifelsfall können Sie Unterbrechungspunkte (breakpoints) in Ihr Programm einfügen. Ersetzen Sie ausgewählte Befehle durch den Befehl `BRK`. Das macht man dadurch, daß man den Opcode eines Befehls durch den Wert `00` ersetzt. Lassen Sie das Programm laufen; wenn es die erste Unterbrechung erreicht, wird es anhalten und der Maschinensprachemonitor wird aktiv werden. Überprüfen Sie nun die Register sorgfältig, um herauszufinden, ob diese die erwarteten Werte enthalten. Untersuchen Sie die Speicherstellen, die das Programm beschrieben haben sollte. An ihrem Inhalt werden Sie sehen, ob das Programm seine Aufgabe korrekt erfüllt hat.

Wenn Sie sich überzeugt haben, daß das Programm bis zu dieser Unterbrechung richtig arbeitet, ersetzen Sie den `BRK` Befehl an dieser Stelle mit dem ursprünglichen Opcode. Geben Sie den Befehl `.G` ein, gefolgt von dieser Adresse, und das Programm läuft bis zum nächsten Unterbrechungspunkt ab. Wenn es für Ihre Untersuchungen hilfreich ist, können Sie sogar vor der Fortsetzung der Programmausführung Speicherinhalte oder Register verändern.

Diese Vorgehensweise kann man soweit treiben, daß man das Programm nach jedem Befehl anhält. Das erfordert zwar einen großen Zeitaufwand, Sie können aber damit jeden einzelnen Programmschritt verfolgen.

Die beste Fehlerbeseitigung erfolgt während des Programmschreibens selbst. Schreiben Sie mit Voraussicht und nicht „superklug". Müssen Sie befürchten, durch eine endlose Schleife gefangen zu werden, fügen Sie einen Test auf die Stoptaste ein (`JSR $FFE1`). Damit behalten Sie weiterhin die Kontrolle über Ihren Computer.

Lernen Sie Ihren Maschinensprachemonitor genau kennen. Der Monitor benutzt eine Reihe von Speicherstellen. Sie werden dann Schwierigkeiten bei der Fehlerbeseitigung in Ihrem Programm bekommen, wenn dieser dieselben Speicheradressen wie Ihr Programm benutzt. Jedesmal nämlich, wenn Sie den Inhalt einer Speicherstelle überprüfen, um zu sehen, was Ihr Programm gemacht hat, werden Sie stattdessen die Arbeitswerte Ihres Monitors vorfinden. Das wäre irreführend und ärgerlich.

Symbolische Assembler

Während unserer Übungen haben wir einen kleinen „nichtsymbolischen" Assembler benutzt, wie wir ihn in einem Maschinensprachemonitor vorfinden. Diese sind für Anfänger gut geeignet. Sie halten sich eng an den Maschinencode und man kann damit die Arbeitsweise der Maschine im Auge behalten.

Schreiben Sie hingegen größere und bessere Programme, sind solche kleinen Assembler weniger geeignet. Verzweigungen nach vorne und Unterroutinen, die wir noch nicht geschrieben haben, zwingen uns, entsprechende Adressen zuerst zu schätzen und die Abschätzung später zu korrigieren. Das führt zu dem möglichen Fehler, eine Adresse falsch einzugeben ($0345 statt $0354). Das Programm würde dann nicht funktionieren.

Um beim Schreiben anspruchsvollerer Programme Unterstützung zu finden, sollten wir uns lieber käuflichen Assemblersystemen zuwenden, die sogenannte Label oder symbolische Adressen zulassen. Wenn wir damit ein Programm schreiben möchten, das eine Unterroutine zur Eingabe von Zahlen aufruft - eine solche Unterroutine haben wir bisher noch nicht geschrieben - können wir programmieren JSR NUMIN. Schreiben wir dann die Unterroutine, stellen wir den Erkennungslabel NUMIN an deren Anfang. Wenn Ihr Programm anschließend assembliert wird, wird die richtige Adresse von NUMIN bestimmt. Diese Adresse wird dann entsprechend eingefügt.

Dadurch spart man Arbeit. Fehler werden vermieden. Symbolische Assembler haben hingegen eine weitere positive Eigenschaft: sie unterstützen Sie bei der Dokumentation und erlauben, das Programm auf den neuesten Stand zu bringen.

Ihre Assemblierung können Sie auf dem Drucker ausgeben. Das ermöglicht Ihnen, Programme zu untersuchen und zu kommentieren und Einzelheiten für spätere Referenzen abzulegen. Der Assembler erlaubt Ihnen auch, Kommentare einzufügen, die die Lesbarkeit eines Listing verbessern, das Maschinenspracheprogramm jedoch nicht beeinflussen.

Das einmal geschriebene Quellprogramm läßt sich speichern und später wieder benutzen. Erscheint Ihnen eine Programmänderung notwendig, laden Sie den Quellcode von der Kassette oder Disk, bringen die entsprechenden Änderungen an und assemblieren es erneut. Auf diese Weise kann man Programme auf einfache Weise korrigieren oder verbessern.

Wie kann es weitergehen?

Eigentlich auf vielen unterschiedlichen Wegen. Bis jetzt haben wir uns eine gewisse Sicherheit verschafft, indem wir versuchten, ein Gefühl dafür zu bekommen, wie die verschiedenen Details zusammenwirken. Jetzt beginnt das eigentliche Vergnügen, die kreative Art der Programmierung.

Benutzer haben unterschiedliche Interessen. Sie könnten mathematische Operationen durchführen wollen. Sie könnten daran interessiert sein, in BASIC Programme einzugreifen, indem Sie diese analysieren, durchsuchen oder umnumerieren. Was Ihnen auch gefällt, Sie könnten sich auch auf Musik, Graphik oder bewegte Bilder stürzen. Durch die Maschinensprache wird Ihnen die Tür zu allem geöffnet. Es macht Spaß. Die Geschwindigkeit ermöglicht spektakuläre Effekte. Vielleicht haben Sie vor, sich mit der Hardware zu beschäftigen und neue Einheiten an Ihren Computer anzuschließen. Das Verständnis der Maschinensprache und der IA Bausteine im besonderen sind dazu hilfreich. Die Möglichkeiten sind endlos.

Selbst wenn Sie nicht sogleich planen, neue Programme in Maschinensprache zu schreiben, haben Sie einen Einblick in die Arbeitsweise Ihrer Maschine gewonnen. Alles, was die Ma-

schine durchführt - BASIC, Kernroutinen, alles - hängt entweder mit Hardware oder Maschinensprache zusammen.

Mit den elementaren Begriffen, die wir hier eingeführt haben, sind Sie in der Lage, fortgeschrittenere Abhandlungen leichter zu verstehen. Viele Programmierbücher behandeln den 650x Baustein in einer abstrakten Weise. Das ist für den Anfänger schwer zu verstehen. Er kann nur schwer erkennen, wie die Befehle mit der Architektur einer richtigen Maschine zusammenpassen oder wie die Programme im Einzelfall innerhalb des Computers angelegt sind. Jetzt aber sind Sie in der Lage, ein abstraktes Programmstück aufzunehmen und es in Ihr System einzubauen.

Bei Ihren ersten Schritten in der Programmierung in Maschinensprache tauchten viele Dinge gleichzeitig auf. Sie mußten den Monitor benutzen lernen. Sie mußten eine Menge über den Aufbau Ihrer Maschine lernen und Sie mußten lernen, alle Teile in Zusammenhang zu bringen. Sie werden einige Zeit brauchen, bis Sie sich an die Lawine von Information gewöhnt haben - aber langsam fügen sich die Einzelteile zu einem Bild zusammen. Vielleicht haben Sie einen genaueren Überblick über den gesamten Computer erlangt: Hardware, Software, Sprachen und Anwendungen.

Was wir gelernt haben

- Die Laufzeit von Maschinenspracheprogrammen läßt sich recht genau abschätzen. In vielen Fällen ist jedoch die Maschinensprache so schnell, daß detaillierte Geschwindigkeitsberechnungen unnötig sind.
- Wir können die Eingabe von anderen Einheiten als der Tastatur durch Umschalten der angesprochenen Eingabeeinheit bearbeiten. Wurde ein Eingabekanal als File eröffnet, können wir uns durch `JSR $FFC6` an ihn anschließen und durch `JSR $FFCC` davon abkoppeln.
- Wir können die Ausgabe auf andere Einheiten als den Bildschirm durch Umschalten der angesprochenen Ausgabeeinheit bearbeiten. Wurde ein Ausgabekanal als File eröffnet, können wir uns durch `JSR $FFC9` an ihn anschließen und durch `JSR $FFCC` davon abkoppeln.
- Wurde eine Eingabe oder eine Ausgabe umgeschaltet, können wir auf übliche Weise mit der Unterroutine bei `$FFE4` empfangen oder mit der Unterroutine bei `$FFD2` senden.
- Achten Sie darauf, daß Sie die Kopplung an einen Kanal nicht verwechseln mit der Eröffnung eines Files. Bei einem typischen Programm eröffnen wir einen File nur einmal, wir können aber die Kopplung an ihn und die Abkopplung Hunderte von Malen durchführen, wenn wir Daten lesen oder schreiben.
- Sie sind allen Befehlen des 650x Mikroprozessors begegnet. Ihre Zahl reicht für eine vielseitige Programmierung aus, es sind jedoch nicht so viele, daß Sie sie nicht alle behalten könnten. Sie haben einen vielversprechenden Einstieg in die Kunst und Wissenschaft der Maschinenspracheprogrammierung getan.

Fragen und Aufgaben

Schreiben Sie ein Programm, mit dem Sie einen sequentiellen File lesen und die Häufigkeit bestimmen können, mit der der Buchstabe „A" (hex 41) in diesem File vorkommt. Benutzen Sie den Befehl PEEK in BASIC, um den Wert auszudrucken. Sie könnten davon ausgehen, daß „A" nicht häufiger als 255 mal vorkommt.

Schreiben Sie das Programm um, um das Vorkommen des Zeichens RETURN ($0D) in einem sequentiellen File zu zählen. Erlauben Sie, daß es 65535mal vorkommt. Können Sie diesen Zahlenwert mit einer Bedeutung versehen?

Schreiben Sie ein Programm, das GUTES NEUES JAHR zehnmal auf dem Drucker ausgibt.

Wenn Sie ein Disksystem besitzen, wissen Sie, daß Sie ein Programm namens UNSINN durch folgende Sequenz löschen können:
OPEN 15,8,15:PRINT#15,"SO:UNSINN". Wandeln Sie den Befehl PRINT# in Maschinensprache um und schreiben Sie ein Programm zum Löschen von UNSINN. Vorsicht: löschen Sie kein Programm, das Sie noch benötigen.

Schreiben Sie ein „Schreibmaschinenprogramm", um eine Textzeile von der Tastatur zu lesen und sie dann auf den Drucker zu übertragen. Das Programm wird nützlicher sein, wenn Sie das, was getippt wurde, auf dem Bildschirm anzeigen und wenn Sie einen zusätzlichen Code schreiben, um die Taste DELETE zu berücksichtigen.

Anhang A
Befehlssatz des 6502/6510/6509/7501

Die vier Bausteine unterscheiden sich lediglich in der Benutzung der Adressen 0 und 1:

Beim 6502 gehören diese Adressen zum normalen Speicher.

Beim 6510 und 7501 handelt es sich bei Adresse 0 um ein Richtungsregister und bei Adresse 1 um ein Eingabe/Ausgabe Register, das für Dinge wie Kassettenrekorder oder Speicherkontrolle benutzt wird.

Beim 6509 wird Adresse 0 dazu benutzt, die Programmausführung auf eine neue Speicherbank umzuschalten. Adresse 1 wird benutzt, um die Speicherbank, die von den beiden Befehlen `LDA (..),Y` und `STA (..),Y` angesprochen wird, umzuschalten.

Adressierungsarten

Akkumulator Adressierung - diese Adressierungsart liegt bei einem Einbyte Befehl vor, wobei man davon ausgeht, daß die Operation auf den Akkumulator wirkt.

Direkte Adressierung - bei der direkten Adressierung steckt der Operand im zweiten Byte des Befehls, weshalb keine weitere Speicheradressierung notwendig ist.

Absolute Adressierung - bei der absoluten Adressierung bezeichnet das zweite Byte des Befehls die acht niederwertigen Bits der Adresse, während das dritte Byte die acht höherwertigen Bits bezeichnet. Auf diese Weise ist es durch absolute Adressierung möglich, die gesamten 65K Bytes des adressierbaren Speichers anzusprechen.

Adressierung der Seite 0 - die Seite 0 Befehle erlauben einen kürzeren Code und schnellere Ausführungszeit, indem sie lediglich das zweite Byte des Befehls aufgreifen und annehmen, daß das höhere Adressbyte null ist. Sorgfältige Ausnutzung der Seite 0 kann die Effizienz des Codes signifikant erhöhen.

Indizierte Seite 0 Adressierung - (`X`, `Y` Indizierung) - diese Adressierungsart wird im Zusammenhang mit dem Indexregister benutzt und als „Zero Page,X" oder „Zero Page,Y" bezeichnet. Die effektive Adresse wird durch Addition des zweiten Byte mit dem Inhalt des Indexregisters berechnet. Da es sich dabei um eine Art der Adressierung der Seite 0 handelt, bezieht sich der Inhalt des zweiten Byte auf eine Stelle in Seite 0. Auf Grund der Natur der Seite 0 Adressierungsart wird darüberhinaus zu den höherwertigen acht Bit des Speichers kein Carry addiert, weshalb eine Seitenüberschreitung nicht stattfindet.

Indizierte absolute Adressierung - (`X`, `Y` Indizierung) - diese Adressierungsart wird im Zusammenhang mit dem Indexregister benutzt und als „Absolute,X" und „Absolute,Y" bezeichnet. Die effektive Adresse wird durch Addition des Inhalts von `X` und `Y` mit der Adresse, die im zweiten und dritten Byte des Befehls enthalten ist, gebildet. Diese Art erlaubt, daß das Indexregister den Index oder Zählwert enthält und der Befehl die Basisadresse. Mit dieser Indizie-

rungsart kann man jede Speicherstelle ansprechen und durch den Index mehrere Felder verändern. Das wirkt sich auf eine Verringerung des Programmieraufwands und auf eine Verkürzung der Ausführungszeit aus.

Implizierte Adressierung - bei der implizierten Adressierungsart wird die Adresse, die den Operand enthält, implizit im Operationscode des Befehls angegeben.

Relative Adressierung - die relative Adressierung wird ausschließlich bei Verzweigungsbefehlen benutzt und bezeichnet das Ziel für eine bedingte Verzweigung.
Das zweite Byte des Befehls wird zum Operanden, welcher einen „Offset" bildet, der zum Inhalt der acht unteren Bits des Befehlszählers addiert wird, wenn der Befehlszähler auf den nächsten Befehl gesetzt wurde. Der Bereich dieses Offset beträgt -128 bis +127 Bytes vom nächsten Befehl ab.

Indizierte indirekte Adressierung - bei der indizierten indirekten Adressierung (Indirect,X genannt) wird das zweite Byte des Befehls zum Inhalt des X Indexregisters addiert, wobei Carry verloren geht. Das Ergebnis dieser Addition weist auf eine Speicherstelle auf Seite 0, deren Inhalt die niederwertigen acht Bits der effektiven Adresse darstellt. Die beiden Speicherstellen, die die höher- und niederwertigen Bytes der effektiven Adresse bezeichnen, müssen sich auf Seite 0 befinden.

Indirekte indizierte Adressierung - bei der indirekten indizierten Adressierung (Indirect,Y genannt) zeigt das zweite Byte des Befehls auf eine Speicherstelle der Seite 0. Der Inhalt dieser Speicherstelle wird zum Inhalt des Y Indexregisters addiert, wobei das Ergebnis die niederwertigen acht Bits der effektiven Adresse darstellt. Der Übertrag (Carry) aus dieser Addition wird zum Inhalt der folgenden Speicherstelle der Seite 0 hinzugefügt, woraus die höherwertigen acht Bits der effektiven Adresse gebildet werden.

Absolute indirekte Adressierung - das zweite Byte des Befehls enthält die niederwertigen acht Bits einer Speicherstelle. Die höherwertigen acht Bits dieser Speicherstelle stecken im dritten Byte des Befehls. Der Inhalt der gesamten bezeichneten Speicherstelle stellt das niederwertige Byte der effektiven Adresse dar. Die darauffolgende Speicherstelle enthält das höherwertige Byte der effektiven Adresse. Diese werden in die sechzehn Bits des Befehlszählers geladen.

Befehlssatz - Alphabetische Reihenfolge

ADC Addiere Speicher zum Akkumulator mit Übertrag (Carry)
AND Logisches „UND": Speicher mit Akkumulator
ASL Schiebe um ein Bit links (Speicher oder Akkumulator)

BCC Verzweige wenn Übertrag (Carry) gelöscht
BCS Verzweige wenn Übertrag (Carry) gesetzt
BEQ Verzweige wenn Ergebnis null
BIT Vergleiche Bits im Speicher mit Akkumulator
BMI Verzweige wenn Ergebnis minus
BNE Verzweige wenn Ergebnis ungleich null

BPL Verzweige wenn Ergebnis positiv
BRK Erzwinge Interrupt
BVC Verzweige wenn Überlauf (Overflow) gelöscht
BVS Verzweige wenn Überlauf (Overflow) gesetzt

CLC Lösche Carry
CLD Lösche Dezimalmodus
CLI Lösche Interruptsperre
CLV Lösche Überlauf (Overflow)
CMP Vergleiche Speicher mit Akkumulator
CPX Vergleiche Speicher mit Index X
CPY Vergleiche Speicher mit Index Y

DEC Erniedrige Speicher um eins
DEX Erniedrige Index X um eins
DEY Erniedrige Index Y um eins

EOR Exklusiv „ODER": Speicher mit Akkumulator

INC Erhöhe Speicher um eins
INX Erhöhe Index X um eins
INY Erhöhe Index Y um eins

JMP Springe auf neue Speicherstelle
JSR Springe auf neue Speicherstelle, behalte Rückkehradresse

LDA Lade Akkumulator mit Speicher
LDX Lade Index X mit Speicher
LDY Lade Index Y mit Speicher
LSR Schiebe um ein Bit rechts (Speicher oder Akkumulator)

NOP Keine Operation

ORA Logisches „ODER": Speicher mit Akkumulator

PHA Lege Akkumulator auf Stapel
PHP Lege Prozessor Status auf Stapel
PLA Hole Akkumulator vom Stapel
PLP Hole Prozessor Status vom Stapel

ROL Rotiere ein Bit links (Speicher oder Akkumulator)
ROR Rotiere ein Bit rechts (Speicher oder Akkumulator)
RTI Kehre von Interrupt zurück
RTS Kehre von Unterroutine zurück

SBC Subtrahiere Speicher und Übertrag vom Akkumulator
SEC Setze Übertrag (Carry)
SED Setze Dezimalmodus
SEI Setze Interruptsperre
STA Speichere Akkumulator in Speicher

STX Speichere Index X in Speicher
STY Speichere Index Y in Speicher

TAX Übertrage Akkumulator nach Index X
TAY Übertrage Akkumulator nach Index Y
TSX Übertrage Stapelzeiger nach Index Y
TXA Übertrage Index X nach Akkumulator
TXS Übertrage Index X nach Stapelzeiger
TYA Übertrage Index Y nach Akkumulator

Programmiermodell

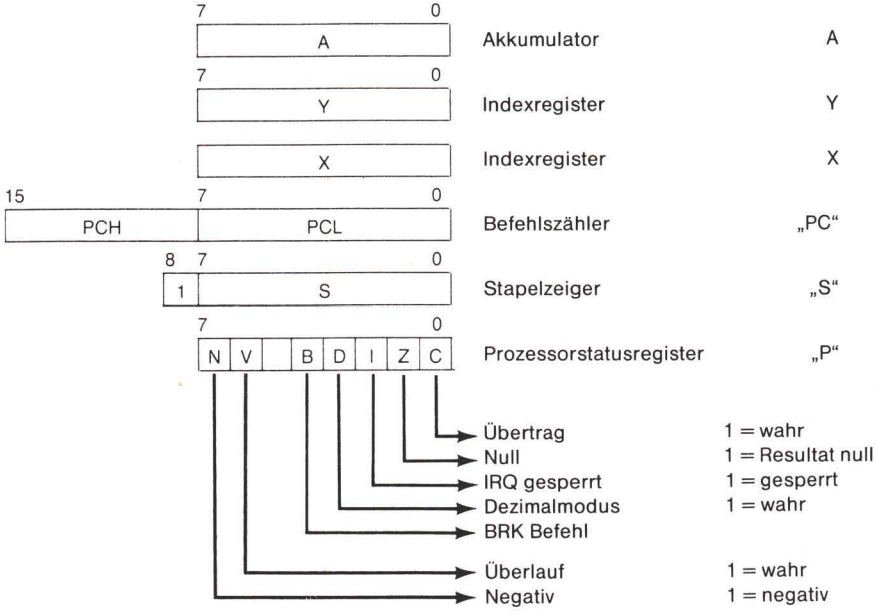


Abbildung A.1

	IMM 2	ZPAG 2	Z,X 2	(I,X) 2	(I),Y 2	ABS 3	A,X 3	A,Y 3
ORA	09	05	15	01	11	0D	1D	19
AND	29	25	35	21	31	2D	3D	39
EOR	49	45	55	41	51	4D	5D	59
ADC	69	65	75	61	71	6D	7D	79
STA		85	95	81	91	8D	9D	99
LDA	A9	A5	B5	A1	B1	AD	BD	B9
CMP	C9	C5	D5	C1	D1	CD	DD	D9
SBC	E9	E5	F5	E1	F1	ED	FD	F9

Op Code endet auf –1, –5, –9 oder –D

	IMM 2	ZPAG 2	Z,X 2	Z,Y 2	ABS 3	A,X 3	A,Y 3
ASL		06	16		0E	1E	
ROL		26	36		2E	3E	
LSR		46	56		4E	5E	
ROR		66	76		6E	7E	
STX		86		96	8E		
LDX	A2	A6		B6	AE		BE
DEC		C6	D6		CE	DE	
INC		E6	F6		EE	FE	

Op Code endet auf –2, –6 oder –E

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

Verzweigungen – 0

	ABS	(IND)
JSR	20	
JMP	4C	6C

Sprünge

	IMM 2	ZPAG 2	Z,X 2	ABS 3	A,X 3
BIT		24		2C	
STY		84	94	8C	
LDY	A0	A4	B4	AC	BC
CPY	C0	C4		CC	
CPX	E0	E4		EC	

Verschiedenes – 0, – 4, – C

	0–	1–	2–	3–	4–	5–	6–	7–	8–	9–	A–	B–	C–	D–	E–	F–
–0	BRK				RTI		RTS									
–8	PHP	CLC	PLP	SEC	PHA	CLI	PLA	SEI	DEY	TYA	TAY	CLV	INY	CLD	INX	SED
–A	ASL-A		ROL-A		LSR-A		ROR-A		TXA	TXS	TAX	TSX	DEX		NOP	

Einzel-Byte Op Codes – 0, – 8, – A

Abbildung A.2

Instruktionen		Immediate			Absolute			Zero Page			Accum.			Implied			(Ind.) X			(Ind.) Y			Z. Page, X			Abs. X			Abs. Y			Relative			Indirect			Z. Page, Y			Bedingung
Mnemonic	Operation	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N Z C I D V
ADC	A + M + C → A (4) (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										✓ ✓ ✓ – – ✓
AND	A∧M →A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										✓ ✓ – – – –
ASL	C ← 7 0 ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													✓ ✓ ✓ – – –
BCC	BRANCH ON C =0 (2)																															90	2	2							– – – – – –
BCS	BRANCH ON C =1 (2)																															B0	2	2							– – – – – –
BEQ	BRANCH ON Z =1 (2)																															F0	2	2							– – – – – –
BIT	A∧M				2C	4	3	24	3	2																															M7 ✓ – – – M6
BMI	BRANCH ON N =1 (2)																															30	2	2							– – – – – –
BNE	BRANCH ON Z =0 (2)																															D0	2	2							– – – – – –
BPL	BRANCH ON N =0 (2)																															10	2	2							– – – – – –
BRK	(See Figure A-1)													00	7	1																									– – – 1 – –
BVC	BRANCH ON V =0 (2)																															50	2	2							– – – – – –
BVS	BRANCH ON V =1 (2)																															70	2	2							– – – – – –
CLC	0 →C													18	2	1																									– – 0 – – –
CLD	0 →D													D8	2	1																									– – – – 0 –
CLI	0 →I													58	2	1																									– – – 0 – –
CLV	0 →V													B8	2	1																									– – – – – 0
CMP	A − M (1)	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										✓ ✓ ✓ – – –
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																															✓ ✓ ✓ – – –
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																															✓ ✓ ✓ – – –
DEC	M − 1 →M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													✓ ✓ – – – –
DEX	X − 1 →X													CA	2	1																									✓ ✓ – – – –
DEY	Y − 1 →Y													B8	2	1																									✓ ✓ – – – –
EOR	A∨M →A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										✓ ✓ – – – –
INC	M + 1 →M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													✓ ✓ – – – –
INX	X + 1 →X													E8	2	1																									✓ ✓ – – – –
INY	Y + 1 →Y													C8	2	1																									✓ ✓ – – – –
JMP	JUMP TO NEW LOC				4C	3	3																													6C	5	3				– – – – – –
JSR	(See Figure A-2) JUMP SUB				20	6	3																																			– – – – – –
LDA	M →A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										✓ ✓ – – – –

Abbildung A.3

Instruktionen		Immediate			Absolute			Zero Page			Accum.			Implied			(Ind.) X			(Ind.) Y			Z, Page, X			Abs. X			Abs. Y			Relative			Indirect			Z, Page, Y			Bedingung
Mnemonic	Operation	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N Z C I D V
LDX	M→X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	✓ ✓ – – – –
LDY	M→Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3												✓ ✓ – – – –
LSR	0→[7 0]→C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3												0 ✓ ✓ – – –
NOP	NO OPERATION													EA	2	1																								– – – – – –
ORA	AVM→A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	5	15	4	2	1D	4	3	19	4	3										✓ ✓ – – – –
PHA	A→MS S −1→S													48	3	1																								– – – – – –
PHP	P→MS S − 1→S													08	3	1																								– – – – – –
PLA	S ←1→S MS→A													68	4	1																								✓ ✓ – – – –
PLP	S ←1→S MS→P													28	4	1																								(RESTORED)
ROL	[→[7 0]←[C]←]				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3												✓ ✓ ✓ – – –
ROR	[→[C]→[7 0]←]				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3												✓ ✓ ✓ – – –
RTI	(See Figure A-1) RTRN INT													40	6	1																								(RESTORED)
RTS	(See Figure A-1) RTRN SUB													60	6	1																								– – – – – –
SBC	A − M − $\overline{C}$→A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										✓ ✓ (3) – – ✓
SEC	1→C													38	2	1																								– – 1 – – –
SED	1→D													F8	2	1																								– – – – 1 –
SEI	1→I													78	2	1																								– – – 1 – –
STA	A→M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										– – – – – –
STX	X→M				8E	4	3	86	3	2																													96	4	2	– – – – – –
STY	Y→M				8C	4	3	84	3	2													94	4	2																– – – – – –
TAX	A→X													AA	2	1																								✓ ✓ – – – –
TAY	A→Y													A8	2	1																								✓ ✓ – – – –
TSX	S→X													BA	2	1																								✓ ✓ – – – –
TXA	X→A													8A	2	1																								✓ ✓ – – – –
TXS	X→S													9A	2	1																								– – – – – –
TYA	Y→A													98	2	1																								✓ ✓ – – – –

(1) ADD 1 TO "N" IF PAGE BOUNDARY IS CROSSED.
(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE.
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE.
(3) CARRY NOT = BORROW.
(4) IF IN DECIMAL MODE Z FLAG IS INVALID
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT.

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
MS MEMORY PER STACK POINTER

+ ADD
− SUBTRACT
∧ AND
∨ OR
∀ EXCLUSIVE OR

✓ MODIFIED
– NOT MODIFIED
M7 MEMORY BIT 7
M6 MEMORY BIT 6
N NO. CYCLES
NO. BYTES

Abbildung A.4

Anhang B
Einige Eigenschaften der Commodore Maschinen

PET - Original ROM

Der erste PET. Man erkennt ihn nach dem Einschalten an der Bildschirmnachricht:

*** COMMODORE BASIC ***

Dabei werden Sterne ohne eine Identifizierungsnummer hinter dem Wort BASIC benutzt.

Die Originalmaschine kann durch Einstecken eines Satzes von ROM Bausteinen, eines Upgrade ROM, verbessert werden. Das ist eine gute Idee, da die Originallogik nicht mit einer Disk arbeiten kann, bei Kassette mit Datenfiles schlecht umgeht, keinen eingebauten Machinensprachemonitor besitzt und eine Architektur der Seite 0 enthält, die sich signifikant von allen späteren PET/CBMs unterscheidet. Die BASIC Sprache ist bei dieser Einheit ebenfalls beschränkt. Beispielsweise können Arrays nicht mehr als 256 Elemente enthalten.

Diese frühe Maschine ist selten geworden.

PET/CBM - Upgrade ROM

Der erste PET, der mit einer Disk arbeiten kann. Man erkennt ihn nach dem Einschalten an der Bildschirmnachricht:

COMMODORE BASIC

Dabei werden die Nummer-Symbole benutzt (auch octothorpe genannt).

Im Gegensatz zur vorhergehenden Maschine ist die Logik sauberer angelegt. Ihre interne Struktur ähnelt mehr den späteren PET/CBM Einheiten (den 4.0 Maschinen), mit denen sie mehr gemein hat.

Sie hat keine speziellen Disk Befehle, wie CATALOG, SCRATCH oder DLOAD (4.0 Disk Befehle). Bei diesen handelt es sich um Befehle, die der Bequemlichkeit dienen. Das Upgrade ROM kann sonst alles, was die späteren Einheiten können.

Maschinen mit dem Upgrade ROM enthalten im BASIC ein Ärgernis: unter gewissen Umständen müssen Stringvariable geordnet werden, wobei eine Technik der „Abfallsammlung" (garbage collection) verwandt wird. Dadurch wird automatisch bei Bedarf Platz geschaffen. Läuft diese Routine ab, friert die Maschine gewissermaßen ein und scheint für eine gewisse Zeit tot zu sein. Das kann einige Sekunden oder bis über eine halbe Stunde dauern.

PET/CBM – 4.0 ROM und 80 Zeichen

Diese Maschinenklasse wurde innerhalb der Commodorelinie für Jahre zu einem Meilenstein. Man kann sie beim Einschalten an der Bildschirmnachricht:

*** COMMODORE BASIC 4.0 ***

erkennen. Zum erstenmal erscheint in der Nachricht eine Zahl.

Diese Maschinen zeichnen sich durch neue BASIC Diskbefehle aus (CATALOG usw.) und durch Beseitigung der garbage collection Verzögerung. Ihre interne Architektur, besonders die der Seite 0, ähnelt stark den früheren Computern mit Upgrade ROM.

Einige Zeit nach der ersten Produktion von 40 Zeilen Maschinen wurden 80 Zeilen Maschinen als auch neue 40 Zeilen Versionen eingeführt, „fat 40" genannt. Die späteren Maschinen unterscheiden sich durch neue Bildschirm/Tastatur Eigenarten. Dabei fällt besonders die automatische Wiederholungsfunktion der Cursorbewegung auf.

Anschließend standen zwei Maschinen zur Verfügung, deren Speicher erweitert war. Der 8096 war mit 96K RAM ausgerüstet. Die zusätzlichen 64K wurden bei Bedarf durch Speicherumschaltung in Blöcken von 16K zum Speicher zugeschaltet. Sie war auch durch einen zusätzlichen Mikroprozessor (den 6809) ausgezeichnet, der hauptsächlich zur Implementierung von Hochsprachen benutzt wurde. Sowohl der 8096 als auch der SuperPET können wie gewöhnliche CBM 8032 Computer benutzt werden. Der zusätzliche Speicher kann ignoriert werden.

VIC-20

Der VIC-20 stellte für Commodore ein neues Entwurfskonzept dar. Farbe, Graphik und Ton wurden in den Computer eingebaut. Die Speicherarchitektur änderte sich radikal. Im Vergleich zu früheren PET/CBM Computern wurden die Speicherstellen der Seite 0 signifikant verschoben.

Das BASIC kehrte zum Stil des Upgrade ROM zurück – es gibt keine speziellen Diskbefehle und keine möglicherweise langsame garbage collection. Im Gegensatz dazu wurde das BASIC nicht angepaßt. Alle Funktionen und Eigenschaften wurden beibehalten und einige attraktive, neue Bildschirm Editiermöglichkeiten hinzugefügt wie etwa automatische Tastenwiederholung.

Der VIC besitzt keinen Maschinensprachemonitor. Man muß diesen laden. Der SYS Befehl besitzt eine neue attraktive Eigenschaft. Man kann damit die Register A, X und Y „vorladen", indem man Werte in die Adresse 780, 781 und 782 durch POKE eingibt. Adresse 783 kann auch zum Setzen des Statusregisters benutzt werden, was jedoch gefährlich ist. Wenn man nicht sorgfältig vorgeht, kann man damit unbeabsichtigt den Dezimalmodus oder die Flags für Interruptsperre setzen.

Der VIC-20 ist bei der Arbeit in Maschinensprache etwas schwieriger. Je nach dem, mit wieviel zusätzlichem Speicher er ausgerüstet ist (0K, 3K oder 8K und mehr), verändert sich die Speicherstelle für den BASIC-Anfang oder der Bildschirmspeicherbereich.

Commodore 64

Der Commodore 64 besitzt große Ähnlichkeit mit dem VIC-20. Besonders die Organisation der Seite 0 gleicht genau dem VIC. Der Commodore 64 ist mit einem 6510 Mikroprozessor ausgerüstet. Die Adressen 0 und 1 sind für die Umschaltung der Speicherbank reserviert.

Das BASIC ist mit dem des VIC identisch - es gibt keine speziellen Diskbefehle und keine möglicherweise langsame garbage collection. Auch er besitzt keinen eingebauten Maschinensprachemonitor, man muß einen laden. Der SYS Befehl erlaubt wie beim VIC auf Wunsch ein Vorladen der Register A, X und Y.

Der Commodore 64 hat eine stabilere Architektur als der VIC. Das BASIC beginnt an einem festen Platz und der Bildschirm befindet sich immer ab hex 0400, außer wenn man ihn verschiebt. Zwischen $C000 und $CFFF befindet sich eine Speicherbank, die vom Computersystem nicht genutzt wird. Diese ist zur Plazierung von Maschinenspracheprogrammen nützlich.

Beim Commodore 64 handelt es sich um die erste Commodore Maschine, bei der es manchmal wünschenswert ist, ohne irgendein BASIC ausschließlich in Maschinensprache zu schreiben. Man kann das BASIC herausnehmen, um zusätzlichen RAM Speicher frei zu machen. Große Anwendungsprogramme (Textverarbeitung, Tabellenkalkulationen usw.) erfordern dies manchmal.

Commodore PLUS/4

Er ähnelt in vielerlei Hinsicht dem Commodore 64. Bei dem Prozessor handelt es sich um einen 7501, der den gleichen Befehlssatz wie der 6502 hat. Bildschirmspeicher und BASIC RAM sind etwas nach oben verschoben. Das BASIC selbst ist deutlich erweitert.

Farbe und Ton sind im Vergleich zum Commodore 64 unterschiedlich implementiert.

Der eingebaute Maschinensprachemonitor erhielt erweiterte Möglichkeiten wie Assembler und Disassember. Er ist für Maschinenspracheprogrammierer sehr gut geeignet.

Die Speicheranordnung ist ausgeklügelter als bei früheren Maschinen. Große Programme können einen tieferen Einblick in die detaillierte Architektur der Maschine erfordern.

B Serie

Die B-128, B-256, der CBM-128 und der CBM-256 wurden als Nachfolger des 80 Zeilen CBM entworfen. Die Architektur wurde radikal verändert: der Prozessor ist ein 6509, der Speicher wird über banks umgeschaltet und die Seite 0 unterscheidet sich deutlich von der anderer Maschinen.

Der Kassettenpuffer befindet sich nicht mehr bei $0330, sodaß die Beispielprogramme in diesem Buch in einen anderen RAM-Bereich verschoben werden müssen (die Adressen $0400

bis $07FF stehen dazu zur Verfügung). Die Umschaltung der Speicherbank ist komplexer als bei anderen Maschinen. Anfänger werden bei dieser Maschine auf noch mehr Dinge achten müssen. Wenn möglich, sollten Sie sich für Ihre ersten Schritte eine einfachere Maschine aussuchen.

Die Implementierung von Programmen großen Ausmaßes erforden ein „Transfer-Sequenz" Programm, um die Speicherbank des Programms mit der der Kernroutinen zu verknüpfen. Gewöhnlich benötigt man einen Urlader (bootstrap program), um alles in Gang zu bringen.

Bei dieser Baureihe ist ein Maschinensprachemonitor eingebaut. Einige neue Befehle stehen zur Verfügung: .V zum Umschalten der Speicherbank, .@ zum Prüfen des Disk Status.

Anhang C
Speicherpläne

Ein Wort über Speicherpläne (memory maps): für die Verwendung, die Sie sich vorstellen, sind diese meist zu umfangreich oder zu knapp bemessen.

Der Anfänger fühlt sich vielleicht von der Detailfülle überschwemmt. Das soll keine Drohung sein. Die Information steht zur Verfügung, wenn man dazu bereit ist. Überfliegen Sie zunächst einiges. Ihre Phantasie wird angeregt. Versuchen Sie, Speicherstellen zu lesen oder zu verändern, um dann zu sehen was passiert.

Der fortgeschrittene Programmierer wünscht vielleicht mehr: ausführliche Information darüber, wie jede Speicherstelle benutzt wird, welche Systemteile diese Stelle nutzen und so weiter. Zeit und beschränkter Platz erlauben diese Details nicht.

Die Pläne sollen einigermaßen vollständig sein. Wer mehr Einzelheiten sucht, findet sie vielleicht zu verschlüsselt; letztlich wird jedoch jede Speicherstelle mit irgendeiner Aktivität in Verbindung zu bringen sein. Verschiedene Maschinen lassen sich durch Überprüfung der entsprechenden Pläne vergleichen. In einigen Fällen kann man Programme je nach Anwendung verändern. Dabei helfen die Pläne, die entsprechenden Speicherstellen in den in Frage kommenden Maschinen zu ermitteln.

Finden Sie einen Hinweis auf eine POKE oder PEEK Stelle - in diesem Buch oder in einem anderen -, überprüfen Sie ihn in diesen Plänen. Sie erweitern damit Ihren Horizont.

„Original-ROM" PET

Die große Jagd auf der Null-Seite

Die meisten Benutzer behelfen sich mit dem oberen Teil des Eingabepuffers ($0040 bis $0059), der unbenutzt bleibt, außer wenn lange Datenzeilen eingegeben werden.

Die meisten Speicherstellen der Null-Seite können in einen anderen Speicherteil in der Weise kopiert werden, daß ihr Originalinhalt später nach der Benutzung zurückgespeichert werden kann. Der Programmierer sollte bei der Veränderung der folgenden Speicherstellen, die innerhalb des Betriebssystems oder des BASIC eine wichtige Rolle spielen, sehr sorgfältig vorgehen: $03, $05, $64 bis $67, $7A bis $87, $89, $A2 bis $A3, $B7, $C2 bis $D9, $E0 bis $E2, $F5.

Speicherplan

Hex	Dezimal	Beschreibung
0000-0002	0-2	USR Sprung
0003	3	Laufende I/O - Prompt Unterdrückung
0005	5	Cursor Kontrollposition
0008-0009	8-9	Integerwert (bei SYS, GOTO usw.)
000A-0059	10-89	Eingabepuffer
005A	90	Suchzeichen
005B	91	Flag für Anführungszeichenmodus
005C	92	Zeiger Eingabepuffer; Anzahl Indizes
005D	93	Flag für vorgegebene Dimensionierung
005E	94	Typ: FF = String; 00 = Numerisch
005F	95	Typ: 80 = Integer 00 = Fließkomma
0060	96	Flag: DATA Abtastung; LIST Anführungszeichenmodus; Speicher
0061	97	Flag: Dimensionierung; FNX
0062	98	0=INPUT; $40=GET; $98=READ
0063	99	Flag TAN Vorzeichen/Vergleichsoperation
0064	100	Eingabeflag (unterdrücke Ausgabe)
0065-0067	101-103	Zeiger für Descriptor-Stapel
0068-0070	104-112	Descriptor-Stapel (temporäre Strings)
0071-0074	113-116	Verschiedene Zeigerbereiche
0075-0078	117-120	Produktbereich bei Multiplikation
007A-007B	122-123	Zeiger: BASIC-Anfang
007C-007D	124-125	Zeiger: Variablenanfang
007E-007F	126-127	Zeiger: Arrayanfang
0080-0081	128-129	Zeiger: Arrayende
0082-0083	130-131	Zeiger: Stringspeicher (bewegt sich abwärts)
0084-0085	132-133	Zeiger benutzter String
0086-0087	134-135	Zeiger: Speichergrenze
0088-0089	136-137	Laufende BASIC-Zeilennummer
008A-008B	138-139	Vorangegangene BASIC-Zeilennummer
008C-008D	140-141	Zeiger: BASIC-Befehl bei CONT
008E-008F	142-143	Laufende DATA-Zeilennummer
0090-0091	144-145	Laufende DATA-Adresse
0092-0093	146-147	Eingabevektor
0094-0095	148-149	Laufender Variablenname
0096-0097	150-151	Laufende Variablenadresse

Hex	Dezimal	Beschreibung
0098-0099	152-153	Variablenzeiger FOR/NEXT
009A-009B	154-155	Zwischenspeicher für Y; Operation; BASIC-Zeiger
009C	156	Akkumulator für Vergleichssymbol
009D-00A2	157-162	Unterschiedlicher Arbeitsbereich, Zeiger usw.
00A3-00A5	163-165	Sprungvektor für Funktionen
00A6-00AF	166-175	Arbeitsbereich bei unterschiedlichen numerischen Anwendungen
00B0	176	Akku #1: Exponent
00B1-00B4	177-180	Akku #1: Mantisse
00B5	181	Akku #1: Vorzeichen
00B6	182	Zähler für Funktionsauswertung
00B7	183	Akku #1 höherwertig (Überlauf)
00B8-00BD	184-189	Akku #2: Exponent usw.
00BE	190	Vorzeichenvergleich, Akku #1 gegen Akku #2
00BF	191	Akku #1 niederwertig (runden)
00C0-00C1	192-193	Zeiger Kassettenpuffer; Länge/Reihe
00C2-00D9	194-217	CHRGET Unterroutine; Holt BASIC-Zeichen
00C9-00CA	201-202	BASIC-Zeiger (innerhalb der Unterroutine)
00DA-00DE	218-222	Anfangswert für Zufallszahl
00E0-00E1	224-225	Zeiger auf Bildschirmzeile
00E2	226	Cursorposition in obiger Zeile
00E3-00E4	227-228	Zeiger für Band, Scrolling
00E5-00E6	229-230	Band-Endadresse/Ende des laufenden Programms
00E7-00E8	231-232	Konstanten für Bandtiming
00E9	233	Bandpuffer-Zeichen
00EA	234	Direkter/programmierter Cursor 0 = direkt
00EB	235	Zeitgeber 1 für Band lesen angeschaltet
00EC	236	EOT von Band empfangen
00ED	237	Fehler Zeichenlesen
00EE	238	Anzahl der Zeichen in Filename
00EF	239	Logische Adresse des laufenden Files
00F0	240	Sekundäradresse des laufenden Files
00F1	241	Einheitennummer des laufenden Files
00F2	242	Zeilenende
00F3-00F4	243-244	Zeiger: Anfang des Bandpuffer
00F5	245	Zeilenposition des Cursor
00F6	246	Letzte Taste/ Prüfsumme Verschiedenes
00F7-00F8	247-248	Band-Anfangsadresse

Hex	Dezimal	Beschreibung
00F9-00FA	249-250	Zeiger Filename
00FB	251	Anzahl ausstehender INSERTs
00FC	252	Schreibe Shift-Wort/Lies Zeichen ein
00FD	253	Noch zu schreibende/lesende Bandblocks
00FE	254	Serieller Wort-Puffer
0100-010A	256-266	Arbeitsbereich STR$
0100-013E	256-318	Zwischenspeicher Bandlesefehler
0100-01FF	256-511	Prozessorstapel
0200-0202	512-513	Momentanwert für TI und TI$
0203	515	Gedrückte Taste: 255 = keine Taste
0204	516	Shift-Taste: 1 wenn gedrückt
0205-0206	517-518	Korrektur für Uhr
0207-0208	519-520	Kassettenstatus, #1 und #2
0209	521	Hauptschalter für PIA: STOP und RVS Flags
020A	522	Zeitgeberkonstante für Band
020B	523	Laden = 0; Verifizieren = 1
020C	524	Statuswort ST
020D	525	Anzahl der Zeichen in Eingabepuffer
020E	526	Flag für invertierte Bildschirmdarstellung
020F-0218	527-536	Tastatur-Eingabepuffer
0219-021A	537-538	IRQ Vektor
021B-021C	539-540	BRK Vektor
021D	541	IEEE Ausgabe: 255 = noch Zeichen auszugeben
021E	542	Zeiger Zeilenende bei Eingabe
0220-0221	544-545	Cursorspeicher (Zeile, Spalte)
0222	546	IEEE Ausgabepuffer
0223	547	Tastenabbild
0224	548	0 = Cursor blinkt
0225	549	Cursor Blinkdauer
0226	550	Zeichen unter Cursor
0227	551	Cursor an/aus
0228	552	EOT von Band empfangen
0229-0241	553-577	Tabelle der Zeilenumschaltpunkte
0242-024B	578-587	Tabelle der logischen Fileadressen
024C-0255	588-597	Tabelle der File-Einheitennummern
0256-025F	598-607	Tabelle der sekundären Fileadressen
0260	608	Eingabe vom Bildschirm/von Tastatur
0261	609	Zwischenspeicher X
0262	610	Anzahl offener Files
0263	611	Eingabeeinheit, normalerweise 0
0264	612	CMD Ausgabeeinheit, normalerweise 3
0265	613	Paritätsbyte für Band

Hex	Dezimal	Beschreibung
0266	614	Flag für Byte empfangen
0268-0269	615-616	Filename Zeiger; Zähler
026C	620	Serieller Bitzähler
026F	623	Zyklenzähler
0270	624	Zähler beim Bandschreiben
0271-0272	625-626	Zeiger für Bandpuffer, #1 und #2
0273	627	Zähler beim Schreiben des Vorspanns; Lesedurchgang 1/2
0274	628	Schreibe neues Byte; Flag für Lesefehler
0275	629	Schreibe Startbit; Bit-Lesefehler
0276-0277	630-631	Zeiger Fehlerspeicher, Durchgang 1/2
0278	632	0 = Abtastung/1-15 = Zählung/ $40 = Laden/$80 = Ende
0279	633	Schreibe Vorspannlänge; Lies Prüfsumme
027A-0339	634-825	Band #1 Eingabepuffer
033A-03F9	826-1017	Band #2 Eingabepuffer
03FA-03FB	1018-1019	Vektor für Monitorerweiterung
0400-7FFF	1024-32767	Zur Verfügung stehendes RAM einschließlich Erweiterung
8000-83E7	32767-33767	Bildschirm RAM-Speicher
C000-E7F8	49152-59384	BASIC ROM; Teil des Kernal ROM
E810-E813	59408-59411	PIA 1 (6520) - Tastaturinterface
E820-E823	59424-59427	PIA 2 (6520) - IEEE Interface
E840-E84F	59456-59471	VIA (6522) - Interface für Verschiedenes, Zeitgeber
F000-FFFF	61440-65535	Kernal ROM Routinen

PIA und VIA Tabellen sind die gleichen, wie sie für den UPGRADE/4.0 gezeigt werden.

UPGRADE und BASIC 4.0 Systeme

Die große Jagd auf der Null-Seite

In diesen Bereichen sind Speicherplätze der Null-Seite schwer zu finden. Die Bereiche $1F bis $27, $4B bis $50 und $54 bis $5D stellen Arbeitsbereiche dar, die für den zeitweisen Gebrauch zur Verfügung stehen. Wenn das Band nicht gelesen oder beschrieben wird, stehen außerdem die Adressen von $B1 bis $C3 zur Verfügung.

Die meisten Speicherstellen der Null-Seite können in einen anderen Speicherteil in der Weise kopiert werden, daß ihr Originalinhalt später nach der Benutzung zurückgespeichert werden kann. Der Programmierer sollte bei der Veränderung der folgenden Speicherstellen, die innerhalb des Betriebssystems oder des BASIC eine wichtige Rolle spielen, sehr sorgfältig vorgehen: $10, $13 bis $15, $28 bis $35, $37, $50 bis $51, $65, $70 bis $87, $8D bis $B0, $C4 bis $FA.

Speicherplan

An den Stellen, an denen das Upgrade ROM sich von dem 4.0 unterscheidet, steht ein Stern (*) und der Wert für 4.0 wird angegeben. Bei der Benutzung ergeben sich zwischen der 40- und 80-Zeilen Maschine einige Unterschiede.

Hex	Dezimal	Beschreibung
0000-0002	0-2	USR Sprung
0003	3	Suchzeichen
0004	4	Flag für Anführungszeichenmodus
0005	5	Zeiger für Eingabepuffer; Anzahl der Indizes
0006	6	Flag bei vorgegebenem DIM
0007	7	Typ:FF = String; 00 = Numerisch
0008	8	Typ:80 = Integer 00 = Fließkomma
0009	9	Flag: DATA; Anführungszeichen bei LIST
000A	10	Flag: Dimensionierung; FNX
000B	11	0=INPUT; $40=GET; $98=READ
000C	12	Flag für TAN Vorzeichen/für Vergleichsoperationen
000D-000F	13-15	*Diskstatus DS$ Descriptor
0010	16	*Laufende I/O Einheit für Unterdrückung des Prompt
0011-0012	17-18	Integerwert (für SYS, GOTO usw.)
0013-0015	19-21	Zeiger für Descriptorstapel
0016-001E	22-30	Descriptorstapel (temporäre Strings)
001F-0022	31-34	Bereich für Anwendungszeiger
0023-0027	35-39	Bereich für Multiplikationsprodukt
0028-0029	40-41	Zeiger: BASIC-Anfang
002A-002B	42-43	Zeiger: Variablenanfang
002C-002D	44-45	Zeiger: Arrayanfang
002E-002F	46-47	Zeiger: Arrayende
0030-0031	48-49	Zeiger: String-Speicherbereich (bewegt sich abwärts)
0032-0033	50-51	Zeiger auf benutzten String
0034-0035	52-53	Zeiger: Speichergrenze
0036-0037	54-55	Laufende BASIC-Zeilennummer
0038-0039	56-57	Vorhergehende BASIC-Zeilennummer
003A-003B	58-59	Zeiger: BASIC-Befehl bei CONT
003C-003D	60-61	Laufende DATA-Zeilennummer
003E-003F	62-63	Laufende DATA-Adresse
0040-0041	64-65	Inputvektor
0042-0043	66-67	Laufender Variablenname
0046-0047	70-71	Variablenzeiger bei FOR/NEXT
0048-0049	72-73	Zwischenspeicher für Y, für Operator; für BASIC-Zeiger

Hex	Dezimal	Beschreibung
004A	74	Akkumulator für Vergleichssymbole
004B-0050	75-80	Verschiedene Arbeitsbereiche, Zeiger usw.
0051-0053	81-83	Sprungvektor bei Funktionen
0054-005D	84-93	Verschiedene Arbeitsbereiche für Funktionsargumente
005E	94	Akku #1: Exponent
005F-0062	95-98	Akku #1: Mantisse
0063	99	Akku #1: Vorzeichen
0064	100	Zähler für Funktionsauswertung
0065	101	Akku #1 höherwertig (Überlauf)
0066-006B	102-107	Akku #2: Exponent usw.
006C	108	Vorzeichenvergleich, Akku #1 gegen Akku #2
006D	106	Akku #1 niederwertig (runden)
006E-006F	110-111	Zeiger Kassettenpuffer; Länge/Reihe
0070-0087	112-135	CHRGET Unterroutine; Holt BASIC-Zeichen
0077-0078	119-120	BASIC-Zeiger (innerhalb der Unterroutine)
0088-008C	136-140	Anfangswert für Zufallszahl
008D-008F	141-143	Momentanwert für TI und TI$
0090-0091	144-145	IRQ Vektor
0092-0093	146-147	BRK Interruptvektor
0094-0095	148-149	NMI Interruptvektor
0096	150	Statuswort ST
0097	151	Gedrückte Taste: 255 = keine Taste
0098	152	Shift-Taste: 1 wenn gedrückt
0099-009A	153-154	Korrektur für Uhr
009B	155	Hauptschalter PIA: STOP und RVS Flags
009C	156	Zeitgeberkonstante für Band
009D	157	Laden = 0; Verifizieren = 1
009E	158	Anzahl Zeichen im Tastaturpuffer
009F	159	Flag für invertierte Bildschirmdarstellung
00A0	160	IEEE Ausgabe: 255 = noch Zeichen auszugeben
00A1	161	Zeiger Zeilenende bei Eingabe
00A3-00A4	163-164	Cursorspeicher (Zeile, Spalte)
00A5	165	IEEE Ausgabepuffer
00A6	166	Tastenabbild
00A7	167	0 = Cursor blinkt
00A8	168	Cursor Blinkdauer
00A9	169	Zeichen unter dem Cursor
00AA	170	Cursor an/aus
00AB	171	EOT von Band empfangen
00AC	172	Eingabe von Bildschirm/Tastatur
00AD	173	Zwischenspeicher X
00AE	174	Anzahl offener Files

Hex	Dezimal	Beschreibung
00AF	175	Eingabeeinheit, normalerweise 0
00B0	176	CMD Ausgabeeinheit, normalerweise 3
00B1	177	Paritätsbyte für Band
00B2	178	Flag für Byte empfangen
00B3	179	Zwischenspeicher für logische Adresse
00B4	180	Bandpuffer Zeichen; MLM Befehl
00B5	181	Zeiger Filename; MLM Flag; Zähler
00B7	183	Serieller Bitzähler
00B9	185	Zyklenzähler
00BA	186	Zähler beim Bandbeschreiben
00BB-00BC	187-188	Zeiger für Bandpuffer, #1 und #2
00BD	189	Zähler beim Schreiben des Vorspanns; Lesedurchgang 1/2
00BE	190	Schreibe neues Byte; Flag für Lesefehler
00BF	191	Schreibe Startbit; Bit-Lesefehler
00C0-00C1	192-193	Zeiger Fehlerspeicher, Durchgang 1/2
00C2	194	0 = Abtastung/1-15 = Zählung/$40 = Laden/$80 = Ende
00C3	195	Schreibe Vorspannlänge; Lies Prüfsumme
00C4-00C5	196-197	Zeiger auf Bildschirmzeile
00C6	198	Cursorposition in obiger Zeile
00C7-00C8	199-200	Zeiger für Band, Scrolling
00C9-00CA	201-202	Band-Endadresse/Ende des laufenden Programms
00CB-00CC	203-204	Konstanten für Bandtiming
00CD	205	0 = Cursor direkt; sonst programmiert
00CE	206	Zeitgeber 1 für Band lesen angeschaltet
00CF	207	EOT von Band empfangen
00D0	208	Fehler Zeichenlesen
00D1	209	Anzahl der Zeichen in Filename
00D2	210	Logische Adresse des laufenden File
00D3	211	Sekundäradresse des laufenden File
00D4	212	Einheitennummer des laufenden File
00D5	213	Rechte Fensterkante oder Zeilenende
00D6-00D7	214-215	Zeiger: Anfang des Bandpuffer
00D8	216	Zeilenposition des Cursor
00D9	217	Letzte Taste/Prüfsumme/Verschiedenes
00DA-00DB	218-219	Zeiger für Filenamen
00DC	220	Anzahl ausstehender INSERTs
00DD	221	Schreibe Shift-Wort/Lies Zeichen ein
00DE	222	Noch zu schreibende/lesende Bandblocks
00DF	223	Serieller Wort-Puffer
00E0-00F8	224-248	(40-Spalten) Tabelle der Zeilenumschaltpunkte
00E0-00E1	224-225	*(80-Spalten) Oberkante, Unterkante des Fensters
00E2	226	*(80-Spalten) linke Fensterbegrenzung

Hex	Dezimal	Beschreibung
00E3	227	*(80-Spalten) Grenze des Tastaturpuffers
00E4	228	*(80-Spalten) Flag Tastenwiederholung
00E5	229	*(80-Spalten) Wiederholungszähler
00E6	230	*(80-Spalten) neuer Tastenmarker
00E7	231	*(80-Spalten) Piep
00E8	232	*(80-Spalten) HOME Zähler
00E9-00EA	233-234	*(80-Spalten) Eingabevektor
00EB-00EC	235-236	*(80-Spalten) Ausgabevektor
00F9-00FA	249-250	Kassettenstatus, #1 und #2
00FB-00FC	251-252	MLM Zeiger/Band-Startadresse
00FD-00FE	253-254	MLM, DOS Zeiger; Verschiedenes
0100-010A	256-266	STR$ Arbeitsbereich, MLM Arbeitsbereich
0100-013E	256-318	Zwischenspeicher Bandlesefehler
0100-01FF	256-511	Prozessorstapel
0200-0250	512-592	MLM Arbeitsbereich; Eingabepuffer
0251-025A	593-602	Tabelle der logischen Fileadressen
025B-0264	603-612	Tabelle der File-Einheitennummern
0265-026E	613-622	Tabelle der sekundären Fileadressen
026F-0278	623-632	Tastatur Eingabepuffer
027A-0339	634-825	Band #1 Eingabepuffer
033A-03F9	826-1017	Band #2 Eingabepuffer
033A-0380	826-896	*DOS Arbeitsbereich
03E9	1001	(Fat 40) Marker neue Taste
03EA	1002	(Fat 40) Tastenwiederholzähler
03EB	1003	(Fat 40) Grenze Tastaturpuffer
03EC	1004	(Fat 40) Piep
03ED	1005	(Fat 40) Zeitgeber 1/10 Sekunden
03EE	1006	(Fat 40) Flag Tastenwiederholung
03EE-03F7	1006-1015	(80-Spalten) Tabelle der Tab-Stops
03EF	1007	(Fat 40) Tab-Arbeitswert
03F0-03F9	1008-1017	(Fat 40) Tab-Stops
03FA-03FB	1018-1019	Vektor für Monitorerweiterung
03FC	1020	*Aufhebung der IEEE Zeitüberschreitung
0400-7FFF	1024-32767	Zur Verfügung stehendes RAM einschließlich Erweiterung
8000-83E7	32767-33767	(40-Spalten) Video RAM
8000-87CF	32768-34767	*(80-Spalten) Video RAM
9000-AFFF	36864-45055	Verfügbarer ROM Erweiterungsbereich
B000-E7FF	45056-59391	BASIC ROM, Teil der Kernroutine (kernal)
E810-E813	59408-59411	PIA 1 - Tastatur I/O
E820-E823	59424-59427	PIA 2 - IEEE-488 I/O
E840-E84F	59456-59471	VIA - I/O Zeitgeber
E880-E881	59520-59521	(80-Spalten und Fat 40) CRT Kontroller
F000-FFFF	61440-65535	Kernal ROM

6520

E810	Diag Sens/ Uncrash	EOI in	Tape Switch Sense #1 #2	Keyboard Row Select		59408
E811	Tape#1 In Latch		(Screen Blank—Orig ROM) EOI Out	DDRA Access	Tape#1 Input L Control	59409
E812	Keyboard Input for selected row					59410
E813	Retrace Latch		Cassette#1 Motor Output	DDRB Access	Retrace Interrupt Control	59411

Abbildung C.1: PIA 1 Tabelle

6520

E820	IEEE-488 Input					59424
E821	$\overline{\text{ATN}}$ Int		$\overline{\text{NDAC}}$ Out	DDRA Access	$\overline{\text{ATN}}$ Int Control	59425
E822	IEEE-488 Output					59426
E823	$\overline{\text{SRQ}}$ Int		$\overline{\text{DAV}}$ Out	DDRB Access	$\overline{\text{SQR}}$ Int Control	59427

Abbildung C.2: PIA 2 Tabelle

6522

E840	$\overline{\text{DAV}}$ In	$\overline{\text{NRFD}}$ In	Retrace In	Tape#2 Motor	Tape Output	$\overline{\text{ATN}}$ Out	NRFD Out	$\overline{\text{NDAC}}$ In	59456
E841	Unused (See E84F)								59457
E842	Data Direction Register B (for E840)								59458
E843	Data Direction Register A (for E84F)								59459
E844	Timer 1								59460
E845									59461
E846	Timer 1 Latch								59462
E847									59463
E848	Timer 2								59464
E849									59465
E84A	Shift Register (unused)								59466
E84B	T1 Control		T2 Cont	Shift Register Control			Latch Controls PB PA		59467
E84C	CB2 (PUP) Control			CB1 Cntl Tape#2	CA2 Control Graphics/Text Mode			CA1 (PUP) Control	59468
E84D	Irq Stats ---- Int Enabl	Timer 1 Int	Timer 2 Int	CB1 Tape#2 Int	CB2 (PUB) Int	SR Unused	CA1 (PUP) Int	CA2 G/T Mode unused. .	59469
E84E									59470
E846F	Parallel User Port Data Register PA								59471

Abbildung C.3: VIA Tabelle

CBM 8032 und FAT-40
6545 CRT Kontroller

Bemerkung:
1. Die Register können nur beschrieben werden.
2. Man vermeide extreme Veränderungen in Register 0. Die Bildröhre könnte beschädigt werden.
3. Register 0 erlaubt zum Anschluß eines externen Monitors eine Justierung der Abtastung
4. Register 12, Bit 4, invertiert das Videosignal.
5. Register 12, Bit 5, schaltet auf alternativen Zeichensatz um. Mit Ausnahme des Super-PET ist der Zeichensatz auf den meisten Maschinen nicht eingebaut.

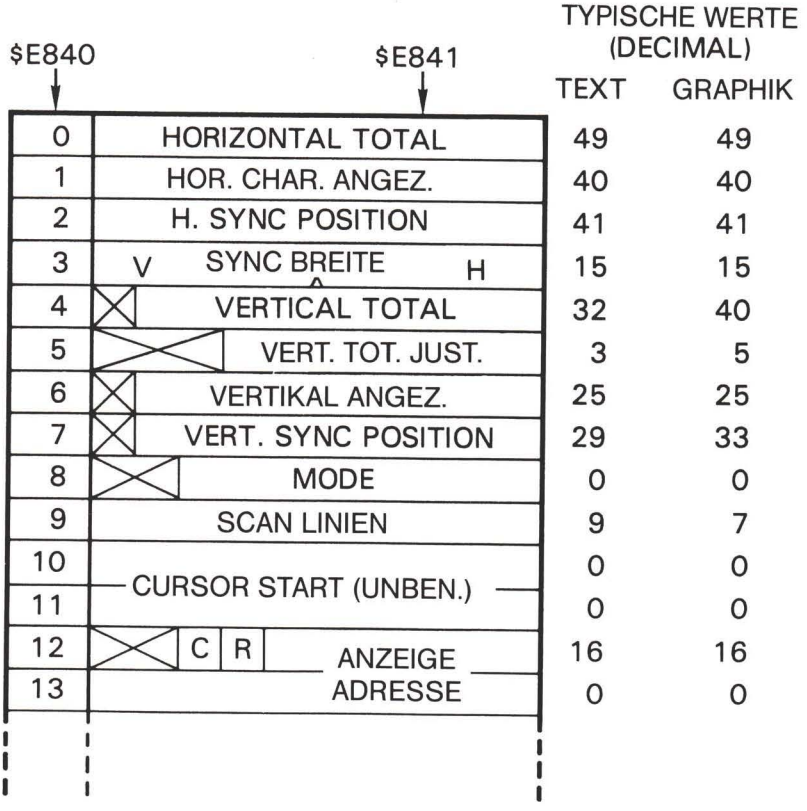


Abbildung C.4

Der 6522 VIA

6522

E840	$\overline{\text{DAV}}$ In	$\overline{\text{NRFD}}$ In	Retrace In	Tape#2 Motor	Tape Output	$\overline{\text{ATN}}$ Out	NRFD Out	$\overline{\text{NDAC}}$ In	59456
E841	Unused (See E84F)								59457
E842	Data Direction Register B (for E840)								59458
E843	Data Direction Register A (for E84F)								59459
E844	Timer 1								59460
E845									59461
E846	Timer 1 Latch								59462
E847									59463
E848	Timer 2								59464
E849									59465
E84A	Shift Register (unused)								59466
E84B	T1 Control		T2 Cont	Shift Register Control			Latch Controls PB	PA	59467
E84C	CB2 (PUP) Control			CB1 Cntl Tape#2	CA2 Control Graphics/Text Mode			CA1 (PUP) Control	59468
E84D	Irq Stats	Timer 1	Timer 2	CB1 Tape#2	CB2 (PUB)	SR	CA1 (PUP)	CA2 G/T Mode	59469
E84E	---- Int Enabl	Int	Int	Int	Int	Unused	Int	unused. .	59470
E846F	Parallel User Port Data Register PA								59471

Abbildung C.5: VIA Tabelle

VIC-20

Die große Jagd auf der Null-Seite

Die Speicherstellen $FC bis $FF stehen zur Verfügung. Speicherstellen $22 bis $2A, $4E bis $53 und $57 bis $60 sind Arbeitsbereiche, die für den vorübergehenden Gebrauch verfügbar sind.

Die meisten Speicherstellen der Null-Seite können in einen anderen Speicherteil in der Weise kopiert werden, daß ihr Originalinhalt später nach der Benutzung zurückgespeichert werden kann. Der Programmierer sollte bei der Veränderung der folgenden Speicherstellen, die innerhalb des Betriebssystems oder des BASIC eine wichtige Rolle spielen, sehr sorgfältig vorgehen: $13, $16 bis $18, $2B bis $38, $3A, $53 bis $54 $68, $73 bis $8A, $90 bis $9A, $A0 bis $A2, $B8 bis $BA, $C5 bis $F4.

Speicherplan

Hex	Dezimal	Beschreibung
0000–0002	0–2	USR Sprung
0003–0004	3–4	Float-fixed Vektor
0005–0006	5–6	Fixed-float Vektor
0007	7	Suchzeichen
0008	8	Flag für Anführungszeichenmodus
0009	9	Speicher TAB Spalte
000A	10	0 = Laden, 1 = Verifizieren
000B	11	Zeiger Eingabepuffer/Zahl der Indizes
000C	12	Flag DIM vorgegeben
000D	13	Typ:FF = String; 00 = Numerisch
000E	14	Typ:80 = Integer; 00 = Fließkomma
000F	15	DATA Abtastung/ Anführungszeichenmodus bei LIST/ Speicherflag
0010	16	Flag Index/FN x
0011	17	0=INPUT; $40=GET; $98=READ
0012	18	Flag TAN Vorzeichen/Vergleichsoperation
0013	19	Flag laufender I/O Prompt
0014–0015	20–21	Integerwert
0016	22	Zeiger: temporärer Stringstapel
0017–0018	23–24	Vektor: letzter temporärer String
0019–0021	25–33	Stapel für temporäre Strings
0022–0025	34–37	Bereich für unterschiedliche Zeiger
0026–002A	38–42	Bereich für Produkt bei Multiplikation
002B–002C	43–44	Zeiger: BASIC-Anfang
002D–002E	45–46	Zeiger: Variablenanfang
002F–0030	47–48	Zeiger: Arrayanfang
0031–0032	49–50	Zeiger: Arrayende
0033–0034	51–52	Zeiger: Stringspeicher (bewegt sich abwärts)
0035–0036	53–54	Zeiger: benutzte String
0037–0038	55–56	Zeiger: Speichergrenze
0039–003A	57–58	Laufende BASIC-Zeilennummer
003B–003C	59–60	Vorangegangene BASIC-Zeilennummer
003D–003E	61–62	Zeiger: BASIC-Befehl für CONT
003F–0040	63–64	Laufende DATA-Zeilennummer
0041–0042	65–66	Laufende DATA-Adresse
0043–0044	67–68	Eingabevektor
0045–0046	69–70	Laufender Variablenname
0047–0048	71–72	Laufende Variablenadresse
0049–004A	73–74	Variablenzeiger bei FOR/NEXT
004B–004C	75–76	Zwischenspeicher für Y; für Operator; für BASIC-Zeiger

Hex	Dezimal	Beschreibung
004D	77	Akkumulator für Vergleichssymbol
004E-0053	78-83	Arbeitsbereich für unterschiedliche Anwendungen, Zeiger usw.
0054-0056	84-86	Sprungvektor für Funktionen
0057-0060	87-96	Arbeitsbereich für unterschiedliche numerische Anwendungen
0061	97	Akku #1: Exponent
0062-0065	98-101	Akku #1: Mantisse
0066	102	Akku #1: Vorzeichen
0067	103	Zeiger für Funktionsauswertung
0068	104	Akku #1 höherwertig (Überlauf)
0069-006E	105-110	Akku #2: Exponent usw.
006F	111	Vorzeichenvergleich, Akku #1 gegen Akku #2
0070	112	Akku #1 niederwertig (runden)
0071-0072	113-114	Zeiger Kassettenpuffer Länge/Reihe
0073-008A	115-138	CHRGET Unterroutine; hole BASIC-Zeichen
007A-007B	122-123	BASIC-Zeiger (innerhalb der Unterroutine)
008B-008F	139-143	Anfangswert für RND
0090	144	Statuswort ST
0091	145	Hauptschalter PIA: STOP und RVS Flags
0092	146	Zeitgeberkonstante für Band
0093	147	Laden = 0; Verifizieren = 1
0094	148	Serielle Ausgabe: Flag für hinausgeschobenes Zeichen
0095	149	Seriell hinausgeschobenes Zeichen
0096	150	EOT von Band empfangen
0097	151	Zwischenspeicher für Register
0098	152	Anzahl offener Files
0099	153	Eingabeeinheit, normalerweise 0
009A	154	CMD Ausgabeeinheit, normalerweise 3
009B	155	Band - Zeichenparität
009C	156	Flag für Byte-empfangen
009D	157	Ausgabekontrolle Direkt = $80/RUN=0
009E	158	Fehlerspeicher Band Durchgang 1 / Zeichenpuffer
009F	159	Speicherfehler Band Durchgang 2 korrigiert
00A0-00A2	160-162	Momentanwert Uhr HML
00A3	163	Serieller Bitzähler/ EOI Flag
00A4	164	Zyklenzähler
00A5	165	Bandschreiben: Zähler / Bitzahl
00A5	166	Zeiger Bandpuffer
00A7	167	Vorspannzähler Bandschreiben/Lesedurchgang Biteingabe

Hex	Dezimal	Beschreibung
00A8	168	Neues Byte Bandschreiben / Lesefehler / Biteingabezähler
00A9	169	Schreibe Startbit / Lesen Bitfehler / Startbit
00AA	170	Bandabtastung; Zähler; Laden; Ende / Byte-Zusammenstellung
00AB	171	Schreiben Vorspannlänge / Lesen Prüfsumme / Parität
00AC-00AD	172-173	Zeiger: Bandpuffer, Scrolling
00AE-00AF	174-175	Band-Endadresse / Programmende
00B0-00B1	176-177	Konstanten für Bandtiming
00B2-00B3	178-179	Zeiger: Anfang Bandpuffer
00B4	180	1 = Zeitgeber Band angestellt; Bitzahl
00B5	181	Band EOT / RS232 nächstes zu sendendes Bit
00B6	182	Lesen Zeichenfehler / Puffer für auszusendendes Byte
00B7	183	Anzahl Zeichen in Filename
00B8	184	Laufender logischer File
00B9	185	Laufende Sekundäradresse
00BA	186	Laufende Einheit
00BB-00BC	187-188	Zeiger auf Filename
00BD	189	Schreiben Shift-Wort / Lesen Eingabezeichen
00BE	190	Anzahl der noch zu schreibenden / lesenden Blocks
00BF	191	Serieller Wortpuffer
00C0	192	Bandmotor-Sperre
00C1-00C2	193-194	I/O Startadresse
00C3-00C4	195-196	Anfangszeiger Kernal
00C5	197	Letzte gedrückte Taste
00C6	198	Anzahl der Zeichen in Eingabepuffer
00C7	199	Flag für invertierte Bildschirmdarstellung
00C8	200	Zeiger Zeilenende bei Eingabe
00C9-00CA	201-202	Speicher Cursor Eingabe (Zeile, Spalte)
00CB	203	Gedrückte Taste: 64 wenn keine Taste
00CC	204	0 = Cursor blinkt
00CD	205	Cursor Zeitgeber
00CE	206	Zeichen unter Cursor
00CF	207	Cursor an/aus
00D0	208	Eingabe von Bildschirm / Tastatur
00D1-00D2	209-210	Zeiger auf Bildschirmzeile
00D3	211	Curserposition in obiger Zeile
00D4	212	0 = direkter Cursor; andernfalls programmiert
00D5	213	Länge der laufenden Bildschirmzeile

Hex	Dezimal	Beschreibung
00D6	214	Spalte des Cursor
00D7	215	Letzte Taste / Prüfsumme / Puffer
00D8	216	Anzahl ausstehender INSERTs
00D9-00F0	217-240	Bildschirm Verbindungstabelle
00F1	241	Leerwert Bildschirmverbindung
00F2	242	Bildschirm Zeilenmarker
00F3-00F4	243-244	Zeiger Bildschirmfarbe
00F5-00F6	245-246	Zeiger Tastatur
00F7-00F8	247-248	RS232 Empfangszeiger
00F9-00FA	249-250	RS232 Sendezeiger
00FF-010A	256-266	Arbeitsbereich Fließkomma nach ASCII
0100-103E	256-318	Zwischenspeicher Bandfehler
0100-01FF	256-511	Prozessor Stapelbereich
0200-0258	512-600	BASIC-Eingabepuffer
0259-0262	601-610	Tabelle logischer Files
0263-026C	611-620	Tabelle Einheitennummern
026D-0276	621-630	Tabelle Sekundäradressen
0277-0280	631-640	Tastaturpuffer
0281-0282	641-642	Anfang des BASIC-Speicher
0283-0284	643-644	Ende des BASIC-Speicher
0285	645	Flag Zeitüberschreitung serieller Bus
0286	646	Laufender Farbcode
0287	647	Farbe unter Cursor
0288	648	Bildschirm Speicherseite
0289	649	Maximalgröße des Tastaturpuffer
028A	650	Tastenwiederholung
028B	651	Zähler Wiederholgeschwindigkeit
028C	652	Zähler Wiederholverzögerung
028D	653	Tastatur Shift / Kontrollflag
028E	654	Letztes Shiftmuster
028F-0290	655-656	Anfangszeiger Tastaturtabelle
0291	657	Tastenmodus (Kattacanna)
0292	658	0 = Scroll möglich
0293	659	RS232 Chipkontrolle
0294	660	RS232 Chipbefehl
0295-0296	661-662	Bit-Timing
0297	663	RS232 Status
0298	664	Anzahl zu sendender Bits
0299-029A	665-666	RS232 Geschwindigkeit / Code
029B	667	RS232 Empfangszeiger
029C	668	RS232 Eingabezeiger
029D	669	RS232 Sendezeiger
029E	670	RS232 Ausgabezeiger

Hex	Dezimal	Beschreibung
029F-02A0	671-672	IRQ Speicher bei Band I/O
0300-0301	768-769	Verbindung Fehlernachricht
0302-0303	770-771	Verbindung BASIC-Warmstart
0304-0305	772-773	Verbindung BASIC-Befehlswerte decodiert
0306-0307	774-775	Verbindung Ausdruck Befehlswerte
0308-0309	776-777	Verbindung Anfang des neuen BASIC-Code
030A-030B	778-779	Verbindung Holen Zahlenelement
030C	780	Speicher A-Register
030D	781	Speicher X-Register
030E	782	Speicher Y-Register
030F	783	Speicher SYS Statusregister
0314-0315	788-789	IRQ Vektor (EABF)
0316-0317	790-791	Break Interrupt Vektor (FED2)
0318-0319	792-793	NMI Interrupt Vektor (FEAD)
031A-031B	794-795	OPEN Vektor (F40A)
031C-031D	796-797	CLOSE Vektor (F34A)
031E-031F	798-799	Set-input Vektor (F2C7)
0320-0321	800-801	Set-output Vektor (F309)
0322-0323	802-803	Speichere I/O Vektor (F3F3)
0324-0325	804-805	INPUT Vektor (F20E)
0326-0327	806-807	Ausgabe Vektor (F27A)
0328-0329	808-809	Test-STOP Vektor (F770)
032A-032B	810-811	GET Vektor (F1F5)
032C-032D	812-813	Vergiß I/O Vektor (F3EF)
032E-032F	814-815	USR Vektor (FED2)
0330-0331	816-817	Verbindung LOAD
0332-0333	818-819	Verbindung SAVE
033C-03FB	828-1019	Kassettenpuffer
0400-0FFF	1024-4095	3K RAM Erweiterungsbereich
1000-1FFF	4096-8191	Normaler BASIC-Speicher
2000-7FFF	8192-32767	Speichererweiterungsbereich
8000-8FFF	32768-36863	Zeichen Bitmuster (ROM)
9000-900F	36864-36879	Video interface chip (6560)
9110-912F	37136-37151	VIA (6522) Interface - NMI
9120-912F	37152-37167	VIA (6522) Interface IRQ
9400-95FF	37888-38399	Bereich alternative Farbnybble
9600-97FF	38400-38911	Bereich Haupt-Farbnybble
A000-BFFF	40960-49151	Bereich für Einsteck-ROM
C000-FFFF	49152-65535	ROM: BASIC und Betriebssystem

VIC 6560 Baustein

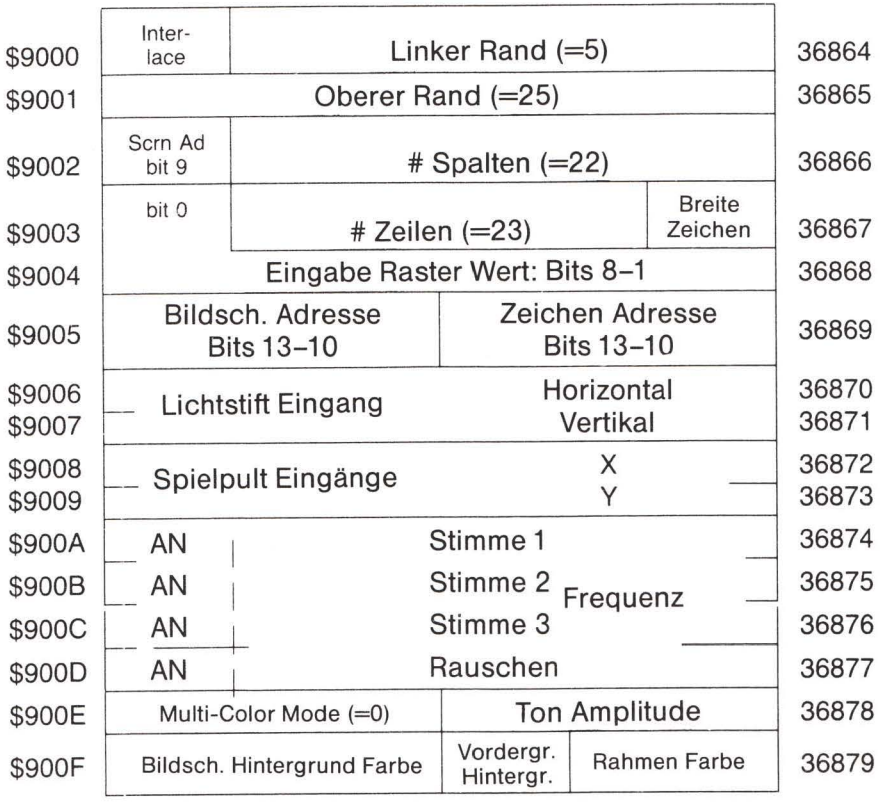


Abbildung C.6

VIC 6522 Benutzung

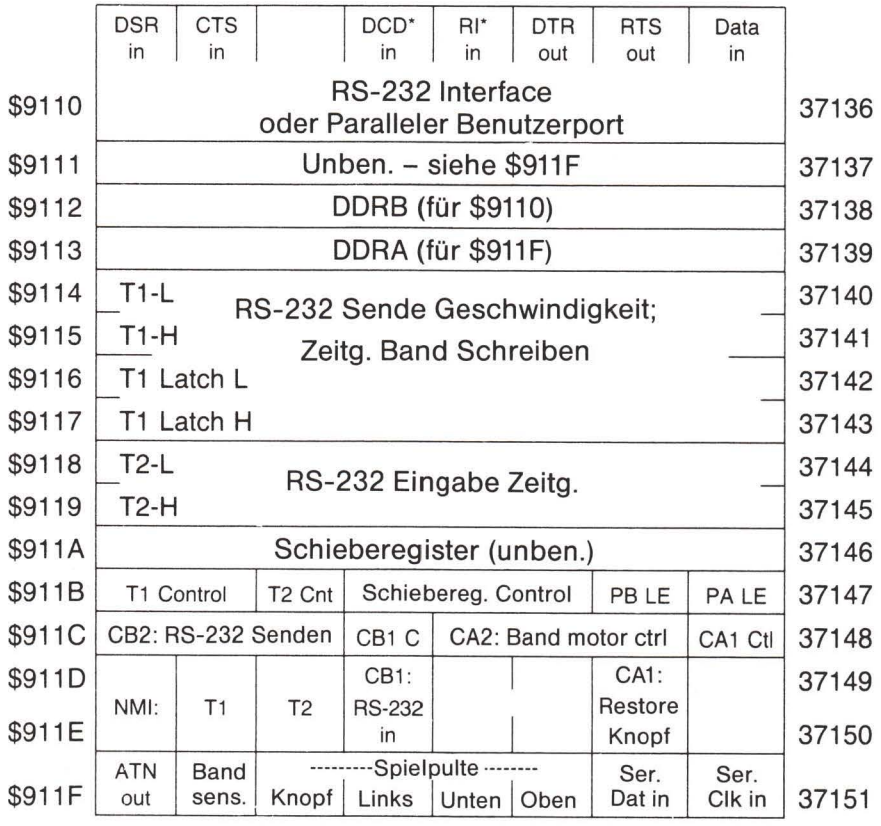


Abbildung C.7

VIC 6522 Benutzung

$9120	Spielpult Rechts			Band Out			37152
	Auswahl Tastatur Reihe						
$9121	Eingabe Tastatur Spalte						37153
$9122	DDRB (für $9120)						37154
$9123	DDRA (für $9121)						37155
$9124	T1-L	Kassette Band Lesen; Tastatur & Clock Interrupt Zeitg.					37156
$9125	T1-H						37157
$9126	T1-L Latch						37158
$9127	T1-H Latch						37159
$9128	T2-L	Ser. Bus Zeitg.					37160
$9129	T2-H	Band R/W Zeitg.					37161
$912A	Schiebereg. (unbenutzt)*						37162
$912B	T1 Control		T2 Ctrl	Schiebereg Contrl		PB LE	PA LE	37163
$912C	Ser. Bus Dat.			CB1 Contl	Ser. Clock Ltg. out		CA1 Contl	37164
$912D				CB1:*		CA1:		37165
$912E	IRQ:	T1	T2	SRQ in		Band in		37166
$912F	*Unben.: siehe $9121							37167

Abbildung C.8

Commodore 64:

Die große Jagd auf der Null-Seite

Die Speicherstellen $FC bis $FF stehen zur Verfügung. Speicherstellen $22 bis $2A, $4E bis $53 und $57 bis $60 sind Arbeitsbereiche, die für den vorübergehenden Gebrauch verfügbar sind.

Die meisten Speicherstellen der Null-Seite können in einen anderen Speicherteil in der Weise kopiert werden, daß ihr Originalinhalt später nach der Benutzung zurückgespeichert werden kann. Der Programmierer sollte bei der Veränderung der folgenden Speicherstellen, die innerhalb des Betriebssystems oder des BASIC eine wichtige Rolle spielen, sehr sorgfältig vorgehen: $13, $16 bis $18, $2B bis $38, $3A, $53 bis $54, $68, $73 bis $8A, $90 bis $9A, $A0 bis $A2, $B8 bis $BA, $C5 bis $F4.

Speicherplan

Hex	Dezimal	Beschreibung
0000	0	Chip Richtungsregister
0001	1	Chip I/O; Speicher- und Bandkontrolle
0003-0004	3-4	Fließkomma-Festkomma Vektor
0005-0006	5-6	Festkomma-Fließkomma Vektor
0007	7	Suchzeichen
0008	8	Flag für Anführungszeichenmodus
0009	9	Speicher TAB Spalte
000A	10	0 = Laden, 1 = Verifizieren
000B	11	Zeiger Eingabepuffer/Zahl der Indizes
000C	12	Flag DIM vorgegeben
000D	13	Typ:FF = String; 00 = Numerisch
000E	14	Typ:80 = Integer; 00 = Fließkomma
000F	15	DATA Abtastung/ Anführungszeichenmodus bei LIST/ Speicherflag
0010	16	Flag Index/FN x
0011	17	0=INPUT; $40=GET; $98=READ
0012	18	Flag TAN Vorzeichen/Vergleichsoperation
0013	19	Flag laufender I/O Prompt
0014-0015	20-21	Integerwert
0016	22	Zeiger: temporärer Stringstapel
0017-0018	23-24	Vektor: letzter temporärer String
0019-0021	25-33	Stapel für temporäre Strings
0022-0025	34-37	Bereich für unterschiedliche Zeiger
0026-002A	38-42	Bereich für Produkt bei Multiplikation
002B-002C	43-44	Zeiger: BASIC-Anfang
002D-002E	45-46	Zeiger: Variablenanfang
002F-0030	47-48	Zeiger: Arrayanfang

Hex	Dezimal	Beschreibung
0031-0032	49-50	Zeiger: Arrayende
0033-0034	51-52	Zeiger: Stringspeicher (bewegt sich abwärts)
0035-0036	53-54	Zeiger benutzte String
0037-0038	55-56	Zeiger: Speichergrenze
0039-003A	57-58	Laufende BASIC-Zeilennummer
003B-003C	59-60	Vorangegangene BASIC-Zeilennummer
003D-003E	61-62	Zeiger: BASIC-Befehl für CONT
003F-0040	63-64	Laufende DATA-Zeilennummer
0041-0042	65-66	Laufende DATA-Adresse
0043-0044	67-68	Eingabevektor
0045-0046	69-70	Laufender Variablenname
0047-0048	71-72	Laufende Variablenadresse
0049-004A	73-74	Variablenzeiger bei FOR/NEXT
004B-004C	75-76	Zwischenspeicher für Y; für Operator; für BASIC-Zeiger
004D	77	Akkumulator für Vergleichssymbol
004E-0053	78-83	Arbeitsbereich für unterschiedliche Anwendungen, Zeiger usw.
0054-0056	84-86	Sprungvektor für Funktionen
0057-0060	87-96	Arbeitsbereich für unterschiedliche numerische Anwendungen
0061	97	Akku #1: Exponent
0062-0065	98-101	Akku #1: Mantisse
0066	102	Akku #1: Vorzeichen
0067	103	Zeiger für Funktionsauswertung
0068	104	Akku #1 höherwertig (Überlauf)
0069-006E	105-110	Akku #2: Exponent usw.
006F	111	Vorzeichenvergleich, Akku #1 gegen Akku #2
0070	112	Akku #1 niederwertig (runden)
0071-0072	113-114	Zeiger Kassettenpuffer; Länge/Reihe
0073-008A	115-138	CHRGET Unterroutine; hole BASIC-Zeichen
007A-007B	122-123	BASIC-Zeiger (innerhalb der Unterroutine)
008B-008F	139-143	Anfangswert für RND
0090	144	Statuswort ST
0091	145	Hauptschalter PIA: STOP und RVS Flags
0092	146	Zeitgeberkonstante für Band
0093	147	Laden = 0; Verifizieren = 1
0094	148	Serielle Ausgabe: Flag für hinausgeschobenes Zeichen
0095	149	Seriell hinausgeschobenes Zeichen
0096	150	EOT von Band empfangen
0097	151	Zwischenspeicher für Register
0098	152	Anzahl offener Files

Hex	Dezimal	Beschreibung
0099	153	Eingabeeinheit, normalerweise 0
009A	154	CMD Ausgabeeinheit, normalerweise 3
009B	155	Band - Zeichenparität
009C	156	Flag für Byte-empfangen
009D	157	Ausgabekontrolle Direkt = $80/RUN=0
009E	158	Fehlerspeicher Band Durchgang 1/Zeichenpuffer
009F	159	Fehlerspeicher Band Durchgang 2 korrigiert
00A0-00A2	160-162	Momentanwert Uhr HML
00A3	163	Serieller Bitzähler/ EOI Flag
00A4	164	Zyklenzähler
00A5	165	Bandschreiben Zähler / Bitzahl
00A5	166	Zeiger Bandpuffer
00A7	167	Vorspannzähler Bandschreiben/Lesedurchgang Biteingabe
00A8	168	Neues Byte Bandschreiben / Lesefehler / Biteingabezähler
00A9	169	Schreibe Startbit / Lesen Bitfehler / Startbit
00AA	170	Bandabtastung; Zähler; Laden; Ende / Byte-Zusammenstellung
00AB	171	Schreiben Vorspannlänge / Lesen Prüfsumme / Parität
00AC-00AD	172-173	Zeiger: Bandpuffer, Scrolling
00AE-00AF	174-175	Band-Endadresse / Programmende
00B0-00B1	176-177	Konstanten für Bandtiming
00B2-00B3	178-179	Zeiger: Anfang Bandpuffer
00B4	180	1 = Zeitgeber Band angestellt; Bitzahl
00B5	181	Band EOT / RS232 nächstes zu sendendes Bit
00B6	182	Lesen Zeichenfehler / Puffer für auszusendendes Byte
00B7	183	Anzahl Zeichen in Filename
00B8	184	Laufender logischer File
00B9	185	Laufende Sekundäradresse
00BA	186	Laufende Einheit
00BB-00BC	187-188	Zeiger auf Filename
00BD	189	Schreiben Shift-Wort / Lesen Eingabezeichen
00BE	190	Anzahl der noch zu schreibenden / lesenden Blocks
00BF	191	Serieller Wortpuffer
00C0	192	Bandmotor-Sperre
00C1-00C2	193-194	I/O Startadresse
00C3-00C4	195-196	Anfangszeiger Kernal
00C5	197	Letzte gedrückte Taste
00C6	198	Anzahl der Zeichen in Eingabepuffer
00C7	199	Flag für invertierte Bildschirmdarstellung

Hex	Dezimal	Beschreibung
00C8	200	Zeiger Zeilenende bei Eingabe
00C9-00CA	201-202	Speicher Cursor Eingabe (Zeile, Spalte)
00CB	203	Gedrückte Taste: 64 wenn keine Taste
00CC	204	0 = Cursor blinkt
00CD	205	Cursor Zeitgeber
00CE	206	Zeichen unter Cursor
00CF	207	Cursor an/aus
00D0	208	Eingabe von Bildschirm / Tastatur
00D1-00D2	209-210	Zeiger auf Bildschirmzeile
00D3	211	Cursorposition in obiger Zeile
00D4	212	0 = direkter Cursor; andernfalls programmiert
00D5	213	Länge der laufenden Bildschirmzeile
00D6	214	Spalte des Cursor
00D7	215	Letzte Taste / Prüfsumme / Puffer
00D8	216	Anzahl ausstehender INSERTs
00D9-00F2	217-242	Bildschirm Verbindungstabelle
00F3-00F4	243-244	Zeiger Bildschirmfarbe
00F5-00F6	245-246	Zeiger Tastatur
00F7-00F8	247-248	RS232 Empfangszeiger
00F9-00FA	249-250	RS232 Sendezeiger
00FF-010A	256-266	Arbeitsbereich Fließkomma nach ASCII
0100-103E	256-318	Zwischenspeicher Bandfehler
0100-01FF	256-511	Prozessor Stapelbereich
0200-0258	512-600	BASIC-Eingabepuffer
0259-0262	601-610	Tabelle logischer Files
0263-026C	611-620	Tabelle Einheitennummern
026D-0276	621-630	Tabelle Sekundäradressen
0277-0280	631-640	Tastaturpuffer
0281-0282	641-642	Anfang des BASIC-Speicher
0283-0284	643-644	Ende des BASIC-Speicher
0285	645	Flag Zeitüberschreitung serieller Bus
0286	646	Laufender Farbcode
0287	647	Farbe unter Cursor
0288	648	Bildschirm Speicherseite
0289	649	Maximalgröße des Tastaturpuffer
028A	650	Tastenwiederholung
028B	651	Zähler Wiederholgeschwindigkeit
028C	652	Zähler Wiederholverzögerung
028D	653	Tastatur Shift / Kontrollflag
028E	654	Letztes Shiftmuster
028F-0290	655-656	Anfangszeiger Tastaturtabelle
0291	657	Tastatur Shiftmodus
0292	658	0 = Scroll möglich

Hex	Dezimal	Beschreibung
0293	659	RS232 Chipkontrolle
0294	660	RS232 Chipbefehl
0295-0296	661-662	Bit-Timing
0297	663	RS232 Status
0298	664	Anzahl zu sendender Bits
0299-029A	665-666	RS232 Geschwindigkeit / Code
029B	667	RS232 Empfangszeiger
029C	668	RS232 Eingabezeiger
029D	669	RS232 Sendezeiger
029E	670	RS232 Ausgabezeiger
029F-02A0	671-672	IRQ Speicher bei Band I/O
02A1	673	CIA 2 (NMI) Interruptkontrolle
02A2	674	CIA 1 Zeitgeber A Kontrollspeicher
02A3	675	CIA 1 Interruptspeicher
02A4	676	CIA 1 Flag Zeitgeber A angeschaltet
02A5	677	Bildschirm Zeilenmarker
02C0-02FE	704-766	(Sprite 7)
0300-0301	768-769	Verbindung Fehlernachricht
0302-0303	770-771	Verbindung BASIC-Warmstart
0304-0305	772-773	Verbindung BASIC-Befehlswerte decodiert
0306-0307	774-775	Verbindung Ausdruck Befehlswerte
0308-0309	776-777	Verbindung Anfang des neuen BASIC-Code
030A-030B	778-779	Verbindung holen; Zahlenelement
030C	780	Speicher A-Register
030D	781	Speicher X-Register
030E	782	Speicher Y-Register
030F	783	Speicher SYS Statusregister
0310-0312	784-785	USR Sprung (B248)
0314-0315	788-789	IRQ Vektor (EA31)
0316-0317	790-791	Break Interrupt Vektor (FE66)
0318-0319	792-793	NMI Interrupt Vektor (FE47)
031A-031B	794-795	OPEN Vektor (F34A)
031C-031D	796-797	CLOSE Vektor (F291)
031E-031F	798-799	Set-input Vektor (F20E)
0320-0321	800-801	Set-output Vektor (F250)
0322-0323	802-803	Speichere I/O Vektor (F333)
0324-0325	804-805	INPUT Vektor (F157)
0326-0327	806-807	Ausgabe Vektor (F1CA)
0328-0329	808-809	Test-STOP Vektor (F6ED)
032A-032B	810-811	GET Vektor (F13E)
032C-032D	812-813	Vergiß I/O Vektor (F32F)
032E-032F	814-815	USR Vektor (FE66)
0330-0331	816-817	Verbindung LOAD (F4A5)

Hex	Dezimal	Beschreibung
0332-0333	818-819	Verbindung SAVE (F5ED)
033C-03FB	828-1019	Kassettenpuffer
0340-037E	832-894	(Sprite 13)
0380-03BE	896-958	(Sprite 14)
03C0-03FE	960-1022	(Sprite 15)
0400-07FF	1024-2047	Bildschirmspeicher
0800-9FFF	2048-40959	BASIC ROM
8000-9FFF	32768-40959	Alternativ: ROM Einsteckbereich
A000-BFFF	40960-49151	ROM: BASIC
A000-BFFF	40960-49151	Alternativ: RAM
C000-CFFF	49152-53247	RAM Speicher, einschließlich Alternative
D000-D02E	53248-53294	Video Baustein (6566)
D400-D41C	54272-54300	Ton Baustein (6581 SID)
D800-DBFF	55296-56319	Speicher Farbnybble
DC00-DC0F	56320-56335	Interface Baustein 1, IRQ (6526 CIA)
DD00-DD0F	56576-56591	Interface Baustein 2, NMI (6526 CIA)
D000-DFFF	53248-53294	Alternativ: Zeichensatz
E000-FFFF	57344-65535	ROM: Betriebssystem
E000-FFFF	57344-65535	Alternativ: RAM

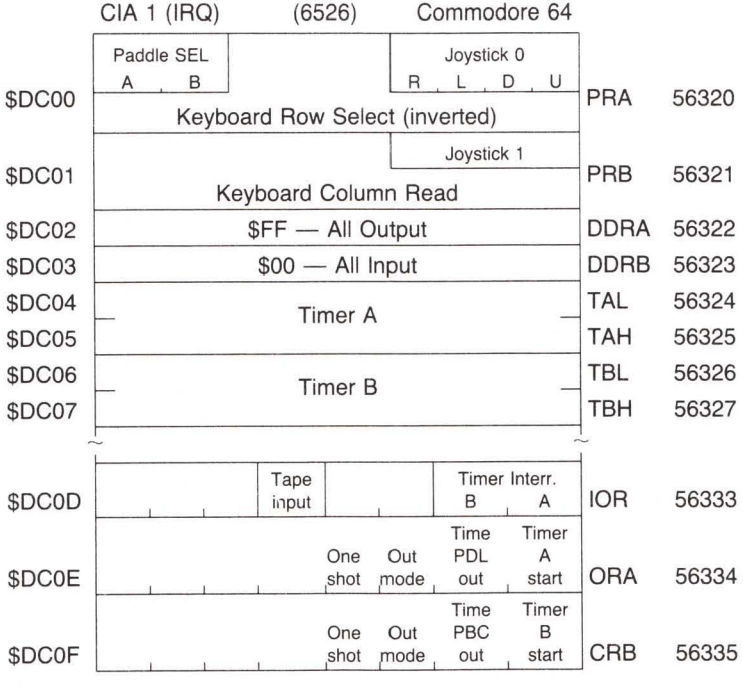


Abbildung C.9

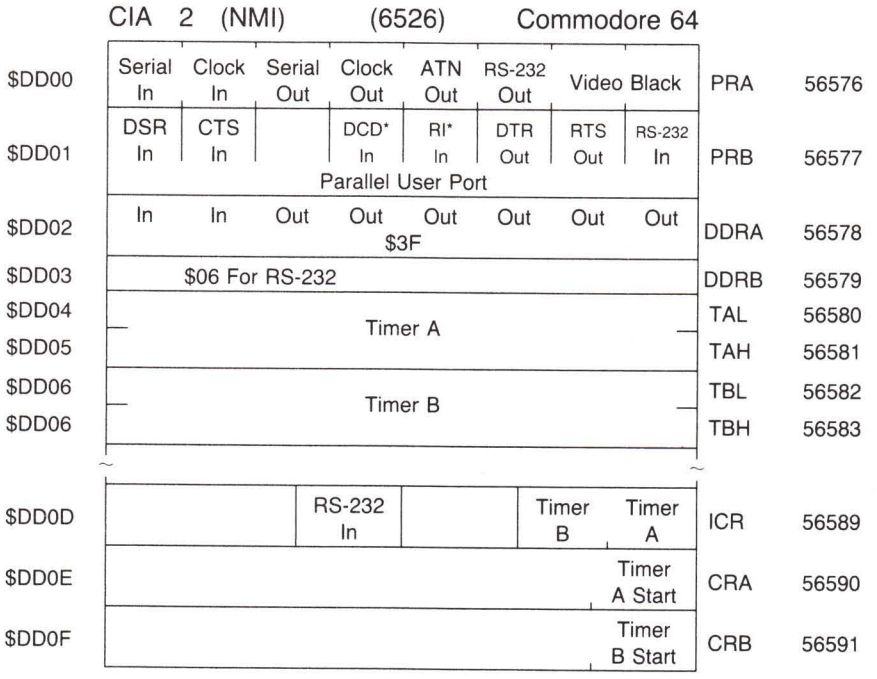


Abbildung C.10

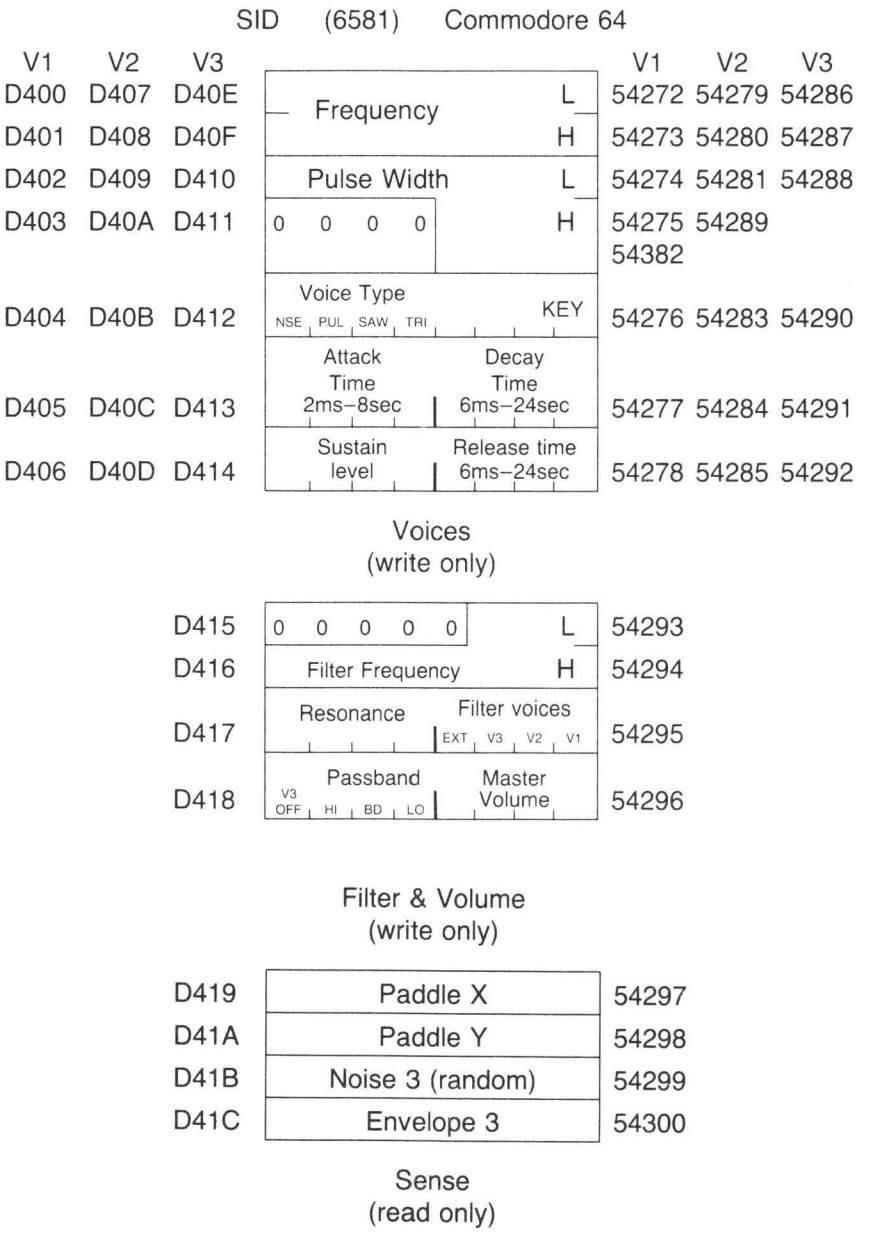
C64 Speicherplan
6566 Video – Sprite Register

Sprite 0	Sprite 7		Sprite 0	Sprite 7
D000	D00E	Position X	53248	53262
D001	D00F	Y	53249	53263
D027	D02E	Farbe	53287	53294

Sprite Bit Positionen

	7 6 5 4 3 2 1 0	
D010	X-Position	53264
D015	Sprite erl.	53269
D017	Y-expand.	53271
D01B	Hintergrund Priorität	53275
D01C	Multicolor	53276
D01D	X-expand.	53277
D01E	Interrupt: Sprite Kollision	53278
D01F	Interrupt: Sprite/Hintergr. Koll.	53279

Abbildung C.11



Spezielle Stimmeneigenschaften (TEST, RING MOD, SYNC) sind in obigem Diagramm ausgelassen.

Abbildung C.12

Commodore PLUS/4 „TED" Baustein – Vorläufige Angaben

Zur Zeit der Publikation standen weder der Commodore 264 (auch PLUS/4 genannt) noch eine verwandte Maschine, der Commodore 16, käuflich zur Verfügung. Vor der Verkaufsfreigabe könnten sich Einzelheiten des Entwurfs geändert haben.

Bei den Einheiten des Prototyp gleicht der größte Teil der Null-Seite derjenigen des VIC und Commodore 64. Besonders die BASIC Zeiger (SOB, SOV usw.) sind gleich.

Vorläufiger Speicherplan

Der größte Teil der Null-Seite gleicht der des Commodore 64. Einige Unterschiede und weitere Informationen sind aufgelistet.

Hex	Dezimal	Beschreibung
0073-008A	115-138	(CHRGET nicht vorhanden)
0097	151	Anzahl der offenen Files
0098	152	Eingabeeinheit, normalerweise 0
0099	153	CMD Ausgabeeinheit, normalerweise 3
00AC	172	Laufender logischer File
00AD	173	Laufende Sekundäradresse
00AE	174	Laufende Einheit
00AF-00B0	175-176	Zeiger auf Filename
00C8-00C9	200-201	Zeiger auf Bildschirmzeile
00CA	202	Cursorposition in obiger Zeile
00CD	205	Zeile des Cursor
00EF	239	Anzahl der Zeichen in Tastaturpuffer
0314-0315	788-789	IRQ Vektor (CE0E)
0316-0317	790-791	Break Interrupt Vektor (F44B)
0318-0319	792-793	OPEN Vektor (EF53)

(Die meisten anderen Vektoren gleichen dem C64, liegen aber 2 Stellen tiefer)

0500-0502	1280-1282	USR Sprung
0509-0512	1289-1298	Tabelle logischer Files
0513-051C	1299-1308	Tabelle der Einheitennummern
051D-0526	1309-1318	Tabelle der Sekundäradressen
0527-0530	1319-1328	Tastaturpuffer
0800-0BE7	2048-3047	Farbspeicher
0C00-0FE7	3072-4071	Bildschirmspeicher
1000-FFFF	4096-65535	BASIC RAM-Speicher
8000-FFFF	32768-65535	ROM: BASIC
FF00-FF3F	65280-65343	TED I/O Kontrollchip

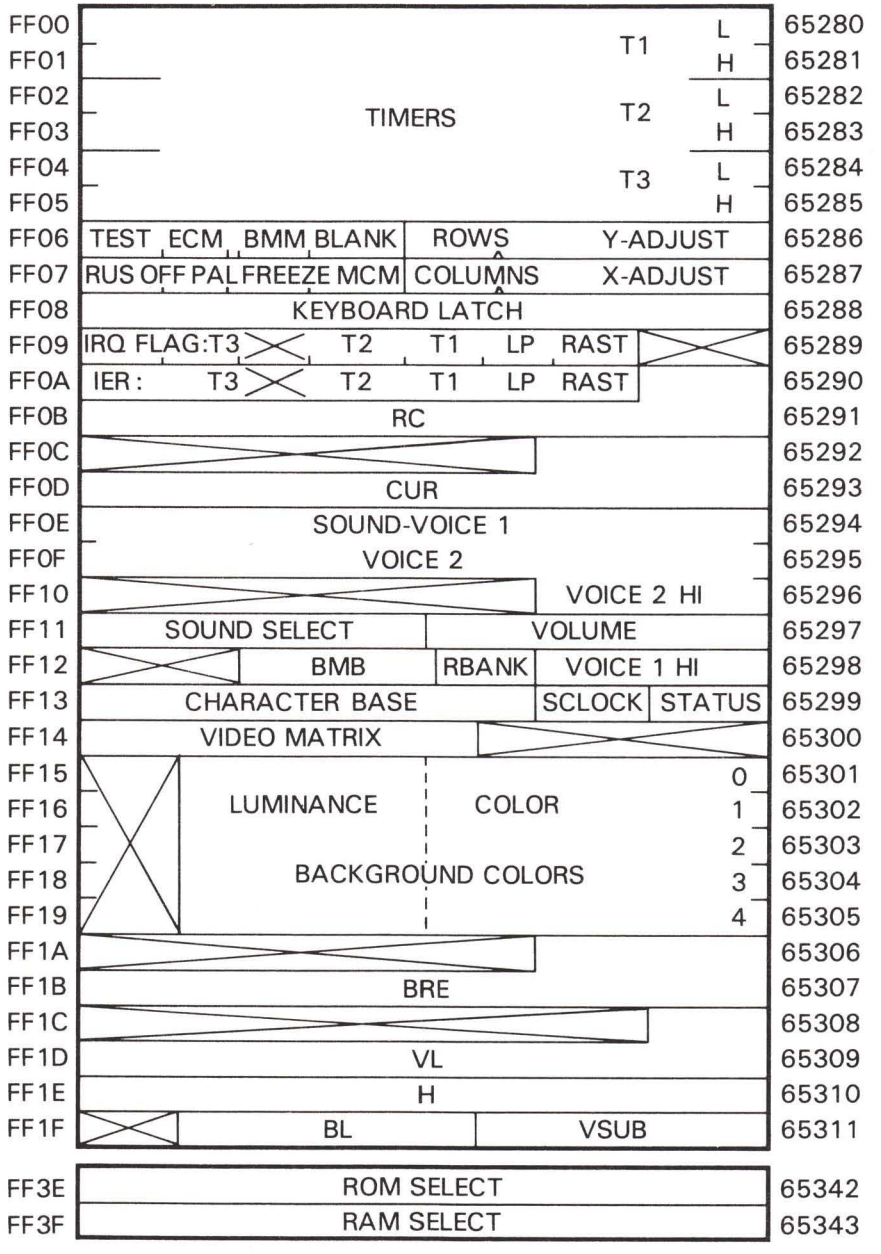


Abbildung C.13

B-Serie (B-128 CBM-256 usw.)

Die große Jagd auf der Null-Seite

Die Null-Seite hat bei der B-Serie eine unterschiedliche Bedeutung. Es gibt mehrere Null-Seiten. Üblicherweise möchte man Werte der Bank 15 benutzen (der ROM Bank, in der die Systemvariablen enthalten sind). Möchten Sie hingegen Programme schreiben, die in einer anderen Bank liegen, sind Sie an einer kompletten Information über die Null-Seite (außer Stelle 0 und 1) interessiert.

Benötigen Sie Platz in der Null-Seite der Bank 15, müssen Sie sich dort ein wenig umsehen. Die Adressen $E6 bis $FF werden vom System nicht benutzt. Speicherstellen $20 bis $2B, $64 bis $6E sind Arbeitsbereiche, die für den vorübergehenden Gebrauch zur Verfügung stehen.

Die meisten Speicherstellen der Null-Seite können in einen anderen Speicherteil in der Weise kopiert werden, daß ihr Originalinhalt später nach der Benutzung zurückgespeichert werden kann. Der Programmierer sollte bei der Veränderung der folgenden Speicherstellen, die innerhalb des Betriebssystems oder des BASIC eine wichtige Rolle spielen, sehr sorgfältig vorgehen: $1A, $1D bis $1F, $2D bis $41, $43, $5B, $78, $85 bis $87, $9E bis $AB, $C0 bis $E5.

Speicherplan

Die folgende Information gilt für B-Systeme, die nach dem April 1983 ausgeliefert wurden und einen überarbeiteten Maschinensprachemonitor enthalten. (Wenn POKE 6,0:SYS 6 keine vollständige Monitoranzeige mit einem Punkt als Eingabeaufforderung ergibt, handelt es sich um eine nichtkompatible Version.)

Zu den erwähnenswerten Eigenschaften im Vergleich zu früheren Commodore Produkten gehört:

- CHRGOT befindet sich nicht mehr im RAM. Programme nach Art des „Keil" müssen bei den Verbindungspunkten $029E und $02A0 eingefügt werden, was die Arbeit erleichtert.
- Die BASIC Zeiger wurden aufgespalten. Beispielsweise gibt es eigene Zeiger für „Variablenanfang" und „Variablenende", die sich vom BASIC-Ende und Arrayanfang unterscheiden. Drei-Byte Zeiger (sie enthalten die Bank Nummer) sind nicht ungewöhnlich.
- Die Sprungtabelle am oberen Speicherbereich ist weiterhin zugänglich und stimmt mit früheren Commodore Produkten einigermaßen überein.
- Einfache Maschinenspracheprogramme passen ohne Schwierigkeiten in den 1K ROM-Ersatzbereich von $0400 bis $07FF. Große Programme muß man entweder in Einsteckspeicher (RAM oder ROM) in Bank 15 implementieren oder in eine andere Bank setzen (vorzugsweise Bank 3). Zusätzlicher Programmaufwand wird benötigt, um all die Programmkomponenten zu koppeln.

Der folgende Plan enthält BASIC Adressen, die für den B256/80 typisch sind. Hinweise auf die Bank 0 bis 4 sind ebenfalls typisch für diese Maschine. Der größte Teil des Plans ist hingegen von allgemeinem Nutzen.

Alle Banks

0000	0	6509 Ausführungsregister
0001	1	6509 Umlenkungsregister

Bank 0: Unbenutzt

Bank 1:

0002-F000	2-61439	BASIC-Programm (Text) RAM
FA5E-FB00	61440-64512	Bereich des Eingabepuffer

Bank 2:

B256:

0002-FFFF	2-65535	BASIC-Arrays im RAM

B128:

0002-FFFF	2-65535	BASIC-Variable, Arrays und Strings Tastendefinitionen

Bank 3: (nur B256)

0002-7FFF	2-32767	Unbenutztes RAM
8000-FFFF	32768-65535	BASIC-Variable im RAM

Bank 4: (nur B256)

0002-FBFF	2-64511	BASIC-Strings im RAM (von oben nach unten)
FC00-FCFF	64512-64767	Unbenutztes RAM (Descriptoren ?)
FD00-FFFF	64768-65535	Laufende Tastendefinitionen

Banks 5 bis 14: Unbenutzt

Bank 15:

0002-0004	2-4	USR Sprung
0005-0008	5-8	Ausgabeelemente von TI$: H, M, S, T
0009-000B	9-11	Zeiger auf Format bei Print Using
000C	12	Suchzeichen
000D	13	Flag für Anführungszeichenmodus
000E	14	Eingabepunkt; Anzahl der Indizes
000F	15	Zeilenzähler bei Catalog
0010	16	Flag für vorgegebenes DIM
0011	17	Typ: 255 = String, 0 = Integer
0012	18	Typ: 128 = Integer, 0 = Fließkomma
0013	19	Flag numerische Berechnung
0014	20	Indexwert
0015	21	INPUT=0, GET=64, READ=152
0016-0019	22-25	Arbeitswerte Diskstatus
001A	26	Laufende I/O Einheit zur Unterdrückung des Prompt
001B-001C	27-28	Integerwert
001D-001F	29-31	Zeiger auf Descriptorstapel
0020-002B	32-43	Verschiedene Arbeitszeiger
002D-002E	45-46	Zeiger BASIC-Anfang

002F-0030	47-48	Zeiger BASIC-Ende
0031-0032	49-50	Zeiger Variablenanfang
0033-0034	51-52	Zeiger Variablenende
0035-0036	53-54	Zeiger Arrayanfang
0037-0038	55-56	Zeiger Arrayende
0039-003A	57-58	Arbeitszeiger Variable
003B-003C	59-60	Zeiger auf Stringende
003D-003E	61-62	Zeiger auf benutzten String
003F-0041	63-65	Zeiger Anfang Stringspeicher
0042-0043	66-67	Laufende BASIC-Zeilennummer
0044-0045	68-69	Alte BASIC-Zeilennummer
0046-0047	70-71	Alter BASIC-Textzeiger
0049-004A	73-74	DATA-Zeilennummer
004B-004C	75-76	DATA-Textzeiger
004D-004E	77-78	Eingabezeiger
004F-0050	79-80	Variablenname
0051-0053	81-83	Variablenadresse
0054-0056	84-86	Zeiger FOR-Schleife
0057-0058	87-88	Zwischenspeicher Textzeiger
005A	90	Akkumulator für Vergleichssymbole
005B-005D	91-92	Funktionsstelle
005E-0060	94-96	Vektor auf Arbeitsstring
0061-0063	97-99	Funktionssprungcode
0064-006E	100-110	Arbeitszeiger, Werte
006F	111	Exponent-Vorzeichen
0070	112	Akku Stringprefix
0071	113	Akku #1: Exponent
0072-0075	114-117	Akku #1: Mantisse
0076	118	Akku #1: Vorzeichen
077	119	Zeiger Funktionsauswertung
0078	120	Akku #1 höherwertig (Überlauf)
0079-007E	121-126	Akku #2
007F	127	Vorzeichenvergleich, Akku #1 gegen Akku #2
0080	128	Akku #1 niederwertig (runden)
0081-0084	129-132	Arbeitszeiger
0085-0087	133-135	Zeiger BASIC-Text
0088-0089	136-137	Eingabezeiger
008B-008E	139-142	DOS Parser-Arbeitsbereiche
008F	143	Nummer des Fehlertyps
0090-0092	144-146	Zeiger auf Filename
0093-0095	147-149	Zeiger: Bandpuffer, Scrolling
0096-0098	150-152	Laden Endadresse / Programmende
0099-009B	153-155	I/O Startadresse
009C	156	Statuswort ST

009D	157	Länge des Filenamen
009E	158	Laufender logischer File
009F	159	Laufende Einheit
00A0	160	Laufende Sekundäradresse
00A1	161	Eingabeeinheit, normalerweise 0
00A2	162	CMD Ausgabeeinheit, normalerweise 3
00A6-00A8	166-168	INBUF
00A9	169	Hauptschalter PIA: Stoptaste, usw.
00AA	170	IEEE Verzögerungsflag
00AB	171	IEEE verzögertes Zeichen
00AC-00AD	172-173	Vektor Segmenttransferroutine
00AE-00B3	174-179	Zwischenspeicher Monitorregister
00B4	180	Zwischenspeicher Monitorstapelzeiger
00B5	181	Zwischenspeicher Monitorbanknummer
00B7-00B8	183-184	Zeiger Speicher Monitor IRQ
00B9-00BA	185-186	Zeiger Monitorspeicher
00BB-00BC	187-188	Monitor Sekundärzeiger
00BD	189	Monitorzähler
00BE	190	Monitor, unterschiedliche Byte
00BF	191	Monitor Einheitennummer
00C0-00C1	192-193	Adresse der Tabelle programmierbarer Tasten
00C2-00C3	194-195	Adresse programmierbare Taste
00C4-00C7	196-199	Zeiger zum Ändern der Tabelle programmierbarer Tasten
00C8-00C9	200-201	Zeiger auf Bildschirmzeile
00CA	202	Bildschirmzeilennummer
00CB	203	Cursorposition auf Zeile
00CC	204	0 = Textmodus, sonst Grafikmodus
00CD	205	Variable gedrückte Taste
00CE	206	Alte Cursorspalte
00CF	207	Alte Cursorzeile
00D0	208	Flag neues Zeichen
00D1	209	Anzahl der Zeichen in Tastaturpuffer
00D2	210	Flag Anführungszeichen
00D3	211	Zähler Inserttaste
00D4	212	Flag Cursortyp
00D5	213	Länge Bildschirmzeile
00D6	214	Anzahl Tasten in „Tasten"puffer
00D7	215	Tastenwiederholverzögerung
00D8	216	Tastenwiederholgeschwindigkeit
00D9-00DA	217-218	Temporäre Variable
00DB	219	Laufendes Ausgabezeichen
00DC	220	Oberste Zeile des laufenden Bildschirms
00DD	221	Unterste Zeile des Bildschirms

00DE	222	Linke Kante des laufenden Bildschirms
00DF	223	Rechte Kante des Bildschirms
00E0	224	Tasten: 255 = keine; 127 = Taste; 111 = Shift
00E1	225	Taste gedrückt: 255 = keine Taste
00E2-00E5	226-229	Bits Zeilenwechsel
0100	256	Hex nach binär Zwischenbereich
0100-010A	256-266	Numerisch nach ASCII Arbeitsbereich
0100-01FE	256-510	Stapelbereich
01FF	511	Zwischenspeicher Stapelzeiger
0200-020F	512-527	Bereich Filenamen
0210-0226	528-550	Arbeitsbereich Diskbefehle
0255-0256	597-598	Unterschiedliche Arbeitswerte für WAIT usw.
0257	599	„Bank"-Wert
0258	600	Ausgabe logischer File (CMD)
0259	601	Vorzeichen TAN
025A-025D	602-605	Unterroutinenaufnahme; unterschiedliche Arbeitswerte
025E-0276	606-630	Arbeitsvariable für PRINT USING
0280-0281	640-641	Verbindung Fehlerroutine (8555)
0282-0283	642-643	Verbindung Warmstart (85CD)
0284-0285	644-645	Verbindung Berechnung aufgenommen (88C2)
0286-0287	646-647	List-Verbindung (89F4)
0288-0289	648-649	Verbindung Befehl absenden (8754)
028A-028B	650-651	Verbindung Aufgenommenes bewerten (96B1)
028C-028D	652-653	Verbindung Ausdruck bewerten (95C4)
028E-028F	654-655	CHRGOT Verbindung (BA2C)
0290-0291	656-657	CHRGET Vektor (BA32)
0292-0293	658-659	Float-fixed Vektor (BA1E)
0294-0295	660-661	Fixed-Float Vektor (9D39)
0296-0297	662-663	Vektor Fehlerfalle
0298-0299	664-665	Zeilennummer Fehler
029A-029B	666-667	Zeiger Fehlerausgang
029C	668	Zwischenspeicher Stapelzeiger
029D-029F	669-671	Temporärer TRAP, DISPOSE Bytes
02A0-02A5	672-677	Temporäre INSTR$ Bytes
02A6-02A7	678-679	Bank Offset
0300-0301	768-769	IRQ Vektor (FBE9)
0302-0303	770-771	BRK Vektor (EE21)
0304-0305	772-773	NMI Vektor (FCAA)
0306-0307	774-775	OPEN Vektor (F6BF)
0308-0309	776-777	CLOSE Vektor (F5ED)
030A-030B	778-779	Vektor Verbinde Eingabe (F549)
030C-030D	780-781	Vektor Verbinde Ausgabe (F5A3)
030E-030F	782-783	Vektor Schalte auf normales I/O (F6A6)
0310-0311	784-785	Eingabevektor (F49C)
0312-0313	786-787	Ausgabevektor (F4EE)

0314-0315	788-789	Vektor Test Stop-Taste (F96B)
0316-0317	790-791	GET Vektor (F43D)
0318-0319	792-793	Vektor Verwirf alle Files (F67F)
031A-031B	794-795	Ladevektor (F746)
031C-031D	796-797	Speichervektor (F84C)
031E-031F	798-799	Vektor Monitorbefehl (EE77)
0320-0321	800-801	Tastaturkontrollvektor (E01F)
0322-0323	802-803	Printkontrollvektor (E01F)
0324-0325	804-805	IEEE Vektor Sende LSA (F274)
0326-0327	806-807	IEEE Vektor Sende TSA (F280)
0328-0329	808-809	IEEE Vektor Empfange Byte (F30A)
032A-032B	810-811	IEEE Vektor Sende Zeichen (F297)
032C-032D	812-813	IEEE Vektor Sende Untalk (F2AB)
032E-032F	814-815	IEEE Vektor Sende Unlisten (F2AF)
0330-0331	816-817	IEEE Vektor Sende Listen (F234)
0332-0333	818-819	IEEE Vektor Sende Talk (F230)
0334-033D	820-829	Tabelle von Adressen logischer Files
033E-0347	830-839	Tabelle von File-Einheiten
0348-0351	840-849	Tabelle von File-Sekundäradressen
0352-0354	850-852	Untere System-Speichergrenze
0355-0357	853-855	Obere System-Speichergrenze
0358-035A	856-858	Untergrenze Benutzerspeicher
035B-035D	859-861	Obergrenze Benutzerspeicher
035E	862	IEEE Zeitüberschreitung; 0 = zugelassen
035F	863	0 = Laden; 128 = Verifizieren
0360	864	Anzahl offener Files
0361	865	Byte Nachrichtenmodus
0363-0366	867-870	Zwischenspeicher unterschiedlicher Registerbytes
0369	873	Zeitgeber Umschalter
036A-036B	874-875	Kassettenvektor (Sackgasse)
036F-0371	879-881	Startadresse Relocation
0375	885	Flag Kassettenmotor (unbenutzt)
0376-0377	886-887	RS232 Kontrolle, Befehl
037A	890	RS232 Status
037B	891	RS232 Quittungseingabe
037C	892	RS232 Eingabezeiger
037D	893	RS232 Ankunftszeiger
0380-0381	896-897	Zeiger Obere Speichergrenze
0382	898	Bank Byte
0383	899	RVS Flag
0384	900	Länge laufender Zeile
0385	901	Zwischenspeicher temporäres Ausgabezeichen
0386	902	0 = normal, 255 = auto insert
0387	903	0 = srolling, 255 = kein Scrolling
0388	904	Unterschiedliche Arbeitsbyte für Bildschirm

0389	905	Index auf programmierte Taste
038A	906	Flag Scrollmodus
038B	907	Flag Piepmodus
038C	908	Zwischenspeicher indirekte Bank
038D-03A0	909-928	Länge der „Tasten"-Worte
03A1-03AA	929-938	Bitmapped Tabulatorstops
03AB-03B4	939-948	Tastatur Eingabepuffer
03B5-03B6	949-950	Verbindung der „Tasten"-Worte (E91B)
03F8-03F9	1016-1017	Restart Vektor
03FA-03FB	1018-1019	Restart Testmaske
0400-07FF	1024-2047	Freies RAM (für DOS reserviert)
0800-0FFF	2048-4095	Reserviert für Einsteck-RAM
1000-1FFF	4096-8191	Reserviert für Einsteck-DOS-ROM
2000-7FFF	8192-23767	Reserviert für Kartuschen
8000-BFFF	32768-49151	BASIC ROM
C000-CFFF	49152-53247	Unbenutzt
D000-D7CF	53248-55247	Bildschirm RAM
D800-D801	55296-55297	Videokontroller 6545
DA00-DA1C	55808-55836	Toninterface-Einheit (SID) 6581
DB00-DB0F	56064-56079	Komplexer Interface Adapter (CIA) 6526
DC00-DC0F	56320-56335	Komplexer Interface Adapter (CIA) 6526
DD00-DD03	56576-56579	Asynchrone Kommunikationen IA 6551
DE00-DE07	56832-56839	Tri Port Interface 6525
DF00-DF07	57088-57095	Tri Port Interface 6525
E000-FFFF	57344-65535	Kernal ROM

Die obige Tabelle zeigt Werte für die Verbindungs- und Vektoradressen zwischen $0280 und $0295; diese stammen von einem B-128 jüngeren Datums.

6545 CRT Kontroller

D800 55296	D801 55297	Typische Werte (Dezimal)
0	Horizontal Total	108 oder 126 oder 127
1	Horizontal Char angezeigt	80
2	Horizontal Sync Position	83 oder 98 oder 96
3	V Sync Breite H	15 oder 10
4	Vertical Total	25 oder 31 oder 38
5	Vert Total just.	3 oder 6 oder 1
6	Vertikal angezeigt	25
7	Vert Sync Position	25 oder 28 oder 30
8	Mode	0
9	Scan Linien	13 oder 7
10	Cursor Start	96 (blinken) oder 0 oder 6 (unterstreichen)
11	Cursor Ende	13 oder 7
12	Display Adresse H	0
13	Display Adresse L	0
14	Cursor Adresse H	Variiert
15	Cursor Adresse L	Variiert
16	Lichtstift In H	0
17	Lichtstift In L	0

Die meisten Register sind nur beschreibbar
14/15 sind Lese/Schreibregister
16/17 sind nur lesbar
Register 10,14 und 15 verändern sich mit der Cursorbewegung

Abbildung C.14

6525 Tri Port

DE00	NRFD	NDAC	EOI	DAV	ATN	RFN		56832
DE01	Sense	Cassette Motor	Out	ARB	Network Rx	Tx	SRQ	IFC 56833
DE02								56834
DE03	Data Direction Register For DE00							56835
DE04	Data Direction Register For DE01							56836
DE05	IRQ			ACIA	IP	ALM	IEEE	PWR 56837
DE06	CB		CA Graphics				IRQ Stack On	56838
DE07	Active Interrupt Register							56839

6525 Tri Port 2

DF00	Keyboard		57088
DF01	Select		57089
DF02	CRT Mode	Keyboard Read	57090
DF03	Data Direction Register for DF00 (out)		57091
DF04	Data Direction Register for DF01 (out)		57092
DF05	Data Direction Register for DF02 (in)		57093
DF06	Unused		57094

Abbildung C.15

Commodore 64: ROM Einzelheiten

Diese Art des ROM Speicherplans ist ursprünglich für Benutzer vorgesehen, die die innere Logik des Computers durchstreifen wollen. Er erlaubt, einen interessanten Bereich zu disassemblieren und den Hintergrund für das Verhalten des Computers zu untersuchen. Mit Hilfe dieses Plans ist der Benutzer in der Lage, Unterroutinen zu erkennen, die von dem untersuchten Programmteil aufgerufen werden.

Ich bin dagegen, ROM Unterroutinen als Teil eines eigenen Programms zu benutzen. Diese führen meist nicht präzise Ihre Wünsche aus. Wenn Sie auf eine andere Maschine übergehen, liegen sie häufig an einer unterschiedlichen Stelle. Mit wenigen Ausnahmen können Sie wahrscheinlich selbst ein besseres Programm zur Erledigung dieser Aufgabe schreiben. Beschäftigen Sie sich intensiv mit der Sprungtabelle der Kernroutinen: speziell mit $FFD2 für die Ausgabe, mit $FFE4 für die Eingabe, $FFE1 zur Überprüfung der RUN/STOP Taste, mit $FFC6 und $FFC9 zum Umschalten der Eingabe bzw. Ausgabe und mit $FFCC, um die normalen Eingabe/Ausgabe Kanäle wiederherzustellen. Diese sind bei allen Commodore Computern gleich.

A000: ROM Kontrollvektoren
A00C: Schlüsselwort Aktionsvektoren

A052: Funktionsvektoren
A080: Operatorvektoren
A09E: Schlüsselworte
A19E: Fehlermeldungen
A328: Fehlermeldungsvektoren
A365: Verschiedene Meldungen
A38A: Überstreiche Stapel bei FOR/GOSUB
A3B8: Verschiebe Speicher
A3FB: Prüfe Stapeltiefe
A408: Prüfe freien Speicher
A435: Drucke „Out of memory"
A437: Fehlerroutine
A469: BREAK Einsprungpunkt
A474: Drucke „ready."
A480: Ready bei BASIC
A49C: Bearbeite neue Zeile
A533: Sortiere Zeilen neu
A560: Empfange Eingabezeile
A579: Verarbeite Annahmen
A613: Finde BASIC-Zeile
A642: Ausführung (NEW)
A65E: Ausführung (CLR)
A68E: Sichere Textzeiger
A69C: Ausführung (LIST)
A742: Ausführung (FOR)
A7ED: Führe Befehl aus
A81D: Ausführung (RESTORE)
A82C: Break
A82F: Ausführung (STOP)
A831: Ausführung (END)
A857: Ausführung (CONT)
A871: Ausführung (RUN)
A883: Ausführung (GOSUB)
A8A0: Ausführung (GOTO)
A8D2: Ausführung (RETURN)
A8F8: Ausführung (DATA)
A906: Suche nächsten Befehl
A928: Ausführung (IF)
A93B: Ausführung (REM)
A94B: Ausführung (ON)
A96B: Hole Festkommazahl
A9A5: Ausführung (LET)
AA80: Ausführung (PRINT#)
AA86: Ausführung (CMD)
AAA0: Ausführung (PRINT)

AB1E: Drucke String von (Y.A)
AB3B: Drucke Formatzeichen
AB4D: Routine Schlechte Eingabe
AB7B: Ausführung (GET)
ABA5: Ausführung (INPUT#)
ABBF: Ausführung (INPUT)
ABF9: Prompt und Input
AC06: Ausführung (READ)
ACFC: Input Fehlermeldungen
AD1E: Ausführung (NEXT)
AD78: Art Vergleichsprüfung
AD9E: Berechnung Ausdruck
AEA8: Konstante Pi
AEF1: Berechnung innerhalb Klammern
AEF7: Prüfe auf „)"
AEFF: Prüfe auf Komma
AF08: Syntaxfehler
AF14: Prüfe Bereich
AF28: Suche Variable
AFA7: Aufstellung FN Referenz
AFE6: Berechnung (OR)
AFE9: Berechnung (AND)
B016: Vergleiche
B081: Ausführung (DIM)
B08B: Bestimme Variable
B113: Prüfe alphabetisch
B11D: Erzeuge Variable
B194: Unterroutine Arrayzeiger
B1A5: Wert 32768
B1B2: Fließkomma-Festkomma
B1D1: Array Aufstellung
B245: Drucke „bad subscript"
B248: Drucke „illegal quantity"
B34C: Berechne Arraygröße
B37D: Berechnung (FRE)
B391: Festkomma-Fließkomma
B39E: Berechnung (POS)
B3A6: Prüfe direkt
B3B3: Ausführung (DEF)
B3E1: Prüfe FN Syntax
B3F4: Berechnung (FN)
B465: Berechnung (STR$)
B475: Berechne Stringvektor
B487: Aufstellung String
B4F4: Schaffe Platz für String

B526: Garbage Collection
B5BD: Prüfe Rettbarkeit
B606: Sammle String
B63D: Verknüpfe
B67A: Füge String in Speicher ein
B6A3: Verwirf unerwünschte Strings
B6DB: Säubere Descriptor-Stapel
B6EC: Berechnung (CHR$)
B700: Berechnung (LEFT$)
B72C: Berechnung (RIGHT$)
B737: Berechnung (MID$)
B761: Lege Stringparameter ab
B77C: Berechnung (LEN)
B782: Verlasse Stringmodus
B78B: Berechnung (ASC)
B79B: Input Byte Parameter
B7AD: Berechnung (VAL)
B7EB: Parameter für POKE/WAIT
B7F7: Fließkomma-Festkomma
B80D: Berechnung (PEEK)
B824: Ausführung (POKE)
B82D: Ausführung (WAIT)
B849: Addiere 0.5
B850: Subtrahiere von
B853: Berechnung (Subtraktion)
B868: Berechnung (Addition)
B947: Komplementiere FAC (Floating accumulator) #1
B97E: Drucke „overflow"
B983: Multipliziere mit Null-Byte
B9EA: Berechnung (LOG)
BA2B: Berechnung (Multiplikation)
BA59: Multipliziere ein Bit
BA8C: Speicher nach FAC#2
BAB7: Justiere FAC#1 und FAC#2
BAD4: Underflow/Overflow
BAE2: Multipliziere mit 10
BAF9: +10 in Fließkomma
BAFE: Dividiere durch 10
BB12: Berechnung (Division)
BBA2: Speicher nach FAC#1
BBC7: FAC#1 nach Speicher
BBFC: FAC#2 nach FAC#1
BC0C: FAC#1 nach FAC#2
BC1B: Runde FAC#1
BC2B: Hole Vorzeichen

BC39: Berechnung (SGN)
BC58: Berechnung (ABS)
BC5B: Vergleiche FAC#1 mit Speicher
BC9B: Fließkomma-Festkomma
BCCC: Berechnung (INT)
BCF3: String nach FAC
BD7E: Hole ASCII Ziffer
BDC2: Drucke „IN. . "
BDCD: Drucke Zeilennummer
BDDD: Fließkomma nach ASCII
BF16: Dezimalkonstanten
BF3A: TI Konstanten
BF71: Berechnung (SQR)
BF7B: Berechnung (Exponent)
BFB4: Berechnung (negativ)
BFED: Berechnung (EXP)
E043: Reihenberechnung 1
E059: Reihenberechnung 2
E097: Berechnung (RND)
E0F9: Kernal-Aufrufe mit Fehlerprüfung
E12A: Ausführung (SYS)
E156: Ausführung (SAVE)
E165: Ausführung (VERIFY)
E168: Ausführung (LOAD)
E1BE: Ausführung (OPEN)
E1C7: Ausführung (CLOSE)
E1D4: Parameter für LOAD/SAVE
E206: Prüfe vorgegebene Parameter
E20E: Prüfe auf Komma
E219: Parameter für Open/Close
E264: Berechnung (COS)
E26B: Berechnung (SIN)
E2B4: Berechnung (TAN)
E30E: Berechnung (ATN)
E37B: Warmstart
E394: Initialisiere
E3A2: CHRGET für Null-Seite
E3BF: Initialisiere BASIC
E447: Vektoren for $300
E453: Initialisiere Vektoren
E45F: Einschaltmeldung
E500: Hole I/O Adresse
E505: Hole Bildschirmgröße
E50A: Gib/Hol Zeile/Spalte
E518: Initialisiere I/O

E544: Lösche Bildschirm
E566: Cursor HOME
E56C: Setze Bildschirmzeiger
E5A0: Setze vorgegebene I/O Werte
E5B4: Eingabe von Tastatur
E632: Eingabe von Bildschirm
E684: Test Anführungszeichen
E691: Aufstellung Bildschirmausgabe
E6B6: Cursor vorrücken
E6ED: Cursor zurücknehmen
E701: Zurück in vorige Zeile
E716: Ausgabe auf Bildschirm
E87C: Geh zur nächsten Zeile
E891: Ausführung (return)
E8A1: Prüfe Zeilenabnahme
E8B3: Prüfe Zeilenzunahme
E8CB: Setze Farbcode
E8DA: Farbcodetabelle
E8EA: Scroll Bildschirm
E965: Öffne Leerstelle auf Bildschirm
E9C8: Bewege eine Bildschirmzeile
E9E0: Synchronisiere Farbtransfer
E9F0: Setze Zeilenanfang
E9FF: Lösche Bildschirmzeile
EA13: Drucke auf Bildschirm
EA24: Synchronisiere Farbzeiger
EA31: Interrupt-Clock usw.
EA87: Lesen Tastatur
EB79: Vektoren Tastaturauswahl
EB81: Tastatur 1 - Nichtshift
EBC2: Tastatur 2 - Shift
EC03: Tastatur 3 - „Commodore" Shift
EC44: Kontrolle Grafik/Text
EC4F: Setze Grafik/Textmodus
EC78: Tastatur 4
ECB9: Voreinstellung Videochip
ECE7: SHIFT/RUN Äquivalent
ECF0: Bild in Adresse niederwertig
ED09: Sende „talk" auf seriellen Bus
ED0C: Sende „listen" auf seriellen Bus
ED40: Sende auf seriellen Bus
EDB2: Seriell Zeitüberschreitung
EDB9: Sende Listen SA
EDBE: Lösche ATN
EDC7: Sende Talk SA

EDCC: Warte auf Clock
EDDD: Sende seriell verzögert
EDEF: Sende „untalk" auf seriellen Bus
EDFE: Sende „unlisten" auf seriellen Bus
EE13: Empfange von seriellen Bus
EE85: Serielle Clock an
EE8E: Serielle Clock aus
EE97: Serielle Ausgabe „1"
EEA0: Serielle Ausgabe „0"
EEA9: Hole serielle Eingangs- und Clocksignale
EEB3: Verzögerung 1 Millisekunde
EEBB: RS232 Senden
EF06: Sende neues RS232 Byte
EF2E: Kein DSR Fehler
EF31: Kein CTS Fehler
EF3B: Sperre Zeitgeber
EF4A: Berechne Bitzahl
EF59: RS232 Empfang
EF7E: Voreinstellung zum Empfang
EFC5: Empfang Paritätsfehler
EFCA: Empfang Überlauf
EFCD: Empfang Break
EFD0: Ratenfehler
EFE1: Übertrage an RS232
F00D: Kein DSR Fehler
F017: Sende an RS232 Puffer
F04D: Eingabe von RS232
F086: Hole von RS232
F0A4: Prüfe ob serieller Bus frei
F0BD: Meldungen
F12B: Drucke wenn direkt
F13E: Hole ...
F14E: ... von RS232
F157: Eingabe
F199: Hole: Band/Seriell/RS232
F1CA: Ausgabe ...
F1DD: ... auf Band
F20E: Setze Eingabeeinheit
F250: Setze Ausgabeeinheit
F291: Schließe File
F30F: Finde File
F31F: Setze Filewerte
F32F: Verwirf alle Files
F333: Wiederherstellung vorgegebener I/O
F34A: Eröffne File

F3D5: Sende SA
F409: Öffne RS232
F49E: Lade Programm
F5AF: Drucke „searching"
F5C1: Drucke Filenamen
F5D2: Drucke „loading/verifying"
F5DD: Speichere Programm
F68F: Drucke „saving"
F69B: Clock anstoßen
F6BC: Zwischenspeicher PIA Tastenlesen
F6DD: Hole Zeit
F6E4: Setze Zeit
F6ED: Prüfe STOP-Taste
F6FB: Ausgabe Fehlermeldungen
F72D: Finde irgendeinen Bandvorspann
F76A: Schreibe Bandvorspann
F7D0: Hole Pufferadresse
F7D7: Setze Zeiger Pufferanfang -ende
F7EA: Finde bestimmten Vorspann
F80D: Zeiger Bandanstoß
F817: Drucke „press play ..."
F82E: Prüfe Bandstatus
F838: Drucke „press record ..."
F841: Initialisiere Bandlesen
F864: Initialisiere Bandschreiben
F875: Allgemeiner Bandcode
F8D0: Prüfe Bandstop
F8E2: Setze Zeitgeber Lesen
F92C: Lesen Bandbits
FA60: Speichere Bandzeichen
FB8E: Setze Zeiger zurück
FB97: Neues Zeichen einstellen
FBA6: Sende Übergang auf Band
FBC8: Schreibe Daten auf Band
FBCD: IRQ Einsprungpunkt
FC57: Schreibe Bandvorspann
FC93: Normales IRQ wiederherstellen
FCB8: Setze IRQ Vektor
FCCA: Bandmotor abschalten
FCD1: Prüfe R/W Zeiger
FCDB: Zeiger R/W anstoßen
FCE2: Einsprungpunkt bei Einschaltreset
FD02: Prüfe 8-ROM
FD10: 8-ROM Maske
FD15: Kernal Reset

FD1A: Kernal Verschiebung
FD30: Vektoren
FD50: Initialisiere Systemkonstanten
FD9B: IRQ Vektoren
FDA3: Initialisiere I/O
FDDD: Zeitgeber eingeschaltet
FDF9: Speichere Filenamendaten
FE00: Speichere Filedetails
FE07: Hole Status
FE18: Flagstatus
FE1C: Setze Status
FE21: Setze Zeitüberschreitung
FE25: Lesen/Setzen obere Speichergrenze
FE27: Lesen obere Speichergrenze
FE2D: Setzen obere Speichergrenze
FE34: Lesen/Setzen untere Speichergrenze
FE43: NMI Einsprungpunkt
FE66: Warmstart
FEB6: Rücksetzen IRQ und Ausgang
FEBC: Interruptausgang
FEC2: RS232 Zeitgebertabelle
FED6: NMI RS232 Eingabe
FF07: NMI RS232 Ausgabe
FF43: IRQ vortäuschen
FF48: IRQ Einsprungpunkt
FF81: Jumbo-Sprungtabelle
FFFA: Hardware-Vektoren

Anhang D
Zeichensätze

Supertabelle

Die „Supertabelle" zeigt die Zeichensätze des PET. In Abhängigkeit von seiner Verwendung kann ein Byte ganz unterschiedliche Bedeutungen erlangen. Um das wiederzugeben, wurde die Tabelle besonders angelegt. „ASCII" bedeutet PET ASCII. Das sind die Zeichen, die eingegeben oder ausgedruckt werden. Mit „Screen" ist der Code des Commodore Bildschirms gemeint, wie er im Bildschirmspeicher benutzt wird. Durch Anwendung der Befehle POKE oder PEEK auf den Bildschirmspeicher ergeben sich diese Codes. Beachten Sie, daß der Zeichensatz für Zahlen sowohl für den Bildschirm als auch für PET ASCII gleich ist.

Innerhalb eines Programms verändert sich der Code abermals. „BASIC" zeigt diese Codes. Sie sind im Bereich $20 bis $5F zu ASCII gleich.

Der Bequemlichkeit und Vollständigkeit wegen sind die Maschinensprache Opcodes hinzugefügt.

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
0	00		@	end-line	BRK	0
1	01		A		ORA(I,X)	1
2	02		B			2
3	03		C			3
4	04		D			4
5	05	white	E		ORA Z	5
6	06		F		ASL Z	6
7	07	bell	G			7
8	08	lock	H		PHP	8
9	09	unlock	I		ORA #	9
10	OA		J		ASL A	10
11	OB		K			11
12	OC		L			12
13	OD	car ret	M		ORA	13
14	OE	text	N		ASL	14
15	OF	top	O			15
16	10		P		BPL	16
17	11	cur down	Q		ORA(I),Y	17
18	12	reverse	R			18
19	13	cur home	S			19
20	14	delete	T			20
21	15	del. line	U		ORA Z,X	21
22	16	ers.begin	V		ASL Z,X	22
23	17		W			23
24	18		X		CLC	24
25	19	scr. up	Y		ORA Y	25
26	1A		Z			26
27	1B		[			27
28	1C	red	\			28
29	1D	cur right	]		ORA X	29

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
30	1E	green	↑		ASL X	30
31	1F	blue	←			31
32	20	space	space	space	JSR	32
33	21	!	!	!	AND(I,X)	33
34	22	"	"	"		34
35	23	#	#	#		35
36	24	$	$	$	BIT Z	36
37	25	%	%	%	AND Z	37
38	26	&	&	&	ROL Z	38
39	27	'	'	'		39
40	28	(	(	(	PLB	40
41	29	)	)	)	AND #	41
42	2A	*	*	*	ROL A	42
43	2B	+	+	+		43
44	2C	,	,	,	BIT	44
45	2D	-	-	-	AND	45
46	2E	.	.	.	ROL	46
47	2F	/	/	/		47
48	30	0	0	0	BMI	48
49	31	1	1	1	AND(I),Y	49
50	32	2	2	2		50
51	33	3	3	3		51
52	34	4	4	4		52
53	35	5	5	5	AND Z,X	53
54	36	6	6	6	ROL Z,X	54
55	37	7	7	7		55
56	38	8	8	8	SEC	56
57	39	9	9	9	AND Y	57
58	3A	:	:	:	CLI	58
59	3B	;	;	;		59
60	3C	<	<	<		60
61	30	=	=	=	AND X	61
62	3E	>	>	>	ROL X	62
63	3F	?	?	?		63
64	40	@	▬	@	RTI	64
65	41	A	♠, a	A	EOR(I,X)	65
66	42	B	▌, b	B		66
67	43	C	▄, c	C		67
68	44	D	▔, d	D		68
69	45	E	▐, e	E	EOR Z	69
70	46	F	▂, f	F	LSR Z	70
71	47	G	▖, g	G		71
72	48	H	▗, h	H	PHA	72
73	49	I	▘, i	I	EOR #	73
74	4A	J	▝, j	J	LSR A	74
75	4B	K	▚, k	K		75
76	4C	L	▙, l	L	JMP	76
77	4D	M	▚, m	M	EOR	77
78	4E	N	▞, n	N	LSR	78
79	4F	O	▛, o	O		79
80	50	P	▜, p	P	BVC	80

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
81	51	Q	, q	Q	EOR(I),Y	81
82	52	R	, r	R		82
83	53	S	, s	S		83
84	54	T	, t	T		84
85	55	U	, u	U	EOR, Z,X	85
86	56	V	, v	V	LSR Z,X	86
87	57	W	, w	W		87
88	58	X	, x	X	CLI	88
89	59	Y	, y	Y	EOR Y	89
90	5A	Z	, z	Z		90
91	5B	[		[		91
92	5C	\		\		92
93	5D	]		]	EOR X	93
94	5E	↑	,	↑	LSR X	94
95	5F	←	,	←		95
96	60				RTS	96
97	61				ADC(I,X)	97
98	62					98
99	63					99
100	64					100
101	65				ADC Z	101
102	66				ROR Z	102
103	67					103
104	68				PLA	104
105	69		,		ADC #	105
106	6A				ROR A	106
107	6B					107
108	6C				JMP(I)	108
109	6D				ADC	109
110	6E				ROR	110
111	6F					111
112	70				RVS	112
113	71				ADC(I),Y	113
114	72					114
115	73					115
116	74					116
117	75				ADC Z,X	117
118	76				ROR Z,X	118
119	77					119
120	78				SEI	120
121	79				ADC Y	121
122	7A		,			122
123	7B					123
124	7C					124
125	7D				ADC X	125
126	7E				ROR X	126
127	7F					127
128	80		r-(α	END		128
129	81	orange	r-A	FOR	STA(I,X)	129
130	82		r-B	NEXT		130
131	83		r-C	DATA		131

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
132	84		r-D	INPUT#	STY Z	132
133	85		r-E	INPUT	STA Z	133
134	86		r-F	DIM	STX Z	134
135	87		r-G	READ		135
136	88		r-H	LET	DEY	136
137	89		r-I	GOTO		137
138	8A		r-J	RUN	TXA	138
139	8B		r-K	IF		139
140	8C		r-L	RESTORE	STY	140
141	8D	car ret	r-M	GOSUB	STA	141
142	8E	graphic	r-N	RETURN	STX	142
143	8F	bottom	r-O	REM		143
144	90	black	r-P	STOP	BCC	144
145	91	cur up	r-Q	ON	STA(I), Y	145
146	92	rvs off	r-R	WAIT		146
147	93	clear	r-S	LOAD		147
148	94	insert	r-T	SAVE	STY Z,X	148
149	95	ins. line/br	r-U	VERIFY	STA Z,X	149
150	96	ers. end/p	r-V	DEF	STX Z,Y	150
151	97	Gray 1	r-W	POKE		151
152	98	Gray 2	r-X	PRINT#	TYA	152
153	99	scr. down	r-Y	PRINT	STA Y	153
154	9A	L. Blue	r-Z	CONT	TXS	154
155	9B	Gray 3	r-[	LIST		155
156	9C	magenta	r-\	CLR		156
157	9D	cur left	r-]	CMD	STA X	157
158	9E	yellow	r- ↑	SYS		158
159	9F	cyan	r-←	OPEN		159
160	A0		■	CLOSE	LDY #	160
161	A1	▌	r-!	GET	LDA(I,X)	161
162	A2	▄	r-"	NEW	LDX #	162
163	A3	▔	r-#	TAB(		163
164	A4	▁	r-$	TO	LDY Z	164
165	A5	▏	r-%	FN	LDA Z	165
166	A6	▒	r-&	SPC(	LDX Z	166
167	A7	▐	r-'	THEN		167
168	A8	▚	r-(	NOT	TAY	168
169	A9	◤ ▨	r-)	STEP	LDA #	169
170	AA	▐	r-*	+	TAX	170
171	AB	▙	r-+	−		171
172	AC	▗	r-,	*	LDY	172
173	AD	▖	r--	/	LDA	173
174	AE	▝	r-.	↑	LDX	174
175	AF	▂	r-/	AND		175
176	B0	▜	r-0	OR	BCS	176
177	B1	▟	r-1	>	LDA(I),Y	177
178	B2	▛	r-2	=		178
179	B3	▌	r-3	<		179
180	B4	▏	r-4	SGN	LDY Z,X	180
181	B5	▏	r-5	INT	LDA Z,X	181
182	B6	▐	r-6	ABS	LDX Z,Y	182

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
183	B7		r-7	USR		183
184	B8		r-8	FRE	CLV	184
185	B9		r-9	POS	LDA Y	185
186	BA		r-:	SQR	TSX	186
187	BB		r-;	RND		187
188	BC		r-<	LOG	LDY X	188
189	BD		r-=	EXP	LDA X	189
190	BE		r->	COS	LDX Y	190
191	BF		r-?	SIN		191
192	C0		TAN		CPY#	192
193	C1	. a	ATN		CMP(I),X	193
194	C2	. b	PEEK			194
195	C3	. c	LEN			195
196	C4	. d	STR$		CPY Z	196
197	C5	. e	VAL		CMP Z	197
198	C6	. f	ASC		DEC Z	198
199	C7	. g	CHR$			199
200	C8	. h	LEFT$		INY	200
201	C9	. i	RIGHT$		CMP #	201
202	CA	. j	MID$		DEX	202
203	CB	. k	GO			203
204	CC	. l	CONCAT		CPY	204
205	CD	. m	DOPEN		CMP	205
206	CE	. n	DCLOSE		DEC	206
207	CF	. o	RECORD			207
208	DO	. p	HEADER		BNE	208
209	D1	. q	COLLECT		CMP(I),Y	209
210	D2	. r	BACKUP			210
211	D3	. s	COPY			211
212	D4	. t	APPEND			212
213	D5	. u	DSAVE		CMP Z,X	213
214	D6	. v	DLOAD		DEC Z,X	214
215	D7	. w	CATALOG			215
216	D8	. x	RENAME		CLD	216
217	D9	. y	SCRATCH		CMP Y	217
218	DA	. z	DIRECTORY			218
219	DB					219
220	DC					220
221	DD				CMP X	221
222	DE	.			DEC X	222
223	DF	.				223
224	EO				CPX #	224
225	E1				SBC(I),X	225
226	E2					226
227	E3					227
228	E4				CPX Z	228
229	E5				SBC Z	229
230	E6				INC Z	230
231	E7					231
232	E8				INX	232
233	E9				SBC #	233

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
234	EA				NOP	234
235	EB					235
236	EC				CPX	236
237	ED				SBC	237
238	EE				INC	238
239	EF					239
240	FO				BEQ	240
241	F1				SBC(I), Y	241
242	F2					242
243	F3					243
244	F4					244
245	F5				SBC Z,X	245
246	F6				INC Z,X	246
247	F7					247
248	F8				SED	248
249	F9				SBC Y	249
250	FA					250
251	FB					251
252	FC					252
253	FD				SBC X	253
254	FE				INC X	254
255	FF			π		255

Darstellung der Kontrollzeichen

NUL	Null	DLE	Data Link Escape (CC)
SOH	Start of Heading (CC)	DC1	Device Control 1
STX	Start of Text (CC)	DC2	Device Control 2
ETX	End of Text (CC)	DC3	Device Control 3
EOT	End of Transmission (CC)	DC4	Device Control 4
ENQ	Enquiry (CC)	NAK	Negative Acknowledge (CC)
ACK	Acknowledge (CC)	SYN	Synchronous Idle (CC)
BEL	Bell	ETB	End of Transmission Block (CC)
BS	Backspace (FE)	CAN	Cancel
HT	Horizontal Tabulation (FE)	EM	End of Medium
LF	Line Feed (FE)	SUB	Substitute
VT	Vertical Tabulation (FE)	ESC	Escape
FF	Form Feed (FE)	FS	File Separator (IS)
→ CR	Carriage Return (FE)	GS	Group Separator (IS)
SO	Shift Out	RS	Record Separator (IS)
SI	Shift In	US	Unit Separator (IS)
		DEL	Delete

(CC) Communication Control
(FE) Format Effector
(IS) Information Separator

Abbildung D.1

Spezielle Grafikzeichen

→ SP	Space	→ <	Less Than
→ !	Exclamation Point	→ =	Equals
→ "	Quotation Marks	→ >	Greater Than
→ #	Number Sign	→ ?	Question Mark
→ $	Dollar Sign	→ @	Commercial At
→ %	Percent	→ [	Opening Bracket
→ &	Ampersand	\	Reverse Slant
→ '	Apostrophe	→]	Closing Bracket
→ (	Opening Parenthesis	^	Circumflex
→)	Closing Parenthesis	_	Underline
→ *	Asterisk	`	Grave Accent
→ +	Plus	{	Opening Brace
→ ,	Comma	|	Vertical Line*
→ -	Hyphen (Minus)	}	Closing Brace
→ .	Period (Decimal Point)	~	Tilde
→ /	Slant		
→ :	Colon		
→ ;	Semicolon		

Mit Pfeil markierte Zeichen entsprechen PET ASCII Zeichensatz.

* (Dieses Zeichen wird manchmal stilisiert, um es von dem Zeichen für OR - logisch Oder - zu unterscheiden, das kein ASCII-Zeichen ist)

Abbildung D.2

ASCII

ASCII bedeutet American Standard Code for Information Interchange. Es handelt sich um einen Kommunikationsstandard. Dieser wird auch häufig bei Nicht-Commodore Druckern benutzt.

Befindet sich eine Commodore Maschine im Grafikmodus, entspricht sein Zeichensatz größtenteils dem ASCII. Zahlen, die großen Buchstaben und die Zeichensetzung sind gleich. Einige Kontrollzeichen, wie etwa RETURN, passen ebenfalls. Die Commodore Grafikzeichen haben keine ASCII Entsprechung.

Befindet sich eine Commodore Maschine im Textmodus, weicht der Zeichensatz deutlich von ASCII ab. Die Zahlenzeichen und die meisten Satzzeichen entsprechen zwar dem ASCII. Dagegen entsprechen die ASCII Zeichen für Großbuchstaben den Kleinbuchstaben bei den Commodore Computern. Commodore's Großbuchstaben liegen nun vollständig außerhalb des ASCII Bereichs, da es sich bei ASCII um einen Sieben-Bit Code handelt.

Als Folge davon müssen Commodore's PET ASCII Zeichen vor der Übermittlung an eine echte ASCII Einheit oder Kommunikationsleitung umgewandelt werden. Das läßt sich entweder durch ein Hardwareinterface oder durch ein Programm erreichen. Die Prozedur läuft kurz folgendermaßen ab:

1. Liegt das Commodore Zeichen unterhalb $3F, kann man es direkt zu der ASCII Einrichtung übermitteln.
2. Liegt das Commodore Zeichen zwischen $40 und $5F, muß man es vor Übermittlung zur ASCII Einrichtung durch logisch OR mit $20 verändern (oder dezimal 32 addieren).
3. Liegt das Commodore Zeichen zwischen $C0 und $DF, muß man es vor Übermittlung zur ASCII Einrichtung durch logisch AND mit $7F verändern (oder dezimal 128 subtrahieren).

Entsprechende Regeln kann man anwenden, um einem Commodore Computer den Empfang von einer ASCII Einrichtung zu ermöglichen. Für jede Übertragungsrichtung benötigen einige Kontrollzeichen eine spezielle Behandlung.

Erste Hexadezimalziffer

Zweite Hexadezimalziffer	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DCI	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	–	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	—	o	DEL

ASCII Werte

Anhang E
Übungen für unterschiedliche Commodore Maschinen

Aus Kapitel 6:

VIC-20 Version (nicht erweitert)

Wir schreiben das BASIC Programm folgendermaßen:

```
100 V%=0
110 FOR J=1 TO 5
120 INPUT "WERT";V%
130 SYS ++++
140 PRINT "MAL ZEHN = ";V%
150 NEXT J
```

Planen Sie den Anfang des Maschinenspracheprogramms bei ungefähr 4097 + 127 oder 4224 (hexadezimal 1080). Auf dieser Basis können wir Zeile 130 zu SYS 4224 verändern. Versuchen Sie noch nicht, das Programm zu starten.

```
.A 1080 LDY #$02
.A 1082 LDA ($2D),Y
.A 1084 STA $033C
.A 1087 STA $033E
.A 108A LDY #$03
.A 108C LDA ($2D),Y
.A 108E STA $033D
.A 1091 STA $033F
.A 1094 ASL $033D
.A 1097 ROL $033C
.A 109A ASL $033D
.A 109D ROL $033C
.A 10A0 CLC
.A 10A1 LDA $033D
.A 10A4 ADC $033F
.A 10A7 STA $033D
.A 10AA LDA $033C
.A 10AD ADC $033E
.A 10B0 STA $033C
.A 10B3 ASL $033D
.A 10B6 ROL $033C
.A 10B9 LDY #$02
.A 10BB LDA $033C
.A 10BE STA ($2D),Y
```

```
.A 10C0 LDY #$03
.A 10C2 LDA $033D
.A 10C5 STA ($2D),Y
.A 10C7 RTS
```

Um den Zeiger des Variablenanfangs zu verändern und ihn auf eine Speicherstelle oberhalb des Maschinenspracheprogramms zeigen zu lassen, bringe man den SOV Zeiger mit dem Befehl `.M 002D 002E` auf den Bildschirm und verändere den Zeiger zu

```
.:002D C8 10 .. .. .. .. .. ..
```

PET/CBM Version

Wir schreiben das BASIC Programm folgendermaßen:

```
100 V%=0
110 FOR J=1 TO 5
120 INPUT "WERT";V%
130 SYS ++++
140 PRINT "MAL ZEHN = ";%
150 NEXT J
```

Planen Sie den Anfang des Maschinenspracheprogramms bei ungefähr 1025 + 127 oder 1152 (hexadezimal 480). Auf dieser Basis können wir Zeile 130 zu SYS 1152 verändern. Versuchen Sie noch nicht, das Programm zu starten.

```
.A 0480 LDY #$02
.A 0482 LDA ($2A),Y
.A 0484 STA $033C
.A 0487 STA $033E
.A 048A LDY #$03
.A 048C LDA ($2A),Y
.A 048E STA $033D
.A 0491 STA $033F
.A 0494 ASL $033D
.A 0497 ROL $033C
.A 049A ASL $033D
.A 049D ROL $033C
.A 04A0 CLC
.A 04A1 LDA $033D
.A 04A4 ADC $033F
.A 04A7 STA $033D
.A 04AA LDA $033C
.A 04AD ADC $033E
.A 04B0 STA $033C
.A 04B3 ASL $033D
.A 04B6 ROL $033C
.A 04B9 LDY #$02
```

```
.A 04BB LDA $033C
.A 04BE STA ($2A),Y
.A 04C0 LDY #$03
.A 04C2 LDA $033D
.A 04C5 STA ($2A),Y
.A 04C7 RTS
```

Um den Zeiger des Variablenanfangs zu verändern und ihn auf eine Speicherstelle oberhalb des Maschinenspracheprogramms zeigen zu lassen, bringe man den SOV Zeiger mit dem Befehl .M 002A 002B auf den Bildschirm und verändere den Zeiger zu

```
.:002A C8 04 .. .. .. .. .. ..
```

Aus Kapitel 7:

Ein Interrupt-Vorhaben VIC-20 Version (nicht erweitert)

Der einzige Unterschied beim VIC-20 besteht darin, daß der Bildschirm sich bei der Speicherstelle $1E00 befindet:

```
.A 033C LDA $91
.A 033E STA $1E00
.A 0341 JMP ($03A0)
```

Um die Verbindungsadresse in $03A0/1 zu setzen:

```
.A 0344 LDA $0314
.A 0347 STA $03A0
.A 034A LDA $0315
.A 034D STA $03A1
```

Um das Programm ablaufen zu lassen:

```
.A 0350 SEI
.A 0351 LDA #$3C
.A 0353 STA $0314
.A 0356 LDA #$03
.A 0358 STA $0315
.A 035B CLI
.A 035C RTS
```

Um das ursprüngliche Interrupt rückzuspeichern:

```
.A 035D SEI
.A 035E LDA $03A0
.A 0361 STA $0314
.A 0364 LDA $03A1
```

```
.A 0367 STA $0315
.A 036A CLI
.A 036B RTS
```

SYS 836 ruft den neuen Interruptcode auf; SYS 861 schaltet ihn wieder aus. Wie beim Commodore 64 besteht die Möglichkeit, daß ein weißes Zeichen auf weißem Hintergrund gedruckt wird, das man dann nicht sehen kann.

PET/CBM Version

Diese Version ist nicht für Maschinen mit dem Original ROM geeignet, deren IRQ Zeiger bei Adresse $0219/A liegt:

```
.A 033C LDA $9B
.A 033E STA $8000
.A 0341 JMP ($03A0)
```

Um die Verbindungsadresse in $03A0/1 zu setzen:

```
.A 0344 LDA $0090
.A 0347 STA $03A0
.A 034A LDA $0091
.A 034D STA $03A1
```

Um das Programm ablaufen zu lassen:

```
.A 0350 SEI
.A 0351 LDA #$3C
.A 0353 STA $0090
.A 0356 LDA #$03
.A 0358 STA $0091
.A 035B CLI
.A 035C RTS
```

Um das ursprüngliche Interrupt rückzuspeichern:

```
.A 035D SEI
.A 035E LDA $03A0
.A 0361 STA $0090
.A 0364 LDA $03A1
.A 0367 STA $0091
.A 036A CLI
.A 036B RTS
```

SYS 836 ruft den neuen Interruptcode auf; SYS 861 schaltet ihn wieder aus. Da der PET/CBM keine Farbdarstellung ermöglicht, sind die Zeichen immer sichtbar.

Aufgabe: Hinzufügen eines Befehls

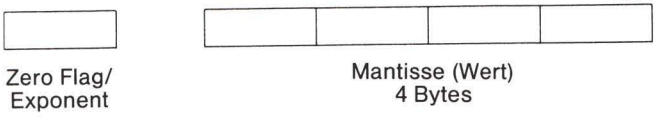
PET/CBM Version

Ein vergleichbares Programm zu schreiben, um einen Befehl hinzuzufügen, ist beim PET/CBM nicht möglich. Diese Maschine besitzt keine „Verbindungsstelle", die bei Adresse $0308/9 gehorsam auf uns wartet. Deshalb müßte ein entsprechendes Programm etwas länger und weniger elegant ausfallen.

Wir wollen ein entsprechendes Programm für den PET/CBM hier nicht anbieten. Es wäre dabei nötig, einen Teil des CHRGET Programms (bei $0070 bis $0087) zu überschreiben. Wir müßten für den Teil des Programms, den wir zerstört haben, einen Ersatz liefern und dann das Programm für den neuen Befehl hinzufügen.

Anhang F
Fließkommadarstellung

Verschachtelt: 5 Bytes (wie in Variable oder Array zu finden)

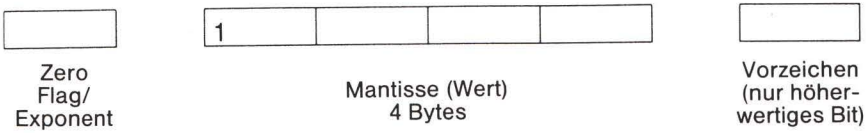


Zero Flag/ Exponent

Mantisse (Wert) 4 Bytes

Oberes Bit repräsentiert Vorzeichen der Mantisse

Unverschachtelt: 6 Bytes (wie in Fließakkumulator zu finden)



Zero Flag/ Exponent

Mantisse (Wert) 4 Bytes

Vorzeichen (nur höherwertiges Bit)

- wenn Exponent = 0, ist ganze Zahl null
- wenn Exponent > $80, wird der Dezimalpunkt um soviele Stellen nach rechts gerückt, wie der Exponent die $80 überschreitet.
- Beispiel: Exponent: $83 Mantisse: 11000000... binär setzt den Punkt um drei Positionen nach rechts: 110.000... , was einen Wert von 6 ergibt.
- wenn Exponent < = $80, ist die Zahl ein Bruch kleiner als 1.

Übung: Stellen Sie +27 in Fließkommadarstellung dar

27 dezimal = 11011 binär; Mantisse = 11011000... der Punkt muß um 5 Positionen nach rechts verschoben werden (11011.000...). Wir erhalten:

Exponent: $85 Mantisse: 11011000... binär oder D8 00 00 00 hexadezimal.

Zum Verschachteln ersetzen wir das erste Bit der Mantisse durch ein Vorzeichen-Bit (0 für positiv) und erhalten:

85 58 00 00 00

Anhang G
Rettung aus der Not

Am besten schreibt man ein Programm, das fehlerfrei ist. Nicht alle von uns sind dabei erfolgreich.

Wenn ein Programm Schwierigkeiten bereitet, sollte es durch Setzen von Unterbrechungspunkten überprüft werden. Dazu wird der Befehl BRK (break) an verschiedenen strategischen Punkten innerhalb des Programms eingefügt. Das Programm stoppt an diesen Punkten und der Programmierer erhält Gelegenheit, an ausgewählten Punkten das korrekte Verhalten des Programms zu bestätigen. Mit dieser Technik läßt sich ein Fehler eng eingrenzen.

Ab und zu stürzt ein Programm meist auf Grund schlechter Planung ab und die Ursache dieses Absturzes kann nicht identifiziert werden. Schlimmer noch, wenn ein langes Programm abstürzt und der Benutzer vergaß, zuvor eine Kopie anzulegen. Der Benutzer ist dann mit der Notwendigkeit konfrontiert, alles nocheinmal von vorne einzugeben.

In solchen Fällen gibt es eine Rettung aus der Not (uncrashing) durch Techniken, die den Computer wieder auf die Beine stellen. Diese Techniken sind selten vollkommen zufriedenstellend. Man sollte sie wirklich als letzte Rettung ansehen.

Die Technik unterscheidet sich von Computer zu Computer.

PET/CBM

PET's mit dem Original ROM lassen sich nicht retten.

Die folgenden Modelle hingegen schon, wenn auch Hardwarezusätze erforderlich sind. Der Leser sollte sich an einen in Computerhardware Erfahrenen wenden, der ihm dabei behilflich ist, den Computer mit Schaltern nachzurüsten.

Ein Wechselschalter ist notwendig, der an eine Leitung des parallelen Userport angeschlossen wird, die als „Diagnosefühler" dient. Das ist Anschluß 5 des PUP. Die andere Seite des Schalters verbindet man mit Masse (Pin 12).

Zusätzlich benötigt man einen Taster. Dieser muß die Reset-Leitung des Computers mit Masse verbinden. Technisch gesprochen ist es besser, den Eingang des Power-on-reset Bausteins des Computers (einen 555 Monovibrator) zu triggern, indem man einen Widerstand benutzt. Dadurch verhindert man die zufällige Erdung eines Schaltkreises, der gerade in Funktion ist.

Zur Rettung aus der Not setzt man den Wechselschalter auf „An" und drückt den Taster. Die Maschine kehrt in Form des Maschinensprachemonitors ins Leben zurück. Man setzt den Wechselschalter anschließend zurück. Es bleibt noch mehr zu tun.

Der Computer befindet sich noch in einem instabilen Zustand. Um das zu korrigieren, kann man zwischen zwei Dingen wählen. Man kann ins BASIC zurückkehren, indem man .X und

gleich danach den Befehl CLR eingibt. Oder man kann . ; gefolgt von der Taste RETURN eintippen.

Welche Untersuchung oder welches andere Vorhaben auch immer notwendig ist, man sollte es schnell durchführen und der Computer sollte in seinen Normalzustand zurückgesetzt werden.

VIC/Commodore 64

Sie können versuchen, die Taste RUN/STOP niedergedrückt zu halten und gleichzeitig die Taste RESTORE zu betätigen, um zu sehen, ob das die Maschine wiederbelebt. Andernfalls müssen Sie einen schwerwiegenderen RESET vornehmen.

Sie müssen sich vorstellen, daß der Computer bei einem RESET einen Speichertest vornimmt, der dessen Inhalt nicht verändert. Es gibt verschiedene käufliche Interfaces für den Kassettenport - gewöhnlich sogenannte „mother boards", die mit Reset-Schaltern ausgerüstet sind.

Wenn der Reset-Schalter betätigt wird, beginnt der Computer ganz von vorne. Der Speicher wird jedoch nicht verändert. Wenn Ihnen der Startpunkt des Maschinensprachemonitors bekannt ist, können Sie diesen durch den entsprechenden SYS Befehl in Aktion bringen.

Commodore PLUS/4

In der Nähe des Netzschalters gibt es einen Reset-Knopf. Bevor Sie diesen betätigen, halten Sie die Tasten RUN/STOP und CTRL gedrückt. Drücken Sie nun den Reset-Knopf und Sie werden sich im Maschinensprachemonitor wiederfinden.

Anhang H
Supermon Anweisungen

Bei dem Programm Supermon handelt es sich nicht um einen Monitor. Vielmehr ist es ein Monitor-Generator, der einen Maschinensprachemonitor für Sie herstellt. Dafür gibt es folgenden Grund. Supermon sucht einen geeigneten Platz im Speicher und baut dort einen MLM auf, damit dieser in den am besten dazu geeigneten Platz paßt.

Laden Sie Supermon und geben Sie RUN ein. Das Programm schreibt für Sie einen MLM und ruft ihn auf. Kehren Sie nun ins BASIC zurück und geben den Befehl NEW. Sie brauchen den MLM Generator nun nicht mehr (er hat seine Arbeit getan). Es bestünde sonst auch die Gefahr, daß zwei oder mehrere MLM's generiert würden. Schaffen Sie sich deshalb das Generatorprogramm vom Hals. Jedesmal, wenn Sie den MLM benutzen möchten, geben SYS 4 oder gegebenenfalls SYS 8 ein.

Supermon enthält die folgenden Grundbefehle:

.R - zur Anzeige (und Veränderung) von Registern.
.M - zur Anzeige (und Veränderung) des Speichers.
.S - zum Sichern des Speichers auf Disk oder Band.
.L - zum Laden von Disk oder Band.
.G - um zu einem ML Programm zu gehen.
.X - um ins BASIC zurückzukehren

Supermon enthält außerdem folgende Zusatzbefehle:

.A - zum Assemblieren
.D - zum Disassemblieren

Die meisten Versionen von Supermon (nicht jedoch die selbstgestrickte Version unten) enthalten die folgenden Befehle. Obwohl sie in diesem Buch nicht benutzt werden, sind sie dennoch brauchbar:

.F - füllt den Speicher mit festen Inhalten:

```
.F 1800 18FF 00
```

.H - durchsucht den Speicher nach einer Zeichenfolge:

```
.H 0800 1800 20 D2 FF
```

.T - transferiert einen Speicherblock an eine neue Stelle

```
.T 0800 0BFF 8000
```

Einige wenige Versionen des Supermon enthalten den Befehl .I, der eine Abarbeitung des Maschinenspracheprogramms in Einzelschritten ermöglicht.

Ein Do-It-Yourself Supermon

Wenn Sie keinen Supermon von Freunden, Händlern, Computerclubs oder auf Disk zur Verfügung haben, wird Ihnen das folgende Programm, das nur für den Commodore 64 gedacht ist, von Nutzen sein.

Geben Sie das Programm ein (Sie brauchen dazu Stunden). Vergewissern Sie sich, daß die Zeilen ab der Nummer 300 aufwärts korrekt eingegeben sind. Die DATA Zeilen darunter werden durch das Programm selbst überprüft.

Wenn Sie RUN eingeben, läuft das Programm in zwei Phasen ab. Teil 1 benötigt über zwei Minuten: er überprüft alle DATA Anweisungen nach fehlenden Zeilen und Fehlern und meldet Ihnen jedes Problem. Teil 2 läuft nur, wenn Teil 1 fehlerfrei war: er bringt das Programm dazu, daß es in sich selbst „zusammenbricht". Als Ergebnis erhält man Supermon. Nachdem dieser Programmlauf beendet ist, sichern Sie Supermon auf Disk oder Band.

Bei dem von diesem Programm generierten Supermon handelt es sich um eine einfache Version (um Ihre Finger zu schonen). Es enthält jedoch alle für dieses Buch notwendigen Befehle.

```
1  DATA 26,8,100,0,153,34,147,18,29,29,-30
2  DATA 29,29,83,85,80,69,82,32,54,52,-16
3  DATA 45,77,79,78,0,49,8,110,0,153,-39
4  DATA 34,17,32,32,32,32,32,32,32,32,-50
5  DATA 32,32,32,32,32,32,32,0,75,8,-3
6  DATA 120,0,153,34,17,32,46,46,74,73,-48
7  DATA 77,32,66,85,84,84,69,82,70,73,-56
8  DATA 69,76,68,0,102,8,130,0,158,40,-4
9  DATA 194,40,52,51,41,170,50,53,54,172,-53
10  DATA 194,40,52,52,41,170,49,50,55,41,-25
11  DATA 0,0,0,170,170,170,170,170,170,170,-64
12  DATA 170,170,170,170,170,170,170,170,170,170,-29
13  DATA 170,170,170,170,170,170,170,165,45,133,-61
14  DATA 34,165,46,133,35,165,55,133,36,165,-12
15  DATA 56,133,37,160,0,165,34,208,2,198,-55
16  DATA 35,198,34,177,34,208,60,165,34,208,-34
17  DATA 2,198,35,198,34,177,34,240,33,133,-52
18  DATA 38,165,34,208,2,198,35,198,34,177,-60
19  DATA 34,24,101,36,170,165,38,101,37,72,-56
20  DATA 165,55,208,2,198,56,198,55,104,145,-1
21  DATA 55,138,72,165,55,208,2,198,56,198,-1
22  DATA 55,104,145,55,24,144,182,201,79,208,-48
23  DATA 237,165,55,133,51,165,56,133,52,108,-17
24  DATA 55,0,79,79,79,79,173,230,255,0,-22
25  DATA 141,22,3,173,231,255,0,141,23,3,-64
```

```
26 DATA 169,128,32,144,255,0,0,216,104,141,-30
27 DATA 62,2,104,141,61,2,104,141,60,2,-41
28 DATA 104,141,59,2,104,170,104,168,56,138,-17
29 DATA 233,2,141,58,2,152,233,0,0,141,-12
30 DATA 57,2,186,142,63,2,32,147,253,0,-57
31 DATA 162,66,169,42,32,205,251,0,169,82,-62
32 DATA 208,42,230,193,208,6,230,194,208,2,-52
33 DATA 230,38,96,32,207,255,201,13,208,248,-24
34 DATA 104,104,169,0,0,133,38,162,13,169,-11
35 DATA 46,32,205,251,0,32,220,249,0,201,-32
36 DATA 46,240,249,201,32,240,245,162,14,221,-23
37 DATA 195,255,0,208,12,138,10,170,189,207,-36
38 DATA 255,0,72,189,206,255,0,72,96,202,-2
39 DATA 16,236,76,80,252,0,165,193,141,58,-29
40 DATA 2,165,194,141,57,2,96,169,8,133,-44
41 DATA 29,160,0,0,32,143,253,0,177,193,-31
42 DATA 32,190,251,0,32,209,249,0,198,29,-61
43 DATA 208,241,96,32,254,251,0,144,11,162,-53
44 DATA 0,0,129,193,193,193,240,3,76,80,-58
45 DATA 252,0,32,209,249,0,198,29,96,169,-56
46 DATA 59,133,193,169,2,133,194,169,5,96,-20
47 DATA 152,72,32,147,253,0,104,162,46,76,-44
48 DATA 205,251,0,162,0,0,189,234,255,0,-31
49 DATA 32,210,255,232,224,22,208,245,160,59,-51
50 DATA 32,86,250,0,173,57,2,32,190,251,-4
51 DATA 0,173,58,2,32,190,251,0,32,75,-31
52 DATA 250,0,32,33,250,0,240,87,32,220,-13
53 DATA 249,0,32,239,251,0,144,46,32,223,-40
54 DATA 251,0,32,220,249,0,32,239,251,0,-51
55 DATA 144,35,32,223,251,0,32,225,255,240,-33
56 DATA 60,166,38,208,56,165,195,197,193,165,-22
57 DATA 196,229,194,144,46,160,58,32,86,250,-21
58 DATA 0,32,183,251,0,32,31,250,0,240,-60
59 DATA 224,76,80,252,0,32,239,251,0,144,-42
60 DATA 3,32,20,250,0,32,75,250,0,208,-43
61 DATA 7,32,239,251,0,144,235,169,8,133,-28
62 DATA 29,32,220,249,0,32,53,250,0,208,-18
63 DATA 248,76,229,249,0,32,207,255,201,13,-22
64 DATA 240,12,201,32,208,209,32,239,251,0,-57
65 DATA 144,3,32,20,250,0,174,63,2,154,-46
66 DATA 120,173,57,2,72,173,58,2,72,173,-35
67 DATA 59,2,72,173,60,2,174,61,2,172,-55
68 DATA 62,2,64,174,63,2,154,108,2,160,-56
69 DATA 160,1,132,186,132,185,136,132,183,132,-27
```

```
70  DATA 144,132,147,169,64,133,187,169,2,133,-19
71  DATA 188,32,207,255,201,32,240,249,201,13,-42
72  DATA 240,56,201,34,208,20,32,207,255,201,-35
73  DATA 34,240,16,201,13,240,41,145,187,230,-39
74  DATA 183,200,192,16,208,236,76,80,252,0,-18
75  DATA 32,207,255,201,13,240,22,201,44,208,-51
76  DATA 220,32,254,251,0,41,15,240,233,201,-46
77  DATA 3,240,229,133,186,32,207,255,201,13,-45
78  DATA 96,108,48,3,108,50,3,32,22,251,-60
79  DATA 0,208,212,169,0,0,32,111,251,0,-37
80  DATA 165,144,41,16,208,201,76,229,249,0,-22
81  DATA 32,22,251,0,201,44,208,191,32,239,-48
82  DATA 251,0,32,223,251,0,32,207,255,201,-25
83  DATA 44,208,178,32,239,251,0,165,193,133,-7
84  DATA 174,165,194,133,175,32,223,251,0,32,-34
85  DATA 207,255,201,13,208,157,32,114,251,0,-36
86  DATA 76,229,249,0,165,194,32,190,251,0,-39
87  DATA 165,193,72,74,74,74,74,32,214,251,-13
88  DATA 0,170,104,41,15,32,214,251,0,72,-16
89  DATA 138,32,210,255,104,76,210,255,9,48,-9
90  DATA 201,58,144,2,105,6,96,162,2,181,-30
91  DATA 192,72,181,194,149,192,104,149,194,202,-25
92  DATA 208,243,96,32,254,251,0,144,2,133,-30
93  DATA 194,32,254,251,0,144,2,133,193,96,-43
94  DATA 169,0,0,133,42,32,220,249,0,201,-39
95  DATA 32,208,9,32,220,249,0,201,32,208,-25
96  DATA 14,24,96,32,37,252,0,10,10,10,-62
97  DATA 10,133,42,32,220,249,0,32,37,252,-39
98  DATA 0,5,42,56,96,201,58,144,2,105,-26
99  DATA 8,41,15,96,96,32,220,249,0,201,-62
100  DATA 32,240,249,96,169,0,0,141,0,0,-22
101  DATA 1,32,47,252,0,32,5,252,0,32,-29
102  DATA 242,251,0,144,9,96,32,220,249,0,-28
103  DATA 32,239,251,0,176,222,174,63,2,154,-35
104  DATA 169,63,32,210,255,76,229,249,0,32,-48
105  DATA 143,253,0,202,208,250,96,165,195,164,-12
106  DATA 196,56,233,2,176,1,136,56,229,193,-61
107  DATA 133,30,152,229,194,168,5,30,96,32,-41
108  DATA 55,252,0,133,32,165,194,133,33,162,-22
109  DATA 0,0,134,40,169,147,32,210,255,169,-32
110  DATA 22,133,29,32,165,252,0,32,5,253,-2
111  DATA 0,133,193,132,194,198,29,208,242,169,-16
112  DATA 145,32,210,255,76,229,249,0,160,44,-41
113  DATA 32,86,250,0,32,143,253,0,32,183,-23
```

```
114 DATA 251,0,32,143,253,0,162,0,0,161,-25
115 DATA 193,32,20,253,0,72,32,90,253,0,-25
116 DATA 104,32,112,253,0,162,6,224,3,208,-43
117 DATA 18,164,31,240,14,165,42,201,232,177,-10
118 DATA 193,176,28,32,253,252,0,136,208,242,-15
119 DATA 6,42,144,14,189,54,255,0,32,187,-3
120 DATA 253,0,189,60,255,0,240,3,32,187,-24
121 DATA 253,0,202,208,213,96,32,8,253,0,-45
122 DATA 170,232,208,1,200,152,32,253,252,0,-39
123 DATA 138,134,28,32,190,251,0,166,28,96,-47
124 DATA 165,31,56,164,194,170,16,1,136,101,-53
125 DATA 193,144,1,200,96,168,74,144,11,74,-21
126 DATA 176,23,201,34,240,19,41,7,9,128,-63
127 DATA 74,170,189,229,254,0,176,4,74,74,-52
128 DATA 74,74,41,15,208,4,160,128,169,0,-20
129 DATA 0,170,189,41,255,0,133,42,41,3,-62
130 DATA 133,31,152,41,143,170,152,160,3,224,-36
131 DATA 138,240,11,74,144,8,74,74,9,32,-6
132 DATA 136,208,250,200,136,208,242,96,177,193,-29
133 DATA 32,253,252,0,162,1,32,92,252,0,-16
134 DATA 196,31,200,144,241,162,3,192,4,144,-18
135 DATA 242,96,168,185,67,255,0,133,40,185,-13
136 DATA 131,255,0,133,41,169,0,0,160,5,-3
137 DATA 6,41,38,40,42,136,208,248,105,63,-27
138 DATA 32,210,255,202,208,236,169,32,208,11,-16
139 DATA 169,13,36,19,16,5,32,210,255,169,-30
140 DATA 10,76,210,255,32,55,252,0,169,3,-21
141 DATA 133,29,32,220,249,0,32,53,250,0,-42
142 DATA 208,248,165,32,133,193,165,33,133,194,-43
143 DATA 76,134,252,0,197,40,240,3,32,210,-60
144 DATA 255,96,32,55,252,0,32,223,251,0,-57
145 DATA 142,17,2,162,3,32,47,252,0,72,-43
146 DATA 202,208,249,162,3,104,56,233,63,160,-37
147 DATA 5,74,110,17,2,110,16,2,136,208,-16
148 DATA 246,202,208,237,162,2,32,207,255,201,-31
149 DATA 13,240,30,201,32,240,245,32,220,254,-9
150 DATA 0,176,15,32,18,252,0,164,193,132,-9
151 DATA 194,133,193,169,48,157,16,2,232,157,-47
152 DATA 16,2,232,208,219,134,40,162,0,0,-10
153 DATA 134,38,240,4,230,38,240,117,162,0,-9
154 DATA 0,134,29,165,38,32,20,253,0,166,-48
155 DATA 42,134,41,170,188,67,255,0,189,131,-47
156 DATA 255,0,32,197,254,0,208,227,162,6,-54
157 DATA 224,3,208,25,164,31,240,21,165,42,-63
```

```
158 DATA 201,232,169,48,176,33,32,203,254,0,-39
159 DATA 208,204,32,205,254,0,208,199,136,208,-28
160 DATA 235,6,42,144,11,188,60,255,0,189,-15
161 DATA 54,255,0,32,197,254,0,208,181,202,-1
162 DATA 208,209,240,10,32,196,254,0,208,171,-51
163 DATA 32,196,254,0,208,166,165,40,197,29,-15
164 DATA 208,160,32,223,251,0,164,31,240,40,-6
165 DATA 165,41,201,157,208,26,32,99,252,0,-35
166 DATA 144,10,152,208,4,165,30,16,10,76,-40
167 DATA 80,252,0,200,208,250,165,30,16,246,-9
168 DATA 164,31,208,3,185,194,0,0,145,193,-62
169 DATA 136,208,248,165,38,145,193,32,5,253,-41
170 DATA 0,133,193,132,194,160,65,32,86,250,-34
171 DATA 0,32,143,253,0,32,183,251,0,32,-56
172 DATA 143,253,0,76,198,253,0,168,32,203,-29
173 DATA 254,0,208,17,152,240,14,134,28,166,-63
174 DATA 29,221,16,2,8,232,134,29,166,28,-60
175 DATA 40,96,201,48,144,3,201,71,96,56,-30
176 DATA 96,64,2,69,3,208,8,64,9,48,-14
177 DATA 34,69,51,208,8,64,9,64,2,69,-50
178 DATA 51,208,8,64,9,64,2,69,179,208,-47
179 DATA 8,64,9,0,0,34,68,51,208,140,-18
180 DATA 68,0,0,17,34,68,51,208,140,68,-5
181 DATA 154,16,34,68,51,208,8,64,9,16,-20
182 DATA 34,68,51,208,8,64,9,98,19,120,-62
183 DATA 169,0,0,33,129,130,0,0,0,0,-41
184 DATA 89,77,145,146,134,74,133,157,44,41,-39
185 DATA 44,35,40,36,89,0,0,88,36,36,-22
186 DATA 0,0,28,138,28,35,93,139,27,161,-10
187 DATA 157,138,29,35,157,139,29,161,0,0,-9
188 DATA 41,25,174,105,168,25,35,36,83,27,-64
189 DATA 35,36,83,25,161,0,0,26,91,91,-24
190 DATA 165,105,36,36,174,174,168,173,41,0,-3
191 DATA 0,124,0,0,21,156,109,156,165,105,-20
192 DATA 41,83,132,19,52,17,165,105,35,160,-26
193 DATA 216,98,90,72,38,98,148,136,84,68,-20
194 DATA 200,84,104,68,232,148,0,0,180,8,-31
195 DATA 132,116,180,40,110,116,244,204,74,114,-32
196 DATA 242,164,138,0,0,170,162,162,116,116,-11
197 DATA 116,114,68,104,178,50,178,0,0,34,-30
198 DATA 0,0,26,26,38,38,114,114,136,200,-27
199 DATA 196,202,38,72,68,68,162,200,58,59,-35
200 DATA 82,77,71,88,76,83,68,44,65,204,-59
201 DATA 250,0,191,250,0,96,250,0,134,250,-25
```

```
202 DATA 0,224,250,0,14,251,0,116,251,0,-23
203 DATA 135,251,0,120,252,0,160,253,0,194,-19
204 DATA 253,0,228,249,0,157,249,0,139,249,-63
205 DATA 0,13,32,32,32,80,67,32,32,83,-3
206 DATA 82,32,65,67,32,88,82,32,89,82,-16
207 DATA 32,83,80,-59
255 DATA 208
300 M=63
310 READ X:L=PEEK(M):H=L=255:IF H THEN L=X
320 V=R<>L:S=(T<>63 AND R>0 AND V)
330 IF V THEN T=L:IF NOT S THEN R=R+1:S=R<>L
340 T=(T*3+X)AND63
350 IF S THEN PRINT "ERROR LINE";R:E=-1
360 R=L:IF NOT H GOTO 310
370 IF E THEN STOP
380 PRINT"HERE WE GO":X=-1:RESTORE:B=2049:FOR A=1
    TO 9999
390 IF X>=0 THEN POKE B,X:B=B+1
400 READ X:L=PEEK(M):IF L<255 THEN NEXT A
410 POKE 45,16:POKE 46,16:CLR
```

Anhang I
Informationen zu IA-Bausteinen

Das folgende Material wurde aus Herstellerspezifikationen übernommen. Die Information ist für Maschinenspracheprogrammierung nicht von grundlegender Bedeutung, kann jedoch eine große Hilfe bei weiteren Untersuchungen darstellen. Einige dieser Spezifikationen sind nicht weit verbreitet und enthalten deshalb Informationen, an die man nur schwer gelangen kann.

6520 PIA	peripheral interface adaptor
6522 VIA	versatile interface adaptor
6525 TPA	tri port adaptor
6526 CIA	complex interface adaptor
6545 CRTC	cathode ray tube (CRT) controller
6560 VIC	video interface chip
6566 VIC-2	video interface chip
6581 SID	sound interface chip

(Grundlegende Herstellerspezifikationen, weniger Hardwaredetails)

6520 Peripheral Interface Adaptor (PIA)

Beim 6520 handelt es sich um eine I/O (Eingabe/Ausgabe) Einheit, die als Schnittstelle zwischen dem Mikroprozessor und der Peripherie wie etwa Drucker, Anzeigen, Tastaturen usw. wirkt. Die Hauptfunktion des 6520 besteht darin, auf Signale von jeder der zwei Welten, die sie bedient, zu antworten. Auf der einen Seite stellt der 6520 eine Verbindung zur Peripherie über einen acht Bit breiten bidirektionalen peripheren Datenport dar. Auf der anderen Seite verbindet diese Einheit über einen acht Bit Datenbus mit dem Mikroprozessor. Zusätzlich zu den oben beschriebenen Leitungen besitzt der 6520 vier Eingabe/Peripherie Interrupt Kontroll-Leitungen und die notwendige Logik zur einfachen und effektiven Kontrolle peripherer Interrupts.

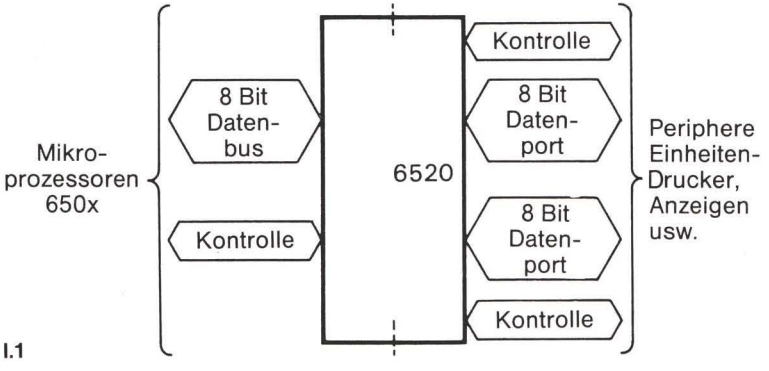


Abbildung I.1

Die funktionelle Konfiguration des 6520 wird während der Initialisierung des Systems durch den Mikroprozessor programmiert. Jede der peripheren Datenleitungen wird so programmiert, daß sie als Eingang oder Ausgang arbeiten kann. Jede der vier Kontroll/Interrupt Leitungen kann für eine von vier möglichen Kontrollarten programmiert werden. Dadurch wird ein hohes Maß an Flexibilität in der gesamten Operationsweise des Interface ermöglicht.

Dateneingangsregister

Wenn der Mikroprozessor Daten in den 6520 schreibt, werden die Daten, die auf dem Datenbus erscheinen, im Dateneingangsregister zwischengespeichert (latching). Anschließend werden sie in eines von sechs internen Registern des 6520 übertragen. Dadurch wird sichergestellt, daß die Daten auf den peripheren Ausgangsleitungen nicht „entgleisen". Es bedeutet, auch wenn die Ausgangsleitungen sanfte Übergänge von hohen zu tiefen und von tiefen zu hohen Spannungswerten machen, bleibt die Spannung solange stabil, bis sie zur entgegengesetzten Polarität umschaltet.

Kontroll-Register (CRA und CAB)

In Abbildung I.2 werden Zuordnung und Funktion der Bits in den Kontroll-Registern gezeigt. Mit den Kontroll-Registern kann der Mikroprozessor die Arbeitsweise der Unterbrechungsleitungen (CA1, CA2, CB1 und CB2) und peripheren Kontrolleitungen (CA2, CB2) beeinflussen. Ein einzelnes Bit in jedem Register kontrolliert die Adressierung der Datenrichtungs-Register (DDRA, DDRB) und der Ausgabe-Register (ORA, ORB), die später behandelt werden. Darüberhinaus stehen zwei Bits (Bit 6 und 7) in jedem Kontroll-Register zur Verfügung, die den Status der Interrupt Eingangsleitungen (CA1, CA2 , CB1 und CB2) anzeigen. Diese Unterbrechungsstatus Bits ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$) werden normalerweise vom Mikroprozessor während der Interrupt Serviceprogramme abgefragt, um die Quelle eines aktiven Interrupt festzustellen. Dies sind die Interruptleitungen, die die Interrupteingänge ($\overline{\text{IRQ}}$, $\overline{\text{NMI}}$) des Mikroprozessors treiben. Die anderen Bits in CRA und CRB werden dann behandelt, wenn wir uns mit der Schnittstelle zu peripheren Einheiten beschäftigen.

Die verschiedenen Bits in den Kontroll-Registern werden während eines Programmlaufs häufig angesprochen, um dem Prozessor Interrupts möglich oder unmöglich zu machen, um Betriebszustände zu verändern usw. je nach dem, wie die periphere Einheit, die kontrolliert wird, es erfordert.

Datenrichtungs-Register (DDRA, DDRB)

Die Datenrichtungs-Register ermöglichen es dem Prozessor, jede Leitung des 8-Bit peripheren I/O Kanals so zu programmieren, daß sie entweder als Eingang oder als Ausgang fungiert. Jedes Bit in DDRA kontrolliert die entsprechende Leitung im peripheren A Kanal und jedes Bit in DDRB kontrolliert die entsprechende Leitung im peripheren B Kanal. Setzt man eine „0" in das Datenrichtungs-Register, arbeitet die entsprechende periphere I/O Leitung als Eingang. Eine „1" macht sie zum Ausgang.

Die Datenrichtungs-Register werden normalerweise nur während der Systeminitialisierung programmiert, die als Antwort auf ein Reset Signal ausgeführt wird. Die Inhalte dieser Register können allerdings während des Systembetriebs verändert werden. Dadurch wird eine sehr bequeme Kontrolle peripherer Einheiten wie z.B. der Tastatur ermöglicht.

Periphere Ausgangsregister (ORA, ORB)

Die peripheren Ausgangsregister speichern die Ausgangsdaten, die am peripheren I/O Kanal anstehen. Schreibt man eine „0" in ein Bit von ORA, führt das dazu, daß die Spannung der entsprechenden Leitung des peripheren A Kanals nach unten geht (⟨0.4 V), wenn diese Leitung als Ausgang programmiert ist. Eine „1" bringt die Spannung des entsprechenden Ausgangs nach oben. Die Leitungen des peripheren B Kanals werden auf gleiche Weise durch ORB kontrolliert.

Interrupt Statuskontrolle

Die vier Interrupt/Peripherie-Kontrolleitungen (CA1, CA2, CB1 und CB2) werden durch die Interrupt – Statuskontrolle (A, B) überwacht. Diese Logik interpretiert die Inhalte der entsprechenden Kontroll-Register, stellt aktive Übergänge bei den Interrupteingängen fest und führt solche Operationen aus, die notwendig sind, um eine ordnungsgemäße Arbeitsweise dieser vier peripheren Interfaceleitungen zu gewährleisten.

Reset ($\overline{\text{RES}}$)

Die Reset-Leitung, die im aktiven Zustand auf niedriger Spannung liegt, setzt die Inhalte aller 6520 Register auf logisch null. Diese Leitung kann entweder beim Einschalten der Netzspannung zum Rücksetzen genutzt werden oder zum Rücksetzen, während das System arbeitet.

Interrupt Anforderungsleitung ($\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$)

Die Interrupt Anforderungsleitungen ($\overline{\text{IRQA}}$ und $\overline{\text{IRQB}}$), die im aktiven Zustand auf niedriger Spannung liegen, können ein Interrupt des Mikroprozessors herbeiführen. Das geschieht entweder direkt oder durch einen externen Schaltkreis, der ein vorrangiges Interrupt auslöst.

Jede Interrupt Anforderungsleitung ist mit zwei Interruptflag Bits verknüpft, die die Spannung auf den Interrupt Anforderungsleitungen nach unten bringen können. Bei diesen Flags handelt es sich um Bit 6 und 7 in den beiden Kontroll-Registern. Diese Flags wirken als Verbindung zwischen den peripheren Interruptsignalen und den Interrupteingängen des Mikroprozessors. Jede Flag hat ein entsprechendes Bit (interrupt disable bit), das ein Interrupt für den Mikroprozessor durch jede der vier Interrupteingänge (CA1, CA2, CB1 und CB2) möglich oder unmöglich macht.

Die vier Interruptflags werden durch aktive Signalübergänge an den Interrupteingängen (CA1, CA2, CB1, CB2) gesetzt. Die Kontrolle dieser aktiven Übergänge wird im nächsten Abschnitt behandelt.

Kontrolle von $\overline{\text{IRQA}}$

Bit 7 des Kontroll-Registers A wird immer durch einen aktiven Übergang des CA1 Interrupteingangssignal gesetzt. Ein Interrupt durch diese Flag kann dadurch gesperrt werden, daß Bit 0 im Kontroll-Register A (CRA) auf logisch 0 gesetzt wird. Entsprechend kann Bit 6 des Kontroll-Registers A durch einen aktiven Übergang des CA2 Interrupteingangssignals gesetzt werden. Ein Interrupt durch diese Flag kann durch Setzen von Bit 3 im Kontroll-Register auf logisch 0 gesperrt werden.

Beide, Bit 6 und Bit 7 in CRA werden durch eine Operation zurückgesetzt, die man als „Lies das periphere Ausgangsregister A" bezeichnen kann. Damit ist eine Operation gemeint, bei der der Prozessor den peripheren I/O Kanal A liest.

Kontrolle von $\overline{\text{IRQB}}$

Die Kontrolle von $\overline{\text{IRQB}}$ erfolgt auf exakt die gleiche Weise wie oben bei $\overline{\text{IRQA}}$ beschrieben. Bit 7 in CRB wird durch einen aktiven Übergang an CB1 gesetzt. Die Unterbrechung durch diese Flag wird durch CRB Bit 0 kontrolliert. Bit 6 in CRB wird ähnlich durch einen aktiven Übergang an CB2 gesetzt. Die Unterbrechung durch diese Flag wird durch CRB Bit 3 kontrolliert.

Genauso werden die beiden Bit 6 und 7 durch eine Operation des „Lies das periphere Ausgangsregister B" zurückgesetzt.

Zusammenfassung

IRQA geht nach unten, wenn CRA-7 = 1 und CRA-0 = 1 oder wenn CRA-6 = 1 und CRA-3 = 1.

IRQB geht nach unten, wenn CRB-7 = 1 und CRB-0 = 1 oder wenn CRB-6 = 1 und CRB-3 = 1.

An dieser Stelle sollte besonders darauf hingewiesen werden, daß die Flags als Verbindung zwischen dem peripheren Interruptsignal und den Interrupteingängen des Prozessors wirken. Die „interrupt disable bits" ermöglichen dem Prozessor die Kontrolle der Interruptfunktion.

Periphere I/O Kanäle

Jede der peripheren I/O Leitungen kann so programmiert werden, daß sie entweder als Eingang oder als Ausgang arbeiten. Das wird dadurch erreicht, daß man eine „1" in die entsprechenden Bits des Datenrichtungs-Registers für diejenigen Leitungen setzt, die als Ausgänge geschaltet werden sollen. Eine „0" in einem Bit des Datenrichtungs-Registers schaltet die entsprechenden peripheren I/O Leitungen als Eingang.

Interrupteingangs/Periphere Kontrolleitungen (CA1, CA2, CB1, CB2)

Diese vier Kontrolleitungen ermöglichen eine Reihe spezieller Peripherie-Kontrollfunktionen. Sie verbessern die Möglichkeiten der beiden allgemein verwendbaren Interface Kanäle (PA0-PA7, PB0-PB7) um ein Vielfaches.

Interrupteingangs/Periphere Kontrolleitungen der Peripherie A (CA1, CA2)

CA1 wirkt nur als Interrupteingang. Ein aktiver Signalübergang an diesem Eingang setzt Bit 7 des Kontroll-Registers A auf logisch 1. Der aktive Übergang kann durch den Mikroprozessor dadurch programmiert werden, daß er eine „0" in Bit 1 des CRA setzt, wenn die Interruptflag (Bit 7 von CRA) durch die negative Flanke des CA1 Signals gesetzt werden soll, oder dadurch, daß er eine „1" für einen positiven Übergang setzt.

Ein Setzen der Interruptflag unterbricht den Prozessor durch IRQA dann, wenn Bit 0 von CRA auf 1 ist, wie wir vorher schon beschrieben haben.

CA2 kann als völlig unabhängiger Interrupteingang oder als ein Peripherie-Kontrollausgang wirken. Als Eingang (CRA, Bit 5 = 0) besteht seine Wirkung darin, daß er die Interruptflag, Bit 6 von CRA, beim aktiven Übergang, der durch Bit 4 von CRA ausgewählt wird, auf logisch 1 setzt.

Diese Kontroll-Register Bits und Interrupteingänge bieten dieselben Grundfunktionen wie die, die oben für CA1 beschrieben wurden. Das Eingangssignal setzt die Interruptflag, die als Verbindung zwischen der Peripherieeinheit und der Interruptstruktur des Prozessors dient. Das „interrupt disable bit" ermöglicht dem Prozessor eine Kontrolle der Systeminterrupts.

Im Ausgangsmodus (CRA, Bit 5 = 1) kann CA2 unabhängig arbeiten, um jedesmal, wenn der Mikroprozessor die Daten am peripheren I/O Kanal A liest, einen einfachen Puls zu erzeugen. Dieser Modus wird dadurch ausgewählt, daß Bit 4 des CRA auf „0" und Bit 3 des CRA auf „1" gesetzt wird. Diese Pulsausgabe kann dazu genutzt werden, um Zähler, Schieberegister usw. zu kontrollieren, die sequentielle Daten auf den peripheren Eingabeleitungen zur Verfügung stellen.

Ein zweiter Ausgangsmodus erlaubt die Benutzung von CA2 in Verbindung mit CA1. Dadurch ist der Austausch eines Quittungssignals (handshake) zwischen dem Prozessor und der Peripherieeinheit möglich. Auf der Seite A ermöglicht diese Technik eine positive Kontrolle der Datenübertragung von der Peripherie zum Mikroprozessor. Der CA1 Eingang signalisiert dem Prozessor durch Prozessorinterrupt, daß Daten zur Verfügung stehen. Der Prozessor liest die Daten und setzt CA2 zurück. Das zeigt der Peripherieeinheit, daß sie neue Daten zur Verfügung stellen kann.

Der letzte Ausgangsmodus kann durch Setzen von Bit 4 des CRA auf „1" ausgewählt werden. In diesem Modus wirkt CA2 als einfacher Peripherie Kontrollausgang, der durch Setzen von Bit 3 des CRA auf „1" oder „0" hoch bzw. niedrig geschaltet werden kann.

Interrupteingangs/Periphere Kontrolleitungen der Peripherie B (CB1, CB2)

CB1 arbeitet nur als Interrupteingang, jedoch auf die gleiche Weise wie CA1. Bit 7 von CRB wird durch einen aktiven Übergang gesetzt, der durch Bit 0 des CRB ausgewählt wurde. Der Eingangsmodus von CB2 arbeitet genauso, wie der Eingangsmodus von CA2. Die Ausgangsmodi von CB2, CRB, Bit 5 = 1, unterscheiden sich ein wenig von denen des CA2. Die Pulsausgabe erfolgt, wenn der Prozessor Daten in das Peripherie Ausgangsregister B schreibt. Auch die Quittierung der Datenübertragung vom Prozessor zur Peripherieeinheit funktioniert ähnlich.

	7	6	5 4 3	2	1 Ø
CRA	IRQA1	IRQA2	CA2 CONTROL	DDRA ACCESS	CA1 CONTROL

	7	6	5 4 3	2	1 Ø
CRB	IRQB1	IRQB2	CB2 CONTROL	DDRB ACCESS	CB2 CONTROL

Abbildung I.2

Der 6545-1 CRT Controller (CRTC)

Konzept

Der 6545-1 ist ein Kathodenstrahlröhren-Kontroller, der dazu entwickelt wurde, die Familie der 6500 Mikroprozessoren an CRT oder TV ähnliche Rasterabtastanzeigen anzuschließen.

Horizontal total (R0)

Dieses 8-Bit Register beinhaltet die Gesamtzahl der pro horizontaler Zeile angezeigten und nichtangezeigten Zeichen minus 1. Die Frequenz von HSYNC wird daher durch dieses Register bestimmt.

Horizontal angezeigt (R1)

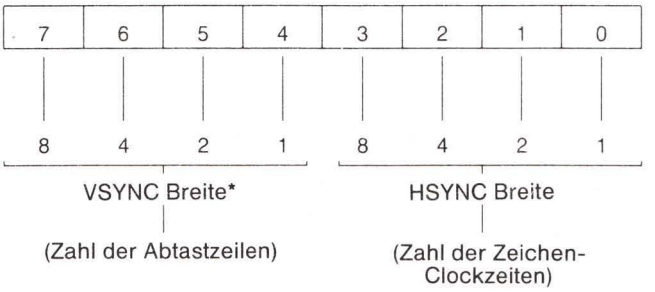
Dieses 8-Bit Register enthält die pro horizontaler Zeile angezeigten Zeichen.

Horizontale Sync Position (R2)

Dieses Register enthält die Position von HSYNC auf der horizontalen Zeile. Diese wird als Zahlenwert für die Zeichenstelle auf der Zeile angegeben. Die Position von HSYNC bestimmt die links-rechts Ausrichtung des auf dem Videobildschirm angezeigten Textes. Auf diese Weise werden die seitlichen Begrenzungen justiert.

Horizontale und Vertikale Sync Breite (R3)

Dieses 8-Bit Register enthält sowohl die HSYNC als auch VSYNC Breite in folgender Weise:



* Wenn Bits 4-7 alle "0", ist VSYNC 16 Abtastzeilen breit

Durch Kontrolle dieser Parameter läßt sich der 6545-1 an viele CRT Monitore anschließen, da die HSYNC und VSYNC Zeitgebersignale ohne ein externes Triggersignal eingestellt werden können.

Vertikal total (R4)

Beim Register „Vertikal total" handelt es sich um ein 7-Bit Register, das die Gesamtzahl von Zeichenzeilen minus 1 in einem Bildrahmen enthält. Dieses Register bestimmt zusammen mit R5 die gesamte Rahmenrate, die nahe bei der Zeilenfrequenz liegen sollte, um ein flimmerfreies Bild zu gewährleisten. Wenn die Rahmenzeit länger als die Periode für die Zeilenfrequenz eingestellt ist, kann man $\overline{RES}$ zur absoluten Synchronisierung benutzen.

Vertikale totale Justierung (R5)

Das Justierungsregister für Vertikal total ist ein 5-Bit Register, das nur beschrieben werden kann und das die Anzahl zusätzlicher Abtastzeilen enthält, die nötig sind, um einen gesamten Abtastvorgang für einen Bildrahmen zu vervollständigen. Damit läßt sich eine Feinjustierung der Bildrahmenzeit erreichen.

Vertikal Angezeigt (R6)

Dieses 7-Bit Register enthält die Zahl der in jedem Rahmen angezeigten Zeichenzeilen. Damit wird die vertikale Größe des angezeigten Textes festgelegt.

Reg. No.	Register Name	gespeicherte Info	RD	WR	Register Bit 7	6	5	4	3	2	1	0
R0	Horiz. total	# Zeichen		√	■	■	■	■	■	■	■	■
R1	Horiz. angezeigt	# Zeichen		√	■	■	■	■	■	■	■	■
R2	Horiz. Sync Position	# Zeichen		√	■	■	■	■	■	■	■	■
R3	VSYNC, HSYNC Breiten	# Scan Linien & # Zeichen Zahl		√	V_3	V_2	V_1	V_0	H_3	H_2	H_1	H_0
R4	Vert. Total	# Zeichen Zeile		√	X	■	■	■	■	■	■	■
R5	Vert. Total just.	# Scan Linien		√	X	X	X	■	■	■	■	■
R6	Vert. angezeigt	# Zeichen Zeilen		√	X	■	■	■	■	■	■	■
R7	Vert. Sync Position	# Zeichen Zeilen		√	X	■	■	■	■	■	■	■
R8	Mode Control			√	■	■	■	■	■	■	■	■
R9	Scan Linie	# Scan Linien		√	X	X	X	■	■	■	■	■
R10	Cursor Start	Scan Linie No.		√	X	B_1	B_0	■	■	■	■	■
R11	Cursor Ende	Scan Linie No.		√	X	X	X	■	■	■	■	■
R12	Display Start Addr (H)			√	X	X	■	■	■	■	■	■
R13	Display Start Addr (L)			√	■	■	■	■	■	■	■	■
R14	Cursor Position (H)		√	√	X	X	■	■	■	■	■	■
R15	Cursor Position (L)		√	√	■	■	■	■	■	■	■	■
R16	Lichtstift Reg. (H)		√		X	X	■	■	■	■	■	■
R17	Lichtstift Reg. (L)		√		■	■	■	■	■	■	■	■

Bemerkungen:

■ Bezeichnet binäres Bit

X Bezeichnet unbenutztes Bit. Beim Lesen ergibt dieses Bit immer „0“, außer für R31, das den Datenbus nicht bedient, und für CS „1“, das ähnlich arbeitet.

Vertikale Sync Position (R7)

Dieses 7-Bit Register wird dazu benutzt, den Zeitpunkt festzulegen, bei dem der VSYNC Puls einer Zeichenzeile auftreten soll. Das bedeutet, damit wird die Position in vertikaler Richtung für den angezeigten Text festgelegt.

Modus Kontrolle (R8)

Dieses Register bestimmt die Betriebsart des 6545-1 und ist folgendermaßen angelegt:

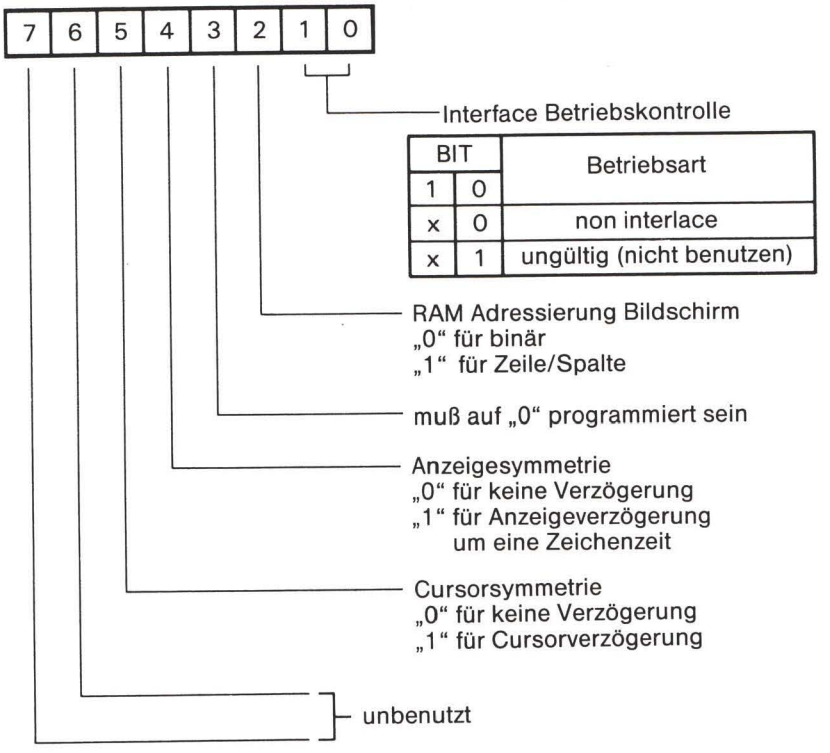


Abbildung I.3

Abtastzeilen (R9)

Dieses 5-Bit Register enthält die Anzahl Abtastzeilen pro Zeichenzeile, Zwischenräume eingeschlossen.

Cursor Anfang (R10) und Cursor Ende (R11)

Dieses 5-Bit Register bestimmt die Abtastanfangs- und Endzeilen für den Cursor. Darüberhinaus werden durch Bits 5 und 6 die Cursormodi folgendermaßen ausgewählt:

BIT		
6	5	CURSOR MODUS
0	0	Nichtblinkend
0	1	Kein Cursor
1	0	Blinkt mit 1/16 Feldrate
1	1	Blinkt mit 1/32 Feldrate

Man beachte, daß durch die Programmierbarkeit von Anfang und Ende der Cursor Abtastzeile sowohl ein blockförmiger als auch ein unterstreichender Cursor gebildet werden kann. Register R14 und R15 bestimmen die Zeichenposition des Cursor innerhalb des gesamten 16K Adreßfeldes.

Anzeige Anfangsadresse; höherwertig (R12), niederwertig (R13)

Diese Register bilden zusammen ein 14-Bit Register, deren Inhalt die Speicheradresse des ersten Zeichens der abgebildeten Abtastung darstellt (das Zeichen in der linken oberen Ecke des Bildschirms). Die darauf folgenden Speicheradressen werden nach einem CCLK Eingangspuls vom 6545-1 erzeugt. Bildschirmrollen (scrolling) wird dadurch herbeigeführt, daß R12 und R13 auf eine Speicheradresse verändert werden, die das erste Zeichen einer gewünschten Zeile enthält, die als erste angezeigt werden soll. Ganze Textseiten lassen sich über R12 und R13 genauso scrollen und verändern.

Cursor Position; höherwertig (R14), niederwertig (R15)

Diese Register bilden zusammen ein 14-Bit Register, deren Inhalt die Speicheradresse der laufenden Cursor Position darstellt. Wenn der Abtastzähler der Videoanzeige (MA Zeilen) mit dem Inhalt dieses Registers übereinstimmt und der Zähler für die abgetasteten Zeilen (RA Zeilen) innerhalb der Grenzen liegt, die durch R10 und R11 gesetzt sind, wird die CURSOR-Ausgabe aktiv. Bit 5 des Modus Kontroll-Registers läßt sich zur Verzögerung der CURSOR-Ausgabe um einen vollen CCLK Zeitraum benutzen, um auch Speicher mit langsamem Zugriff anzupassen.

LPEN; höherwertig (R16) niederwertig (R17)

Diese Register bilden zusammen ein 14-Bit Register, deren Inhalt die Strobe Position eines Lichtstifts darstellen. Damit wird die Videoadresse, an der der Strobepuls erfolgte, angezeigt. Wenn der LPEN Eingang von niedriger auf hohe Spannung wechselt, wird bei der nächsten negativen Flanke des CCLK der Inhalt des internen Abtastzählers in die Register R16 und R17 gespeichert.

6560 (VIC) Video Interface Chip

Der 6560 Videointerface Baustein (VIC) wurde für Farbvideografik Anwendungen entwickelt, wie etwa für preiswerte CRT Terminals, biomedizinische Monitore, Anzeigen von Kontrollsystemen und für Arkade- oder Heimvideospiele. Dieser enthält alle Schaltkreise, die zur Darstellung farbiger, programmierbarer Zeichengraphik auf einem hochauflösenden Bildschirm notwendig sind. Der VIC bietet außerdem Toneffekte und A/D Wandler, die bei Videospielen Anwendung finden.

Eigenschaften

- Vollständig erweiterbares System mit einem 16K Byte Adreßraum
- Das System benutzt 8-Bit breite ROM und 4-Bit breite RAM nach Industriestandard
- Maskenprogrammierbare Sync Erzeugung, NTSC-6560, PAL-6561
- Farberzeugung auf dem Baustein (16 Farben)
- Bis zu sechshundert unabhängig programmierbare und verschiebbare Hintergrundbereiche auf einem Standard TV
- Bildzentriereinrichtung
- Bildschirm-Darstellungsgröße bis zu 192 Punkte horizontal mal 200 Punkte vertikal
- Zwei wählbare Grafikzeichengrößen
- Im Baustein eingebautes Tonsystem enthält:
  a) drei unabhängige programmierbare Tongeneratoren
  b) Generator für weißes Rauschen
  c) Amplitutenmodulator
- Zwei auf dem Chip eingebaute 8-Bit A/D Wandler
- Auf dem Baustein DMA und Adreßerzeugung
- Während der Bildschirmauffrischung keine CPU Wartezyklen oder Bildschirmzugriffe
- Schalter für Interlace/Non-Interlace Modus
- Sechzehn adressierbare Kontrollregister
- Lichtgewehr, Lichtstift für Zielspiele
- Zwei Modi für Farbdarstellung

A: Interlace Modus: Ein normales Videobild wird sechzigmal pro Sekunde (in Europa fünfzigmal) auf einen Fernseher gegeben. Der Interlace Modus (verschachtelter Modus) halbiert die Anzahl Wiederholungen. Wird eine Multiplex Einrichtung benutzt, ist es möglich, das VIC Bild mit einem Bild aus einer anderen Quelle zu mischen.

Ausschalten: POKE 36864, PEEK (36864) AND 127
Anschalten: POKE 36864, PEEK (36864) OR 128

B: Bildschirm Anfang - horizontal: Hiermit wird die Positionierung des Bildes auf dem TV Bildschirm bestimmt. Der Normalwert beträgt 5. Erniedrigung des Wertes verschiebt das Bild nach links, Erhöhung verschiebt es nach rechts.

Zur Veränderung des Wertes: POKE 36864, PEEK (36864) AND 128 OR X

Speicher-stelle	Startwert-5K VIC		Bitfunktion
Hex	Binär	Dezimal	
9000	00000101	5	ABBBBBBB
9001	00011001	25	CCCCCCCC
9002	10010110	150	HDDDDDDD
9003	b0101110	46 oder 176	GEEEEEEF
9004	bbbbbbbb	b	GGGGGGGG
9005	11110000	240	HHHHIIII
9006	00000000	0	JJJJJJJJ
9007	00000000	0	KKKKKKKK
9008	11111111	255	LLLLLLLL
9009	11111111	255	MMMMMMMM
900A	00000000	0	NRRRRRRR
900B	00000000	0	OSSSSSSS
900C	00000000	0	PTTTTTTT
900D	00000000	0	QUUUUUUU
900E	00000000	0	WWWWVVVV
900F	00011011	27	XXXXYZZZ

A: Interlace Modus:
0 = aus, 1 = an
B: Bildanfang-horizontal
C: Bildanfang-vertikal
D: Anzahl Videospalten
E: Anzahl Videozeilen
F: Zeichengröße:
0 = 8x8, 1 = 8x16
G: Rasterwert
H: Bildschirm-Speicherstelle
I: Zeichen-Speicherstelle
J: Lichtstift-horizontal
K: Lichtstift-vertikal
L: Steuerknüppel 1
M: Steuerknüppel 2
N: Baß-Schalter
O: Alt-Schalter
P: Sopran-Schalter
Q: Rausch-Schalter
R: Baß-Frequenz
S: Alt-Frequenz
T: Sopran-Frequenz
U: Rausch-Frequenz
V: Lautstärke
W: Hilfsfarbe
X: Bildfarbe
Y: Invers Modus
0 = an, 1 = aus
Z: Randfarbe
(b = beliebig)

C: Bildanfang-vertikal: Damit wird die vertikale Ausrichtung des Bildes bestimmt. Der Normalwert ist 25. Durch Erniedrigung wird für jeden Zahlenwert das Bild um 2 Punktreihen nach oben verschoben, durch Erhöhung nach unten.

Zur Veränderung des Wertes: POKE 36865,X

D: Zahl der Videospalten: Ist normalerweise auf 22 gesetzt. Eine Veränderung dieses Wertes verändert die Anzeige entsprechend. Zahlen über 27 führen zu einem Bild mit 27 Spalten. Die Cursorkontrolle basiert auf einem festen Wert von 22 Spalten und die Veränderung dieser Zahl führt zu einem Fehlverhalten der Cursorkontrolle.

Zur Veränderung des Wertes: POKE 36866, PEEK (36866) AND 128 OR X

E: Zahl der Videozeilen: Die Zeilenzahl darf im Bereich 0 bis 23 liegen. Eine größere Zeilenzahl verursacht eine unsinnige Anzeige am unteren Bildschirmrand.

Zur Veränderung des Wertes: POKE 36867, PEEK (36867) AND 129 OR (X*2)

F: Zeichengröße: Durch dieses Bit wird die Matrixgröße für jedes Zeichen bestimmt. Eine 0 setzt den Normalmodus, bei dem die Zeichen 8x8 Punkte groß sind. Eine 1 setzt den 8x16 Modus, bei dem jedes Zeichen zweimal so groß ist. Der 8x16 Modus wird normalerweise für hochauflösende Graphik benutzt, bei dem man viele unterschiedliche Zeichen auf dem Bildschirm darstellen möchte.

Zum Setzen des 8x8 Modus: POKE 36867, PEEK (36867) AND 254
Zum Setzen des 8x16 Modus: POKE 36867, PEEK (36867) OR 1

G: Rasterwert: Diese Zahl wird zur Synchronisierung des Lichtstifts mit dem TV Bild benutzt.

H: Bildschirm-Speicherstelle: Damit wird festgelegt, in welchem Teil des Speichers der VIC den Inhalt des Bildschirms findet. Das höchste Bit in Speicherstelle 36869 muß eine 1 sein. Die Bits 4-6 der Stelle 36869 sind die Bits 10-12 der Bildschirmadresse und Bit 7 der Stelle 36866 ist Bit 9 der Adresse des Bildschirms. Zur Bestimmung der Bildschirmspeicherstelle benutze die Formel:

S = 4*(PEEK (36866) AND 128) + 64* (PEEK (36869) AND 112)

Man beachte, daß mit Bit 7 der Stelle 36866 außerdem die Speicherstelle des Farbspeichers bestimmt wird. Ist dieses Bit eine 0, beginnt der Farbspeicher bei 37888. Ist dieses Bit eine 1, beginnt er bei 38400. Die entsprechende Formel lautet:

C = 37888 + 4* (PEEK (36866) AND 128)

I: Zeichen-Speicherstelle: Hiermit wird festgelegt, wo Informationen über die Zeichenform gespeichert sind. Normalerweise weist dieser Zeiger auf den Zeichengenerator ROM, der sowohl die Großbuchstaben/Graphikzeichen als auch den Groß/Kleinbuchstabensatz enthält. Man kann jedoch durch einen einfachen POKE Befehl diesen Zeiger auf eine RAM Stelle weisen lassen, wodurch ein eigener Zeichensatz und hochauflösende Graphik möglich ist.

Zur Veränderung des Wertes: POKE 36869, PEEK (36869) AND 240 OR X
(beachte die nächste Tabelle)

J: Lichtstift-horizontal: Dieses enthält die zwischengespeicherte Punktnummer unter dem Lichtstift, vom linken Bildrand gezählt.

K: Lichtstift-vertikal: Dieses enthält die zwischengespeicherte Punktnummer unter dem Lichtstift, vom oberen Bildrand gezählt.

X	Speicherstelle		Inhalt
Wert	Hex	Dezimal	
0	8000	32768	normale Großbuchstaben
1	8400	33792	invertierte Großbuchstaben
2	8800	34816	normale Kleinbuchstaben
3	8C00	35840	invertierte Kleinbuchstaben
4	9000	36864	nicht verfügbar
5	9400	37888	nicht verfügbar
6	9800	38912	VIC Chip-nicht verfügbar
7	9C00	39936	ROM-nicht verfügbar
8	0000	0	nicht verfügbar
9	-	-	nicht verfügbar
10	-	-	nicht verfügbar
11	-	-	nicht verfügbar
12	1000	4096	RAM
13	1400	5120	RAM
14	1800	6144	RAM
15	1C00	7168	RAM

L: Steuerknüppel X: Enthält den digitalisierten Wert eines veränderbaren Widerstandes (Spielknüppel). Der Wert liegt im Bereich zwischen 0 und 255.

M: Steuerknüppel Y: Siehe Steuerknüppel X für einen zweiten Analogeingang.

N: Bass-Schalter: Ist dieses Bit eine 0, bleibt Stimme 1 stumm. Eine 1 in diesem Bit führt zu einem Ton, der durch Frequenz 1 bestimmt ist.

Ausschalten: POKE 36874, PEEK(36874) AND 127
Anschalten: POKE 36874, PEEK(36874) OR 128

O: Alt-Schalter: Siehe Baß-Schalter.

P: Sopran-Schalter: Siehe Baß-Schalter.

Q: Rausch-Schalter: Siehe Baß-Schalter.
R: Baß-Frequenz: Dabei handelt es sich um einen Wert, der der Frequenz des zu spielenden Tons entspricht. Je höher die Zahl, um so höher die Tonhöhe.

Die Frequenz des Tons wird durch folgende Formel in Schwingungen pro Sekunde (Hertz) bestimmt:

Frequenz = Clock / (127-X)

X ist eine Zahl zwischen 0 und 127, die in das Frequenzregister gesetzt wird. Ist X gleich 127, dann benutze für X in der Formel -1. Den Wert für Clock kann man der folgenden Tabelle entnehmen:

Register	NTSC (US TV's)	PAL (Europäisch)
36874	3995	4329
36875	7990	8659
36876	15980	17320
36877	31960	34640

Zum Einstellen des Wertes: POKE 36874, PEEK (36874) AND 128 OR X

S: Alt-Frequenz: Dabei handelt es sich um einen Wert, der der Frequenz des zu spielenden Tons entspricht. Je höher die Zahl, um so höher die Tonfrequenz.

T: Sopran-Frequenz: Dabei handelt es sich um einen Wert, der der Frequenz des zu spielenden Tons entspricht. Je höher die Zahl, um so höher die Tonfrequenz.

Zum Einstellen des Wertes: POKE 36876, PEEK (36876) AND 128 OR X

U: Rausch-Frequenz: Dabei handelt es sich um einen Wert, der der Frequenz des zu spielenden Rauschens entspricht. Je höher die Zahl, um so höher die Rauschfrequenz.

Zum Einstellen des Wertes: POKE 36877, PEEK (36877) AND 128 OR X

V: Lautstärke der Geräusche: Dabei handelt es sich um eine Lautstärkekontrolle für alle zu spielenden Geräusche. 0 bedeutet aus und 15 größte Lautstärke.

Zur Veränderung des Wertes: POKE 36878, PEEK (36878) AND 240 OR X

W: Hilfsfarbe: Dieses Register enthält die Farbnummer der Hilfsfarbe. Der Wert kann im Bereich von 0 bis 15 liegen.

Zum Einstellen des Wertes: POKE 36878, PEEK (36878) AND 15 OR (16*X)

X: Bildfarbe: Eine Zahl zwischen 0 und 15 setzt die Farbe des Bildes.
Zum Einstellen des Wertes: POKE 36879, PEEK(36879) AND 240 OR X

Y: Invertierter Modus: Eine 1 in diesem Bit bedeutet ein normales Zeichen und eine 0 an dieser Stelle führt dazu, daß alle Zeichen invertiert dargestellt werden.

Zum Anschalten des invertierten Modus: POKE 36879, PEEK (36879) AND 247
Zum Ausschalten des invertierten Modus: POKE 36879, PEEK (36879) OR 8

Z: Randfarbe: Eine Zahl zwischen 0 und 7 bestimmt die Farbe des Bildschirmrandes.

Zum Einstellen des Wertes: POKE 36879, PEEK (36879) AND 248 OR X

6522 Vielseitiger Interface Adapter (VIA)

Der 6522 VIA enthält zwei Peripheriekanäle mit Eingangszwischenspeicher, zwei leistungsfähige Intervall-Zeitgeber und ein Schieberegister seriell-parallel/parallel-seriell.

Beschreibung des 6522 VIA

Adresse	Beschreibung	Register
9110	Kanal B	AAAAAAAA
9111	Kanal A (mit Quittung)	BBBBBBBB
9112	Datenrichtung B	CCCCCCCC
9113	Datenrichtung A	DDDDDDDD
9114	Zeitgeber #1, unteres Byte	EEEEEEEE
9115	Zeitgeber #1, oberes Byte	FFFFFFFF
9116	Zeitgeber #1, unteres zu ladendes Byte	GGGGGGGG
9117	Zeitgeber #1, oberes zu ladendes Byte	HHHHHHHH
9118	Zeitgeber #2, unteres Byte	IIIIIIII
9119	Zeitgeber #2, oberes Byte	JJJJJJJJ
911A	Schieberegister	KKKKKKKK
911B	Hilfskontrolle	LLMNNNOP
911C	Peripheriekontrolle	QQQRSSST
911D	Interruptflags	UVWXYZab
911E	Interrupt möglich	cedfghij
911F	Kanal A (ohne Quittung)	kkkkkkkk

Kanal B I/O Register

Diese acht Bits sind mit den acht Anschlüssen verbunden, die den Kanal B bilden. Jeder Anschluß kann als Eingang oder Ausgang geschaltet werden.

Eine Eingangszwischenspeicherung steht bei diesem Kanal zur Verfügung. Wenn der Zwischenspeichermodus erlaubt ist, werden die Daten im Register dann eingefroren, wenn die CB1 Interruptflag gesetzt ist. Das Register bleibt solange zwischengespeichert, bis die Interruptflag gelöscht wird.

Bei einer Ausgabe ist bei diesem Kanal eine Quittierung möglich. CB2 wirkt dabei wie ein DATA READY Signal. Dieses muß vom Benutzerprogramm überprüft werden. CB1 wirkt als DATA ACCEPTED Signal, das von der Einheit, die an den Kanal angeschlossen ist, kontrolliert werden muß. Wenn DATA ACCEPTED an den 6522 gesandt wurde, wird die DATA READY Leitung gelöscht und die Interruptflag gesetzt.

Kanal A I/O Register

Diese acht Bits sind mit den acht Anschlüssen verbunden, die den Kanal A bilden. Jeder Anschluß kann als Eingang oder Ausgang geschaltet werden. Eine Quittierung ist sowohl bei Lese- als auch bei Schreiboperationen möglich. Die Quittierung beim Schreiben ähnelt der bei Kanal B. Quittierung des Lesens erfolgt automatisch. Der CA1 Eingangsanschluß wirkt als DATA READY Signal. Der CA2 Anschluß (als Ausgang benutzt) wird für ein DATA ACCEPTED Signal benutzt. Bei Empfang eines DATA READY Signals wird eine Flag gesetzt. Der Baustein kann so eingestellt werden, daß er ein Interrupt erzeugt, oder die Flag kann un-

ter Programmkontrolle erfaßt werden. Das DATA ACCEPTED Signal kann entweder als Puls oder als Gleichspannungswert vorliegen. Es wird durch die CPU heruntergeschaltet und durch das DATA READY Signal gelöscht.

Datenrichtungsregister für Kanal B

Mit diesem Register wird kontrolliert, ob ein bestimmtes Bit in Kanal B als Eingang oder Ausgang benutzt wird. Jedes Bit des Datenrichtungsregisters (DDR) entspricht einem Bit von Kanal B. Ist ein Bit in DDR auf 1 gesetzt, ist das entsprechende Bit des Kanals ein Ausgang. Ist ein Bit in DDR auf 0 gesetzt, ist das entsprechende Bit des Kanals ein Eingang.

Ist beispielsweise DDR auf 7 gesetzt, wird Kanal B folgendermaßen eingestellt:

Bit-Nummer	DDR	Funktion von Kanal B
0	1	Ausgang
1	1	Ausgang
2	1	Ausgang
3	0	Eingang
4	0	Eingang
5	0	Eingang
6	0	Eingang
7	0	Eingang

Datenrichtungsregister für Kanal A

Dieses gleicht dem DDR für Kanal B mit Ausnahme, daß es auf Kanal A wirkt.

E, F, G, H: Zeitgeberkontrollen

Auf dem 6522 Baustein gibt es zwei Zeitgeber. Die Zeitgeber können so gesetzt werden, daß sie entweder automatisch abwärts zählen oder Pulse zählen, die sie über den VIA empfangen haben. Die Betriebsart wird durch das Hilfskontrollregister ausgewählt.

Zeitgeber T1 auf dem 6522 besteht aus zwei 8-Bit Zwischenspeichern und einem 16-Bit Zähler. Die verschiedenen Betriebsarten des Zeitgebers werden durch Setzen des Hilfskontrollregisters (ACR) ausgewählt. Die Zwischenspeicher werden dazu benutzt, ein 16-Bit Datenwort in den Zähler zu laden. Während eine Zahl in die Zwischenspeicher geladen wird, wird der Zählvorgang selbst nicht beeinflußt.

Nach dem Setzen zählt der Zähler mit einer Rate von 1MHz abwärts. Erreicht er den Wert 0, wird eine Interruptflag gesetzt und IRQ schaltet nach unten. Abhängig davon, wie der Zeitgeber gesetzt wurde, werden entweder weitere Interrupts gesperrt oder die Werte der beiden Zwischenspeicher automatisch in den Zähler geladen und der Zählvorgang fortgesetzt. Der Zeitgeber kann auch so eingestellt werden, daß er das Ausgangssignal an einem peripheren Anschluß invertiert und zwar jedesmal dann, wenn er 0 erreicht und zurückgesetzt wird.

Die Zeitgeberspeicherstellen arbeiten beim Lesen und Schreiben unterschiedlich.

In den Zeitgeber schreiben

E: In den niederwertigen Teil des Zwischenspeichers schreiben. Dieser Zwischenspeicher kann in das untere Byte des 16-Bit Zählers geladen werden.

F: In den höherwertigen Teil des Zwischenspeichers schreiben, in den höherwertigen Teil des Zählers schreiben, den niederwertigen Teil des Zwischenspeichers in den niederwertigen Teil des Zählers transferieren und die Interruptflag des Zeitgebers T1 zurücksetzen. Mit anderen Worten, wenn diese Speicherstelle gesetzt wird, wird der Zähler geladen.

G: Das gleiche wie unter E.

H: In den höherwertigen Teil des Zwischenspeichers schreiben und die Interruptflag des Zeitgebers T1 zurücksetzen.

Zeitgeber T1 lesen

E: Den niederwertigen Teil des Zählers von Zeitgeber T1 lesen und die Interruptflag von Zeitgeber T1 zurücksetzen.

F: Den höherwertigen Teil des Zählers von Zeitgeber T1 lesen.

G: Den niederwertigen Teil des Zwischenspeichers von Zeitgeber T1 lesen.

H: Den höherwertigen Teil des Zwischenspeichers von Zeitgeber T1 lesen.

Zeitgeber T2

Dieser Zeitgeber arbeitet als Intervallzeitgeber (Monovibrator) oder als ein Zähler, der die negativen Pulse an Anschluß 6 des Kanal B zählt. Ein Bit des ACR wählt den entsprechenden Modus von Zeitgeber T2 aus.

In Zeitgeber T2 schreiben

I: Niederwertiges Byte in Zwischenspeicher von Zeitgeber T2 schreiben.

J: Höherwertiges Byte in Zähler von Zeitgeber T2 schreiben, niederwertigen Teil des Zwischenspeichers in niederwertigen Teil des Zählers transferieren, Interruptflag von Zeitgeber T2 löschen.

Zeitgeber T2 lesen

I: Niederwertiges Byte des Zählers von Zeitgeber T2 lesen und Interruptflag von Zeitgeber T2 löschen.

K: Schieberegister

Beim Schieberegister handelt es sich um ein Register, das sich selbst durch den CB2 Anschluß rotiert. Das Schieberegister kann mit einem beliebigen 8-Bit Muster geladen werden,

welches durch den CB1 Anschluß herausgeschoben werden kann. Ebenso kann die Eingabe über den CB1 Anschluß in das Schieberegister geschoben und dann gelesen werden. Auf diese Weise ist es für seriell-parallel und parallel-seriell Umwandlungen sehr gut geeignet.

Das Schieberegister wird durch die Bits 2-4 des Hilfskontrollregisters (ACR) beeinflußt.

L, M, N, O, P: Hilfskontrollregister (ACR)

L: Kontrolle von Zeitgeber 1

Bit # 7 6

0 0 Monovibrator Modus (Ausgabe auf PB7 gesperrt)
0 1 Multivibrator Modus (Ausgabe auf PB7 gesperrt)
1 0 Monovibrator Modus (Ausgabe auf PB7 erlaubt)
1 1 Multivibrator Modus (Ausgabe auf PB7 erlaubt)

M: Kontrolle von Zeitgeber 2

Zeitgeber 2 erlaubt zwei Betriebsarten. Wenn dieses Bit 0 ist, wirkt Zeitgeber 2 wie ein Intervallzeitgeber als Monovibrator. Wenn dieses Bit gleich 1 ist, zählt Zeitgeber 2 eine vorherbestimmte Anzahl von Pulsen an Anschluß PB6.

N: Kontrolle des Schieberegister

Bit # 4 3 2

0 0 0 Schieberegister gesperrt
0 0 1 Hineinschieben (von CB1) unter Kontrolle des Zeitgeber 2
0 1 0 Hineinschieben unter Kontrolle der System Clock-Pulse
0 1 1 Hineinschieben unter Kontrolle externer Clock-Pulse
1 0 0 Multivibrator Modus mit einer von Zeitgeber 2 bestimmten Rate
1 0 1 Herausschieben unter Kontrolle des Zeitgeber 2
1 1 0 Herausschieben unter Kontrolle der System Clock-Pulse
1 1 1 Herausschieben unter Kontrolle externer Clock-Pulse

O: Kanal B Zwischenspeicherung erlaubt

Solange dieses Bit 0 ist, spiegelt das Register von Kanal B direkt die Daten an diesen Anschlüssen wieder.

Wenn dieses Bit auf 1 gesetzt ist, werden die Daten, die an den Eingangsanschlüssen von Kanal B anliegen, im Baustein dann zwischengespeichert, wenn die Interruptflag CB1 gesetzt ist. Solange die Interruptflag CB1 gesetzt bleibt, können sich die Daten an den Anschlüssen verändern ohne den Inhalt des Registers von Kanal B zu beeinflussen. Man beachte, daß die CPU immer die Register (die Zwischenspeicher) und nicht die Anschlüsse liest.

Zwischenspeicherung der Eingabe kann bei jeder der für CB2 verfügbaren Eingangs- oder Ausgangsmodi verwandt werden.

P: Kanal A Zwischenspeicherung erlaubt

Solange dieses Bit 0 ist, spiegelt das Register von Kanal A direkt die Daten an diesen Anschlüssen wieder.

Wenn dieses Bit auf 1 gesetzt ist, werden die Daten, die an den Eingangsanschlüssen von Kanal A anliegen, im Baustein dann zwischengespeichert, wenn die Interruptflag CA1 gesetzt ist. Solange die Interruptflag CA1 gesetzt bleibt, können sich die Daten an den Anschlüssen verändern, ohne den Inhalt des Registers von Kanal A zu beeinflussen. Man beachte, daß die CPU immer die Register (die Zwischenspeicher) und nicht die Anschlüsse liest.

Zwischenspeicherung der Eingabe kann bei jeder der für CA2 verfügbaren Eingangs- oder Ausgangsmodi verwandt werden.

Q, R, S, T: Das Peripherie-Kontrollregister

Q: Kontrolle von CB2

	Q	Q	Q	
Bit #	**7**	**6**	**5**	**Beschreibung**
	0	0	0	Interrupteingabe Modus
	0	0	1	Unabhängiger Interrupteingabe Modus
	0	1	0	Eingabemodus
	0	1	1	Unabhängiger Eingabemodus
	1	0	0	Ausgabemodus mit Quittung
	1	0	1	Pulsausgabe Modus
	1	1	0	Manueller Ausgabemodus (CB2 wird unten gehalten)
	1	1	1	Manueller Ausgabemodus (CB2 wird oben gehalten)

Interrupteingabe Modus:

Die CB2 Interruptflag (IFR Bit 3) wird durch eine negative Flanke (hoch nach niedrig) auf der CB2 Eingangsleitung gesetzt. Durch einen Lese- oder Schreibvorgang an Kanal B wird das CB2 Interruptbit gelöscht.

Unabhängiger Interrupteingabe Modus:

Wie oben wird die CB2 Interruptflag durch eine negative Flanke auf der CB2 Eingangsleitung gesetzt. Ein Lese- oder Schreibvorgang an Kanal B löscht dagegen die Flag nicht.

Eingabemodus:

Die CB2 Interruptflag (IFR Bit 3) wird durch eine positive Flanke (niedrig nach hoch) auf der CB2 Eingangsleitung gesetzt. Durch einen Lese- oder Schreibvorgang an Kanal B wird das CB2 Interruptbit gelöscht.

Unabhängiger Eingabemodus:

Wie oben wird die CB2 Interruptflag durch eine positive Flanke auf der CB2 Eingangsleitung gesetzt. Ein Lese- oder Schreibvorgang an Kanal B löscht dagegen die Flag nicht.

Ausgabemodus mit Quittung:

Beim Schreiben auf Kanal B wird die CB2 Leitung nach unten gesetzt. Bei einem aktiven Übergang auf der CB1 Leitung wird sie wieder auf hoch gesetzt.

Pulsausgabe Modus:

Nach einem Schreiben auf Kanal B wird die CB2 Leitung für einen Zyklus heruntergesetzt.

Manueller Ausgabemodus:

Die CB2 Leitung wird unten gehalten.

Manueller Ausgabemodus:

Die CB2 Leitung wird oben gehalten.

R: Kontrolle von CB1

Dieses Bit selektiert den aktiven Übergang des Eingangssignals, das am Anschluß CB1 angelegt wird. Wenn dieses Bit 0 ist, wird die CB1 Interruptflag bei einem negativen Übergang (hoch nach niedrig) gesetzt. Wenn dieses Bit eine 1 ist, wird die CB1 Interruptflag bei einem positiven Übergang (niedrig nach hoch) gesetzt.

S: Kontrolle von CA2

	S	S	S	
Bit #	**3**	**2**	**1**	**Beschreibung**
	0	0	0	Interrupteingabe Modus
	0	0	1	Unabhängiger Interrupteingabe Modus
	0	1	0	Eingabemodus
	0	1	1	Unabhängiger Eingabemodus
	1	0	0	Ausgabemodus mit Quittung
	1	0	1	Pulsausgabe Modus
	1	1	0	Manueller Ausgabemodus (CA2 wird unten gehalten)
	1	1	1	Manueller Ausgabemodus (CA2 wird oben gehalten)

Interrupteingabe Modus:

Die CA2 Interruptflag (IFR Bit 0) wird durch eine negative Flanke (hoch nach niedrig) auf der CA2 Eingangsleitung gesetzt. Durch einen Lese- oder Schreibvorgang an Kanal A wird das CA2 Interruptbit gelöscht.

Unabhängiger Interrupteingabe Modus:

Wie oben wird die CA2 Interruptflag durch eine negative Flanke auf der CA2 Eingangsleitung gesetzt. Ein Lese- oder Schreibvorgang an Kanal A löscht dagegen die Flag nicht.

Eingabemodus:

Die CA2 Interruptflag (IFR Bit 0) wird durch eine positive Flanke (niedrig nach hoch) auf der CA2 Eingangsleitung gesetzt. Durch einen Lese- oder Schreibvorgang an Kanal A wird das CA2 Interruptbit gelöscht.

Unabhängiger Eingabemodus:

Wie oben wird die CA2 Interruptflag durch eine positive Flanke auf der CA2 Eingangsleitung gesetzt. Ein Lese- oder Schreibvorgang an Kanal A löscht dagegen die Flag nicht.

Ausgabemodus mit Quittung:

Beim Schreiben auf Kanal A wird die CA2 Leitung heruntergesetzt. Bei einem aktiven Übergang auf der CA1 Leitung wird sie wieder auf hoch gesetzt.

Pulsausgabe Modus:

Nach einem Schreiben auf Kanal A wird die CA2 Leitung für einen Zyklus heruntergesetzt.

Manueller Ausgabemodus:

Die CA2 Leitung wird unten gehalten.

Manueller Ausgabemodus:

Die CA2 Leitung wird hoch gehalten.

T: Kontrolle von CA1

Dieses Bit des PCR selektiert den aktiven Übergang des Eingangssignals, das am Anschluß CA1 angelegt wird. Wenn dieses Bit 0 ist, wird die CA1 Interruptflag bei einem negativen Übergang (hoch nach niedrig) am Anschluß CA1 gesetzt. Wenn dieses Bit eine 1 ist, wird die CA1 Interruptflag bei einem positiven Übergang (niedrig nach hoch) gesetzt.

Es gibt zwei Register, die mit Interrupts im Zusammenhang stehen: das INTERRUPT FLAG REGISTER (IFR) und das INTERRUPT ENABLE REGISTER (IER). Das IFR enthält 8 Bit, von denen jedes mit einem Register im 6522 verbunden ist. Jedes Bit im IFR hat ein entsprechendes Bit im IER. Die Flag wird gesetzt, wenn ein Register ein Interrupt wünscht. Ein Interrupt findet jedoch solange nicht statt, bis das entsprechende Bit im IER gesetzt ist.

UVWXYZab: Interruptflag Register

Wenn die Flag gesetzt ist, versucht der Anschluß, der dieser Flag entspricht, den 6502 zu unterbrechen. Bei Bit U handelt es sich nicht um eine normale Flag. Es geht hoch, wenn sowohl die Flag und das entsprechende Bit im IER Register gesetzt sind. Es kann nur dadurch gelöscht werden, daß alle Flags im IFR gelöscht werden oder dadurch, daß alle aktiven Interrupts im IER gesperrt werden.

	Gesetzt durch	Gelöscht durch
U	IRQ Status	
V	Zeitgeber 1 abgelaufen	Lesen von Zeitgeber 1 niederwertiger Zähler und Schreiben von Zeitgeber 1 höherwertiger Zwischenspeicher
W	Zeitgeber 2 abgelaufen	Lesen von Zeitgeber 2 niederwertiger Zähler und Schreiben von Zeitgeber 2 höherwertiger Zwischenspeicher
X	CB1 Anschluß aktiver Übergang	Lesen oder Schreiben an Kanal B
Y	CB2 Anschluß aktiver Übergang	Lesen oder Schreiben an Kanal B
Z	Vollendung von 8 mal Schieben	Lesen oder Schreiben im Schieberegister
a	CA1 Anschluß aktiver Übergang	Lesen oder Schreiben an Kanal A (BBBBBBBB in obiger Tabelle)
b	CA2 Anschluß aktiver Übergang	Lesen oder Schreiben an Kanal A (BBBBBBBB in obiger Tabelle)

cdefghij: INTERRUPT ENABLE REGISTER (IER)

c: ENABLE Kontrolle
Wenn dieses Bit während eines Schreibvorgangs in dieses Register 0 ist, löscht jede 1 in den Bits 0-6 das entsprechende Bit im IER. Wenn dieses Bit während eines Schreibvorgangs in dieses Register 1 ist, setzt jede 1 in den Bits 0-6 das entsprechende Bit im IER.

d Zeitgeber 1 Zeitablauf erlaubt
e Zeitgeber 2 Zeitablauf erlaubt
f CB1 Interrupt erlaubt
g CB2 Interrupt erlaubt
h Schiebe Interrupt erlaubt
i CA1 Interrupt erlaubt
j CA2 Interrupt erlaubt

Kanal A

Dieser gleicht BBBBBBBB mit der Ausnahme, daß die Quittungsleitungen (CA1 und CA2) durch Operationen an diesem Kanal nicht beeinflußt werden.

6526 Komplexer Interface Adapter (CIA)

Registerplan

RS3	RS2	RS1	RS0	REG	Name	
0	0	0	0	0	PRA	Peripheres Datenregister A
0	0	0	1	1	PRB	Peripheres Datenregister B
0	0	1	0	2	DDRA	Datenrichtungsregister A
0	0	1	1	3	DDRB	Datenrichtungsregister B
0	1	0	0	4	TA LO	Zeitgeber A unteres Register
0	1	0	1	5	TA HI	Zeitgeber A oberes Register
0	1	1	0	6	B LO	Zeitgeber B unteres Register
0	1	1	1	7	B HI	Zeitgeber B oberes Register
1	0	0	0	8	TOD 1/10	1/10 Sekunden Register
1	0	0	1	9	TOD SEC	Sekunden Register
1	0	1	0	A	TOD MIN	Minuten Register
1	0	1	1	B	TOD HR	Stunden AM/PM Register
1	1	0	0	C	SDR	Serielles Datenregister
1	1	0	1	D	ICR	Interrupt Kontrollregister
1	1	1	0	E	CRA	Kontrollregister A
1	1	1	1	F	CRB	Kontrollregister B

I/O Kanäle (PRA, PRB, DDRA, DDRB)

Die Kanäle A und B bestehen beide aus einem peripheren Datenregister (PR) von acht Bit und aus einem Datenrichtungsregister (DDR) von acht Bit. Wenn ein Bit in DDR auf 1 gesetzt ist, fungiert das entsprechende Bit in PR als Ausgang. Wird ein DDR Bit auf 0 gesetzt, ist das entsprechende PR Bit als Eingang definiert. Bei einem Lesevorgang gibt das PR die Information wieder, die an den aktuellen Kanalanschlüssen (PA0-PA7, PB0-PB7) für Eingabe- wie Ausgabebits anliegen. Kanal A und Kanal B haben sowohl passive als auch aktive pull-up Einrichtungen. Dadurch sind sie sowohl CMOS als auch TTL kompatibel. Beide Kanäle können bis zu zwei TTL Eingänge treiben. Neben der normalen I/O Operation können PB6 und PB7 auch als Zeitgeberausgänge arbeiten.

Quittung (handshaking)

Eine Quittierung bei Datenübermittlung kann durch Benutzung des $\overline{PC}$ Ausgangsanschluß und des $\overline{FLAG}$ Eingangsanschluß erreicht werden. Nach einem Lese- oder Schreibvorgang an Kanal B geht $\overline{PC}$ für einen Zyklus nach unten. Dieses Signal kann dazu benutzt werden, „Daten fertig" an Kanal B oder „Daten akzeptiert" von Kanal B anzuzeigen. Die Quittierung einer Datenübermittlung von 16 Bit (indem man Kanal A und Kanal B benutzt) ist dadurch möglich, daß man immer zuerst Kanal A liest bzw. schreibt. $\overline{FLAG}$ ist ein Eingang, der auf eine negative Flanke reagiert und dazu benutzt werden kann, die $\overline{PC}$ Ausgabe eines anderen 6526 zu empfangen. Er läßt sich auch als Interrupteingang allgemein verwenden. Jeder negative Übergang von $\overline{FLAG}$ setzt das $\overline{FLAG}$ Interruptbit.

REG	NAME	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	PRA	PA_7	PA_6	PA_5	PA_4	PA_3	PA_2	PA_1	PA_0
1	PRB	PB_7	PB_6	PB_5	PB_4	PB_3	PB_2	PB_1	PB_0
2	DDRA	DPA_7	DPA_6	DPA_5	DPA_4	DPA_3	DPA_2	DPA_1	DPA_0
3	DDRB	DPB_7	DPB_6	DPB_5	DPB_4	DPB_3	DPB_2	DPB_1	DPB_0

Intervall Zeitgeber (Zeitgeber A, Zeitgeber B)

Jeder Intervall Zeitgeber besteht aus einem 16-Bit Zähler, der nur gelesen werden, und einem 16-Bit Zwischenspeicher, der nur beschrieben werden kann. Daten, die in den Zeitgeber geschrieben werden, werden im Zwischenspeicher des Zeitgebers abgelegt. Beim Lesen der Daten aus dem Zeitgeber erhält man den aktuellen Inhalt des Zeitgeberzählers. Die Zeitgeber können unabhängig voneinander verwendet oder für erweiterte Anwendungen miteinander gekoppelt werden. Die verschiedenen Zeitgeber-Betriebsarten erlauben die Erzeugung von langen Verzögerungszeiten, von Pulsen variabler Breite, von Pulsfolgen und von unterschiedlichen Wellenformen. Unter Benutzung des CNT Eingangs können die Zeitgeber externe Pulse zählen oder Frequenzen, Pulsbreiten und Zeitverzögerungen externer Signale messen. Zu jedem Zeitgeber gehören entsprechende Kontrollregister, die eine unabhängige Kontrolle der folgenden Funktionen ermöglichen:

Start/Stop

Mit einem Kontrollbit kann man die Zeit durch den Mikroprozessor zu jeder Zeit starten oder stoppen.

PB An/Aus:

Ein Kontrollbit ermöglicht es, die Ausgabe des Zeitgebers auf eine Ausgangsleitung des Kanal B zu legen (PB6 für Zeitgeber A und PB7 für Zeitgeber B). Diese Funktion überschreibt das DDRB Kontrollbit und zwingt die entsprechende PB Leitung zur Ausgabe.

Umschalten/Puls

Ein Kontrollbit bestimmt die Ausgabeform von Kanal B. Jedesmal, wenn der Zeitgeber auf 0 gezählt hat, kann der Ausgang entweder den Spannungszustand wechseln oder einen einzelnen positiven Puls von der Dauer eines Zyklus erzeugen. Der Umschaltausgang wird beim Start des Zeitgebers hochgesetzt und durch $\overline{RES}$ heruntergesetzt.

Einzelpuls/Kontinuierlich

Ein Kontrollbit wählt zwischen zwei Zeitgeber-Betriebsarten. Beim Einzelpulsmodus zählt der Zeitgeber vom zwischengespeicherten Wert bis 0 herunter, erzeugt ein Interrupt, lädt den zwischengespeicherten Wert zurück und stoppt. Beim kontinuierlichen Modus zählt der

Zeitgeber vom zwischengespeicherten Wert bis 0 zurück, erzeugt ein Interrupt, lädt den zwischengespeicherten Wert zurück und wiederholt die ganze Prozedur kontinuierlich.

Erzwungenes Laden

Ein Strobe-Bit erlaubt, den Zeitgeberzwischenspeicher jederzeit in den Zeitgeberzähler zu laden, gleichgültig, ob der Zeitgeber läuft, oder nicht.

Eingabemodus:

Durch Kontrollbits kann man den Clock Puls, der den Zeitgeber herunterzählt, auswählen. Zeitgeber A kann Clock Pulse der Phase 2 oder externe Pulse, die am CNT Anschluß anliegen, zählen. Zeitgeber B kann Pulse der Phase 2 zählen, externe CNT Pulse, Pulse, welche beim Nulldurchgang des Zeitgeber A erzeugt werden (underflow) oder Underflowpulse des Zeitgeber A, während der CNT Anschluß oben gehalten wird.

Der Zeitgeberzwischenspeicher wird bei jedem Underflow des Zeitgebers in den Zeitgeberzähler geladen. Das gleiche geschieht durch ein erzwungenes Laden oder bei stehendem Zeitgeber nach einem Schreibvorgang in das obere Byte des Prescaler. Wenn der Zeitgeber läuft, lädt ein Schreibvorgang in das obere Byte den Zeitgeberzwischenspeicher, nicht aber den Zähler.

Lesen (Zeitgeber)

4	TA LO	TAL_7	TAL_6	TAL_5	TAL_4	TAL_3	TAL_2	TAL_1	TAL_0
5	TA HI	TAH_7	TAH_6	TAH_5	TAH_4	TAH_3	TAH_2	TAH_1	TAH_0
6	TB LO	TBL_7	TBL_6	TBL_5	TBL_4	TBL_3	TBL_2	TBL_1	TBL_0
7	TB HI	TBH_7	TBH_6	TBH_5	TBH_4	TBH_3	TBH_2	TBH_1	TBH_0

Schreiben (Voreinsteller)

4	TA LO	PAL_7	PAL_6	PAL_5	PAL_4	PAL_3	PAL_2	PAL_1	PAL_0
5	TA HI	PAH_7	PAH_6	PAH_5	PAH_4	PAH_3	PAH_2	PAH_1	PAH_0
6	TB LO	PB_7	PB_6	PB_5	PB_4	PB_3	PB_2	PB_1	PB_0
7	TB HI	PBH_7	PBH_6	PBH_5	PBH_4	PBH_3	PBH_2	PBH_1	PBH_0

Tageszeit Uhr (TOD)

Bei der TOD Uhr handelt es sich um einen speziellen Zeitgeber für Anwendungen in Echtzeit. TOD enthält eine 24 Stunden (AM/PM) Uhr mit einer Auflösung von 1/10 Sekunde. Es ist in vier Register unterteilt: 1/10 Sekunden, Sekunden, Minuten und Stunden. Die AM/PM Flag befindet sich im MSB (most significant bit) des Stundenregisters und ist einem Test leicht zugänglich. Jedes Register wird im BCD Format ausgegeben, um eine Umwandlung bei Ansteuerung von Anzeigen o.ä. zu erleichtern. Die Uhr benötigt für eine genaue Zeitangabe eine externe 60 Hertz oder 50 Hertz (programmierbar) TTL Eingabe am TOD Anschluß. Neben der Zeitangabe steht ein programmierbarer ALARM zur Verfügung, der zu einer gewünschten Zeit ein Interrupt erzeugt. Die ALARM Register befinden sich in derselben Adresse wie die entsprechenden TOD Register. Der Zugang zum ALARM ist durch ein Bit des Kontrollregisters geregelt. Bei ALARM handelt es sich um ein Nurschreib-Register. Jeder Lesevorgang einer TOD Adresse gibt unabhängig vom Zustand des ALARM Zugangsbits die Zeit aus.

Man muß eine spezifische Ereignisabfolge einhalten, um TOD richtig zu setzen und zu lesen. Jedesmal, wenn in das Stundenregister geschrieben wird, bleibt der TOD automatisch stehen. Die Uhr startet erst nach einem Schreibvorgang in das 1/10 Sekunden Register. Dadurch wird sichergestellt, daß TOD immer zur gewünschten Zeit startet. Da während einer Leseoperation jederzeit ein Übertrag von einer Stelle der Zeitangabe zur nächsten erfolgen kann, ist eine Zwischenspeicherfunktion eingebaut, um die gesamte Tageszeitinformation während einer Lesesequenz konstant zu halten. Alle vier TOD Register werden beim Lesen der Stunden zwischengespeichert und bleiben dies, bis die 1/10 Sekunden ausgelesen sind. Während die Ausgaberegister zwischengespeichert bleiben, zählt die TOD Uhr weiter. Soll nur ein Register gelesen werden, besteht kein Übertragsproblem und die Register können gewissermaßen im Vorübergehen gelesen werden. Das gilt nur unter der Voraussetzung, daß jedem Lesevorgang, der die Stunden betrifft, ein Lesen der 1/10 Sekunden folgt, um die Zwischenspeicherung wieder abzuschalten.

Lesen

REG	NAME								
8	TOD 10THS	0	0	0	0	T_8	T_4	T_2	T_1
9	TOD SEC	0	SH_4	SH_2	SH_1	SL_8	SL_4	SL_2	SL_1
A	TOD MIN	0	MH_4	MH_2	MH_1	ML_8	ML_4	ML_2	ML_1
B	TOD HR	PM	0	0	HH	HL_8	HL_4	HL_2	HL_1

Schreiben

CRB: Bit 7 = 0, TOD eingeben
CRB: Bit 7 = 1, ALARM eingeben
(gleiches Format wie bei Lesen)

Serieller Kanal (SDR)

Beim seriellen Kanal handelt es sich um ein gepuffertes, synchrones Schieberegister von 8-Bit Breite. Ein Kontrollbit bestimmt den Eingabe- oder Ausgabemodus. Beim Eingabemodus werden Daten, die am SP Anschluß anliegen, während einer positiven Signalflanke am Anschluß CNT in das Schieberegister geschoben. Nach acht CNT Pulsen werden die Daten des Schieberegisters in das serielle Datenregister befördert und ein Interrupt erzeugt. Im Ausgabemodus wird der Zeitgeber A als Baudraten-Generator benutzt. Die Daten werden am Anschluß SP mit der halben Underflow Rate des Zeitgeber A herausgeschoben. Die größtmögliche Baudrate ist gleich der von Phase 2 geteilt durch 4. Die maximal nutzbare Baudrate wird jedoch durch die Leitungsqualität und durch die Geschwindigkeit, mit der der Empfänger auf die Dateneingabe antwortet, bestimmt. Die Übertragung beginnt nach einem Schreibvorgang in das serielle Datenregister (vorausgesetzt, Zeitgeber A läuft und befindet sich in der kontinuierlichen Betriebsart). Das vom Zeitgeber A abgeleitete Zeitsignal erscheint als Ausgabe am Anschluß CNT. Die Daten des seriellen Datenregisters werden in das Schieberegister geladen und bei Eintreffen eines CNT Pulses am SP Anschluß herausgeschoben. Herausgeschobene Daten werden bei negativer Flanke von CNT gültig und bleiben dies bis zur nächsten negativen Flanke. Nach acht CNT Pulsen wird ein Interrupt erzeugt, um anzuzeigen, daß mehr Daten gesendet werden können. Wenn das serielle Datenregister vor diesem Interrupt mit neuer Information geladen wurde, werden diese neuen Daten automatisch in das Schieberegister geladen und die Übertragung wird fortgesetzt. Solange der Mikroprozessor ein Byte vor dem Schieberegister bleibt, ist die Übertragung kontinuierlich. Sind nach dem achten CNT Puls keine weiteren Daten zu übertragen, kehrt CNT in den oberen Zustand zurück und SP bleibt auf dem Niveau des letzten übertragenen Datenbit stehen. SDR Daten werden mit dem MSB voran herausgeschoben und serielle Eingabedaten sollten ebenfalls in diesem Format erscheinen.

Die bidirektionale Eigenschaft von seriellem Kanal und CNT-Clock ermöglicht es, viele 6526 Einheiten an einen gemeinsamen seriellen Kommunikationsbus anzuschließen. Dabei arbeitet ein 6526 als Steuergerät, das die Daten und das Schiebesignal liefert, während alle anderen 6526 Bausteine Empfangsgeräte sind. Sowohl die CNT als auch die SP Ausgänge sind in „open drain" Bauart ausgeführt, um einen solchen gemeinsamen Bus zu ermöglichen. Das Protokoll zur Auswahl des Steuer- oder Empfangsgeräts kann über den seriellen Bus oder über bestimmte Quittungsleitungen übermittelt werden.

REG	NAME								
C	SDR	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

Interruptkontrolle (ICR)

Es gibt beim 6526 fünf Quellen für Interrupts: Underflow vom Zeitgeber A, Underflow vom Zeitgeber B, TOD ALARM, Serieller Kanal voll/leer und $\overline{\text{FLAG}}$. Ein einzelnes Register erlaubt eine Maskierung und liefert Interruptinformation. Das Interrupt Kontrollregister besteht aus einem Nurschreib-MASK-Register und einem Nurlese-DATA-Register. Jeder Interrupt setzt

im DATA Register ein entsprechendes Bit. Jedes Interrupt, das durch das MASK Register erlaubt ist, setzt das IR Bit (MSB) des DATA Registers und zieht den $\overline{IRQ}$ Anschluß nach unten. Bei einem Multichip System kann das IR Bit durch Abfrage feststellen, welcher Baustein die Interruptanforderung erzeugt hat. Das Interrupt Datenregister wird gelöscht, die $\overline{IRQ}$ Leitung geht nach dem Lesen des Datenregisters nach oben. Da jedes Interrupt unabhängig von MASK ein Interruptbit setzt und jedes Interrupt selektiv maskiert werden kann, um eine Prozessorunterbrechung zu vermeiden, ist es möglich, ausgewählte Interrupts mit echten Interrupts zu mischen. Auf der anderen Seite führt eine Auswahl des IR Bit zu einem Löschen des Datenregisters. Deshalb bleibt es Aufgabe des Benutzers, die Information, die das Datenregister enthält, zu sichern, wenn irgendwelche ausgewählten Interrupts vorlagen.

Mit dem MASK Register kann man die einzelnen MASK Bits einfach kontrollieren. Beim Schreiben in das MASK Register werden die hineingeschriebenen Daten folgendermaßen verändert. Handelt es sich beim Bit 7 (SET/$\overline{CLEAR}$) der hineingeschriebenen Daten um eine 0, wird jedes MASK Bit, das als 1 hineingeschrieben wurde, gelöscht, während diejenigen MASK Bits, die mit einer 0 hineingeschrieben wurden, unverändert bleiben. Handelt es sich beim Bit 7 der hineingeschriebenen Daten um eine 1, wird jedes MASK Bit, das hineingeschrieben wurde, gesetzt, während diejenigen MASK Bits, die mit einer 0 hineingeschrieben wurden, unverändert bleiben. Soll eine Interruptflag IR setzen und damit ein Interrupt anfordern, muß das entsprechende MASK Bit gesetzt sein.

Lesen (INT Daten)

REG	NAME								
D	ICR	IR	0	0	FLG	SP	ALRM	TB	TA

Schreiben (INT Maske)

REG	NAME								
D	ICR	S/$\overline{C}$	X	X	FLG	SP	ALRM	TB	TA

Kontrollregister

Im 6526 gibt es zwei Kontrollregister, CRA und CRB. CRA ist dem Zeitgeber A und CRB dem Zeitgeber B zugeordnet. Das Registerformat sieht folgendermaßen aus:

CRA:

Bit	Name	Funktion
0	START	1 = START Zeitgeber A, 0 = STOP Zeitgeber A. Dieses Bit wird automatisch im Monovibratormodus beim Underflow zurückgesetzt.
1	PBON	1 = Zeitgeber A Ausgabe erscheint an PB6, 0 = PB6 normale Arbeitsweise.
2	OUTMODE	1 = Umschalten, 0 = Puls
3	RUNMODE	1 = Einzelpuls, 0 = Kontinuierlich
4	LOAD	1 = Erzwungenes Laden (das ist der STROBE Eingang, keine Datenspeicherung, Bit 4 liest immer eine 0 zurück und das Schreiben einer 0 führt zu keiner Veränderung).
5	INMODE	1 = Zeitgeber A zählt positive CNT Übergänge, 0 = Zeitgeber A zählt phi2 Pulse.
6	SPMODE	1 = Ausgang serieller Kanal (CNT liefert Clock Schiebepuls), 0 = Eingang serieller Kanal (externer Clock Schiebepuls erforderlich).
7	TODIN	1 = 50 Hz Clock Puls an TOD Anschluß für genaue Zeit erforderlich, 0 = 60 Hz Clock Puls an TOD Anschluß für genaue Zeit erforderlich.

CRB:

Bit	Name	Funktion
		(Bits CRB0-CRB4 sind bei Zeitgeber B identisch mit CRA0-CRA4 mit der Ausnahme, daß Bit 1 die Ausgabe von Zeitgeber B an PB7 kontrolliert)
5,6	INMODE	Die Bits CRB5 und CRB6 wählen für Zeitgeber B eine von vier Eingangsmodi aus: **CRB6 CRB5** 0 0 Zeitgeber B zählt Phase 2 Pulse 0 1 Zeitgeber B zählt positive CNT Pulse 1 0 Zeitgeber B zählt Zeitgeber A Underflow Pulse 1 1 Zeitgeber B zählt Zeitgeber A Underflow Pulse, wenn CNT oben.
7	ALARM	1 = Schreiben in TOD Register setzt ALARM 0 = Schreiben in TOD Register setzt TOD Clock

REG	NAME	TOD IN	SP MODE	IN MODE	LOAD	RUN MODE	OUT MODE	PB ON	START
E	CRA	0 = 60Hz 1 = 50Hz	0 = INPUT 1 = OUTPUT	0 = ϕ2 1 = CNT	1 = FORCE LOAD (STROBE)	0 = CONT 1 = O.S.	0 = PULSE 1 = TOGGLE	0 = PB_6 OFF 1 = PB_6 ON	0 = STOP 1 = START

TA

REG	NAME	ALARM	IN	MODE	LOAD	RUN MODE	OUT MODE	PB ON	START
F	CRB	0 = TOD 1 = ALARM	0 1 1 1	0 = ϕ2 1 = CNT 0 = TA 1 = CNT-TA	1 = FORCE LOAD (STROBE)	0 = CONT. 1 = O.S.	0 = PULSE 1 = TOGGLE	0 = PB_7 OFF 1 = PB_7 ON	0 = STOP 1 = START

TB

Alle unbenutzten Register Bits bleiben bei einem Schreibvorgang unverändert und werden bei einem Lesevorgang auf 0 gesetzt.

Die COMMODORE HALBLEITERGRUPPE behält sich das Recht von Produktänderungen vor, welche die Zuverlässigkeit, die Funktion oder den Aufbau verbessern. Die COMMODORE HALBLEITERGRUPPE lehnt jede Haftung ab, die durch Anwendung oder Benutzung eines Produktes oder Schaltkreises entstehen kann, der hier beschrieben ist. Es wird weder die Lizenz aus den eigenen Patentrechten, noch werden die Rechte anderer überantwortet.

6566/6767 (VIC-II) Baustein Spezifikationen

Bei den 6566/6567 Bausteinen handelt es sich um Farb-Videocontroller, die veilseitig anwendbar sowohl für Computer Videoterminals als auch für Videospiele eingesetzt werden können. Beide Einheiten enthalten 47 Kontrollregister, die über einen standardmäßigen 8-Bit Mikroprozessorbus (65xx) angesprochen werden können. Diese können ihrerseits bis zu 16K Speicher für Anzeigeinformationen ansprechen. Im folgenden werden die unterschiedlichen Betriebsarten und die Auswahlmöglichkeiten innerhalb jeder Betriebsart beschrieben.

Zeichenanzeige-Modus

Im Zeichenanzeige-Modus holen sich die 6566/6567 die CHARACTER POINTER (Zeichenzeiger) aus dem Speicherbereich der VIDEO MATRIX und übersetzen diese Zeiger in Zeichenpunkt-Adressen innerhalb der 2048 Byte des Speicherbereichs, der CHARACTER BASE (Zeichenbasis). Die Videomatrix setzt sich aus 1000 aufeinanderfolgenden Speicherstellen zusammen, von denen jede einen 8-Bit Zeichenzeiger enthält. Die Speicherstelle der Videomatrix innerhalb des Speichers wird durch VM13-VM10 im Register 24 ($18) bestimmt,

die als die 4 MSB der Adresse der Videomatrix benutzt werden. Die niederwertigen 10 Bits werden durch einen internen Zähler zur Verfügung gestellt (VC3-VC0), der die 1000 Zeichenpositionen durchläuft. Man beachte, daß bei den 6566/6567 14 Adreßausgänge zur Verfügung stehen. Aus diesem Grund ist für eine vollständige Decodierung des Systemspeichers zusätzliche Hardware notwendig.

Zeichenzeiger-Adresse

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
VM13	VM12	VM11	VM10	VC9	VC8	VC7	VC6	VC5	VC4	VC3	VC2	VC1	VC0

Der 8-Bit Zeichenzeiger stellt gleichzeitig bis zu 256 unterschiedliche Zeichendefinitionen zur Verfügung. Jedes Zeichen ist als eine 8x8 Punktmatrix in Form von acht aufeinanderfolgenden Bytes in der Zeichenbasis gespeichert. Die Speicherstelle der Zeichenbasis wird durch CB13-CB11 bestimmt, außerdem wird sie im Register 24 ($18) festgelegt, das für die 3 MSB der Zeichenbasis Adresse benutzt wird. Die 11 niederwertigen Adressen werden durch einen 8-Bit Zeichenzeiger aus der Videomatrix (D7-D0) gebildet, der ein einzelnes Zeichen auswählt, und durch einen 3-Bit Rasterzähler (RC2-RC0), der eines der 8 Zeichenbytes auswählt. Die sich daraus ergebenden Zeichen werden zu 25 Zeilen von jeweils 40 Zeichen angeordnet. Zusätzlich zu dem 8-Bit Zeichenzeiger ist jeder Position der Videomatrix ein 4-Bit COLOUR NYBBLE zugeordnet (der Videomatrix Speicher muß 12 Bits breit sein), welches eine von 16 Farben eines jeden Zeichens definiert.

Zeichendaten-Adresse

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
CB13	CB12	CB11	D7	D6	D5	D4	D3	D2	D1	D0	RC2	RC1	RC0

Standardzeichen Modus (MCM = BMM = ECM = 0)

Im Standardzeichen Modus werden die 8 aufeinanderfolgenden Bytes der Zeichenbasis direkt auf den 8 Zeilen einer jeden Zeichenregion dargestellt. Ein Bit „0" führt zur Hintergrundfarbe 0 (von Register 33 ($21)), während die durch das COLOR NYBBLE ausgewählte Farbe (Vordergrund) durch ein Bit „1" angezeigt wird (siehe Farbcode Tabelle).

Funktion	Zeichenbit	Angezeigte Farbe
Hintergrund	0	Hintergrund #0 Farbe (Register 33 ($21))
Vordergrund	1	Farbe durch 4-Bit COLOR NYBBLE bestimmt

Dadurch hat jedes Zeichen eine eindeutige Farbe, die durch das COLOR NYBBLE bestimmt wird (1 von 16), und alle Zeichen haben den gleichen gemeinsamen Hintergrund.

Multi-Color Zeichenmodus (MCM = 1, BMM = ECM = 0)

Durch den Multi-Color Modus wird eine zusätzliche Flexibilität erreicht, die bis zu 4 Farben innerhalb eines jeden Zeichens erlaubt, allerdings mit reduzierter Auflösung. Der Multi-Color Modus wird gewählt, indem man das MCM Bit in Register 22 ($16) auf 1 setzt. Dadurch werden die Punktdaten, die in der Zeichenbasis gespeichert sind, andersartig interpretiert. Ist das MSB des COLOR NYBBLE eine 0, wird das Zeichen dargestellt wie unter dem Modus Standardzeichen beschrieben. Dadurch ist es möglich, die beiden Modi zu mischen (allerdings stehen nur die niederwertigen 8 Farben zur Verfügung). Ist das MSB des COLOR NYBBLE eine 1 (wenn MCM:MSB(CM) = 1), werden die Zeichen im Multi-Color Modus dargestellt:

Funktion	Zeichen Bitpaar	Angezeigte Farbe
Hintergrund	00	Hintergrundfarbe #0 (Register 33 ($21))
Hintergrund	01	Hintergrundfarbe #1 (Register 34 ($22))
Vordergrund	10	Hintergrundfarbe #2 (Register 35 ($23))
Vordergrund	11	Farbe durch 3 LSB des COLOR NYBBLE bestimmt

Da zwei Bit nötig sind, um eine Punktfarbe zu bestimmen, wird das Zeichen nun in einer 4x8 Matrix dargestellt, wobei jeder Punkt die doppelte horizontale Ausdehnung des Standardmodus hat. Man beachte aber, daß jede Zeichenregion jetzt vier verschiedene Farben enthalten kann, zwei als Vordergrund und zwei als Hintergrund (siehe MOB Priorität).

Erweiterter Color Modus (ECM = 1, BMM = MCM = 0)

Der erweiterte Color Modus erlaubt die Wahl zwischen individuellen Hintergrundfarben für jede Zeichenregion in der normalen 8x8 Zeichenauflösung. Diese Betriebsart wird durch Setzen des ECM Bit aus Register 17 ($11) auf „1“ ausgewählt. Die Zeichenpunktdaten werden wie im Standardmodus dargestellt (die Vordergrundfarbe, die durch das COLOR NYBBLE bestimmt wird, wird bei einem Datenbit „1“ angezeigt). Die 2 MSB des Zeichenzeigers werden jedoch dazu benutzt, die Hintergrundfarbe für jede Zeichenregion auszuwählen:

Zeichenzeiger MSB Bitpaar	Angezeigte Hintergrundfarbe für 0 Bit
00	Hintergrundfarbe #0 (Register 33 ($21))
01	Hintergrundfarbe #1 (Register 34 ($22))
10	Hintergrundfarbe #2 (Register 35 ($23))
11	Hintergrundfarbe #3 (Register 36 ($24))

Da die beiden MSB des Zeichenzeigers für Farbinformationen benutzt werden, stehen nur 64 verschiedene Zeichendefinitionen zur Verfügung. Die 6566/6567 zwingen CB10 und CB9 auf

„0", unabhängig vom ursprünglichen Zeigerwert, sodaß lediglich die ersten 64 Zeichendefinitionen zugänglich sind. Mit dem erweiterten Color Modus hat jedes Zeichen eine von sechzehn individuell definierten Vordergrundfarben und eine der vier zur Verfügung stehenden Hintergrundfarben.

Bemerkung: Der erweiterte Color Modus und der Multi-Color Modus sollten nicht gleichzeitig zugelassen werden.

Bitmap Modus

Beim Bitmap Modus holen sich die 6566/6567 die Daten auf eine andere Weise aus dem Speicher. Dadurch entsteht eine eindeutige Entsprechung zwischen jedem angezeigten Punkt und einem Speicherbit. Im Bitmap Modus steht eine Bildschirmauflösung von 320H x 200V individuell kontrollierbaren Punkten zur Verfügung. Der Bitmap Modus wird durch Setzen des BMM Bit im Register 17 ($11) auf „1" ausgewählt. Die VIDEO MATRIX wird genauso wie im Zeichenmodus angegangen, die Videomatrixdaten werden jedoch nicht mehr als Zeichenzeiger, sondern als Farbdaten interpretiert. Der VIDEO MATRIX COUNTER (Videomatrixzähler) wird dabei ebenfalls als Adresse benutzt, um die Punktdaten aus der 8000-Byte großen DISPLAY BASE für die Anzeige zu holen. Die DISPLAY BASE Adresse wird folgendermaßen gebildet:

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
CB13	VC9	VC8	VC7	VC6	VC5	VC4	VC3	VC2	VC1	VC0	RC2	RC1	RC0

VCx bezeichnet die Ausgaben des Videomatrixzählers. RCx bezeichnet den 3-Bit Rasterzeilenzähler und CB13 stammt von Register 24 ($18). Der Videomatrixzähler durchläuft für 8 Rasterzeilen die gleichen 40 Speicherstellen, um nach jeweils 8 Rasterzeilen zu den nächsten 40 Speicherstellen überzugehen, während der Rasterzähler nach jeder horizontalen Rasterlinie um 1 erhöht wird. Diese Adressierung erfolgt bei jeder der 8 aufeinander folgenden Speicherzellen, die als ein 8x8 Punktblock auf dem Bildschirm formatiert werden.

Standard Bitmap Modus (BMM = 1, MCM = 0)

Bei Benutzung des Standard Bitmap Modus wird die Farbinformation ausschließlich von den Daten abgeleitet, die in der Videomatrix gespeichert sind (das Color Nybble wird nicht verwendet). Die 8 Bits werden in zwei 4-Bit Nybbles unterteilt. Dadurch kann man unabhängig zwei Farben in jedem 8x8 Punktblock wählen. Wenn ein Bit im Anzeigespeicher „0" ist, wird die Farbe des ausgegebenen Punktes durch das niederwertige Nybble (LSN) gesetzt. Genauso wählt ein Bit „1" des Anzeigespeichers die Ausgabefarbe, die durch das MSN (oberes Nybble) bestimmt wird.

Bit	Anzeigefarbe
0	unteres Nybble des Videomatrixzeigers
1	oberes Nybble des Videomatrixzeigers

Multi-Color Bitmap Modus (BMM = MCM = 1)

Der Multi-Color Bitmap Modus wird durch Setzen des MCM Bit in Register 22 ($16) auf „1" in Verbindung mit dem BMM Bit ausgewählt. Der Multi-Color Modus geht den Speicher in der gleichen Folge an wie der Standard Bitmap Modus, interpretiert die Punktdaten jedoch folgendermaßen:

Bit Paar	Angezeigte Farbe
00	Hintergrundfarbe #0 (Register 33 ($21))
01	Oberes Nybble des Videomatrixzeigers
10	Unteres Nybble des Videomatrixzeigers
11	Videomatrix Color Nybble

Man beachte, daß das Color Nybble (DB11-DB8) für den Multi-Color Bitmap Modus genauso benutzt wird, wie zwei Bits benutzt werden, um eine Punktfarbe auszuwählen. Die horizontale Punktausdehnung wird verdoppelt, was zu einer Bildschirmauflösung von 160H x 200V führt. Bei der Benutzung des Multi-Color Bitmap Modus können zusätzlich zu der Hintergrundfarbe in jedem 8x8 Block drei unabhängig wählbare Farben dargestellt werden.

Verschiebbare Objektblöcke

Der verschiebbare Objektblock (movable object block, MOB) ist ein spezieller Zeichentyp, der an jeder Position des Bildschirms dargestellt werden kann. Dabei gelten nicht die Einschränkungen, die die Blocks bei der Zeichen- und Bitmap-Darstellung bilden. Bis zu 8 eigene MOBs können gleichzeitig dargestellt werden. Jeder wird durch 63 Bytes im Speicher definiert, die als eine 24x21 Punktanordnung angezeigt werden (siehe unten). MOBs eignen sich aufgrund ihrer spezifischen Eigenschaften besonders für Videographik und Spielanwendungen.

MOB Anzeigeblock

Byte	Byte	Byte
00	01	02
03	04	05
.	.	.
.	.	.
.	.	.
57	58	59
60	61	62

Zulassen

Jeder MOB kann selektiv zur Anzeige dadurch zugelassen werden, daß das ihm entsprechende Zulassungsbit (enable bit, MnE) in Register 21 ($15) auf „1" gesetzt wird. Ist das MnE Bit auf „0", sind für den nicht zugelassenen MOB keine Operationen möglich.

Positionieren

Jeder MOB wird über seine X und Y Positionsregister (siehe Registerplan) mit einer Auflösung von 512 Positionen horizontal und 256 vertikal positioniert. Die Position eines MOB wird durch die obere linke Ecke der Anordnung bestimmt. Die X Positionen 23 bis 347 ($17-$157) und Y Positionen 50 bis 249 ($32-$F9) sind sichtbar. Da nicht alle zur Verfügung stehenden MOB Positionen auf dem Bildschirm vollständig sichtbar sind, können MOBs weich in den Bildschirm hinein und aus diesem heraus bewegt werden.

Färben

Jeder MOB besitzt ein eigenes 4-Bit Register, das die MOB Farbe bestimmt. Die beiden MOB Farbmodi sind:

Standard MOB (MnMC = 0)

Im Standard Modus führt ein „0" Bit bei den MOB Daten dazu, daß die Hintergrundfarben durchscheinen (transparent). Ein „1" Bit wird als MOB Farbe dargestellt, die durch das entsprechende MOB Farbregister bestimmt ist.

Multi-Color MOB (MnMC = 1)

Jeder MOB kann über MnMC Bits im MOB Multi-Color Register 28 ($1C) individuell als ein Multi-Color MOB ausgewählt werden. Ist das MnMC Bit „1", wird das entsprechende MOB im Multi-Color Modus angezeigt. Im Multi-Color Modus werden die MOB Daten paarweise (wie bei anderen Multi-Color Modi) folgendermaßen interpretiert:

Bitpaar	Angezeigte Farbe
00	Transparent
01	MOB Multi-Color #0 (Register 37 ($25))
10	MOB Farbe (Register 39-46 ($27-$2E))
11	MOB Multi-Color #1 (Register 38 ($26))

Da zwei Bits für jede Farbe notwendig sind, wird die Auflösung des MOB auf 12x21 reduziert, wobei jeder horizontale Punkt um seine Standardgröße verdoppelt wird, sodaß sich die Gesamtgröße des MOB nicht verändert. Man beachte, daß bis zu 3 Farben in jedem MOB angezeigt werden können (zusätzlich zu transparent), zwei der Farben werden jedoch zwischen allen MOBs im Multi-Color Modus aufgeteilt.

Vergrößern

Jeder MOB kann unabhängig vom anderen sowohl in horizontaler als auch vertikaler Richtung ausgedehnt werden (2x). Zwei Register enthalten die Kontrollbits (MnXE, MnYE) für die Vergrößerung:

Register	Funktion
23($17)	Horizontale Vergrößerung MnXE „1" = vergrößert „0" = normal
29($1D)	Vertikale Vergrößerung MnXE „1" = vergrößert „0" = normal

Wenn die MOBs vergrößert sind, erreicht man allerdings keine größere Auflösung. Die gleiche 24x21 Anordnung wird angezeigt (12x21 bei Multi-Color). Die gesamte MOB Ausdehnung ist jedoch in der gewünschten Richtung verdoppelt (der kleinste MOB-Punkt beträgt bis zu 4x der Dimension eines Standardpunktes, wenn der MOB sowohl mehrfarbig, also Multi-Color, als auch vergrößert ist).

Priorität

Die Priorität eines jeden MOB kann man mit Rücksicht auf die anderen angezeigten Informationen, sei es Zeichenmodus oder Bit Map Modus, individuell kontrollieren. Die Priorität jedes MOB wird durch das entsprechende Bit (MnDP) des Registers 27 ($1B) folgendermaßen gesetzt:

Reg Bit	Priorität für Zeichen- oder Bit Map Daten
0	Nicht transparente MOB Daten werden angezeigt (MOB vorangestellt)
1	Nicht transparente MOB Daten werden nur dargestellt anstelle von Hintergrund #0 oder Multi-Color Bit Paar 01 (MOB nachgestellt)

MOB-Anzeige Datenpriorität

MnDP = 1	MnDP = 0
MOBn	Vordergrund
Vordergrund	MOBn
Hintergrund	Hintergrund

Die MOB Daten Bits von „0" („00" in Multi-Color Modus) sind transparent und erlauben immer die Anzeige anderer Information.

Die MOBs haben untereinander eine festgelegte Priorität, wobei MOB 0 die höchste und MOB 7 die niedrigste Priorität hat. Wenn zwei MOB Daten (ausgenommen transparente Daten) übereinstimmen, werden die Daten des MOB mit der niedrigeren Nummer angezeigt. Die Prioritätsfolge sieht folgendermaßen aus: MOB, MOB Daten, Zeichendaten, Bit Map Daten.

Kollisionsfeststellung

Zwei Arten von MOB Kollision (Koinzidenz) werden festgestellt. MOB-MOB Kollision und MOB-Anzeigedaten Kollision:

1. Eine Kollision zwischen zwei MOBs liegt dann vor, wenn die nichttransparenten Ausgabedaten von zwei MOBs zusammenfallen (Koinzidenz). Die Koinzidenz von transparenten MOB Bereichen erzeugt keine Kollision. Wenn eine Kollision vorliegt, werden die MOB Bits (MnM) im MOB-MOB COLLISION Register 30 ($1E) für beide kollidierenden MOBs auf „1" gesetzt. Wenn eine Kollision zwischen zwei oder mehreren MOBs stattfindet, wird das MOB-MOB Kollisionsbit für jeden kollidierenden MOB gesetzt. Die Kollisionsbits bleiben solange gesetzt, bis ein Lesevorgang des Kollisionsregisters erfolgte, bei dem alle Bits automatisch gelöscht werden. MOB Kollisionen werden sogar dann festgestellt, wenn sie außerhalb des Bildschirmbereichs liegen.

2. Bei der zweiten Art von Kollision handelt es sich um eine MOB-Daten Kollision zwischen einem MOB und den angezeigten Vordergrunddaten des Zeichen- oder Bitmap-Modus. Das MOB-DATA COLLISION Register 31 ($1F) besitzt für jeden MOB ein Bit (MnD), das auf „1" gesetzt wird, wenn der MOB und die angezeigten Nicht-Hintergrund Daten zusammenfallen. Wiederum führt nur die Koinzidenz von transparenten Daten zu keiner Kollision. Bei speziellen Anwendungen verursachen außerdem die angezeigten Daten des 0-1 Multi-Color Bitpaares keine Kollision. Durch diese Eigenschaft kann man Hintergrunddaten anzeigen, ohne daß diese mit echten MOB Kollisionen interferieren. Eine MOB-Datenkollision kann in horizontaler Richtung außerhalb des Bildschirms erfolgen, wenn die aktuellen angezeigten Daten in eine Position außerhalb des Bildschirms gerutscht sind (siehe scrolling). Bei einem Lesevorgang werden auch die MOB-DATA COLLISION Register automatisch gelöscht.

Die Zwischenspeicher für Kollisionsinterrupt werden immer dann gesetzt, wenn das erste Bit eines der Register auf „1" gesetzt wurde. Jedesmal, wenn ein Kollisionsbit in einem Register hoch gesetzt wurde, setzen darauf folgende Kollisionen solange nicht den Interruptzwischenspeicher, bis dieses Kollisionsregister in allen Bit durch einen Lesevorgang gelöscht wurde.

MOB Speicherzugriff

Die Daten eines jeden MOB werden in 63 aufeinanderfolgenden Bytes des Speichers abgelegt. Jeder MOB Datenblock ist durch einen MOB Zeiger definiert, der am Ende der VIDEO MATRIX liegt. Bei der normalen Anzeigeart werden lediglich 1000 Bytes der Videomatrix benutzt. Dadurch lassen sich die Matrixplätze 1016-1023 (VM Basis + $3F8 bis VM Basis + $3FF) für die MOB Zeiger 0-7 jeweils benutzen. Der 8-Bit MOB Zeiger aus der Videomatrix bestimmt zusammen mit den 6 Bit des MOB Bytezähler (zur Adresse 63 Bytes) das gesamte 14-Bit Adreßfeld:

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
MP7	MP6	MP5	MP4	MP3	MP2	MP1	MP0	MC5	MC4	MC3	MC2	MC1	MC0

Hierbei stellen MPx die MOB Zeigerbits aus der Video Matrix und MCx die intern erzeugten MOB Zählerbits dar. Die MOB Zeiger werden am Ende jeder Rasterlinie aus der Videomatrix gelesen. Wenn das Y Positionsregister eines MOB mit dem Zählwert der laufenden Rasterlinie übereinstimmt, werden die aktuellen MOB Daten geholt. Interne Zähler durchlaufen automatisch die 63 Bytes der MOB Daten, wobei sie 3 Bytes auf jeder Rasterlinie anzeigen.

Weitere Eigenschaften

Bildschirm löschen

Durch Setzen des DEN Bit in Register 17 ($11) auf „0" kann der Bildschirm gelöscht werden. Dabei wird der gesamte Bildschirm mit der externen Farbe gefüllt, die in Register 32 ($20) festgelegt ist. Ist der Löschvorgang aktiv, werden nur transparente (Phase 1) Speicherzugriffe benötigt, die dem Prozessor den vollen Zugriff auf den Systembus erlauben. MOB Daten hingegen sind nur dann zugänglich, wenn die MOBs nicht auch gesperrt sind. Für eine normale Videoanzeige muß das DEN Bit auf „1" gesetzt sein.

Zeilen/Spalten Auswahl

Eine normale Anzeige besteht aus 25 Zeilen zu jeweils 40 Zeichen (oder Zeichenbereichen). Für spezielle Anwendungen kann das Anzeigefenster auf 24 Zeilen und 38 Zeichen verkleinert werden. Dabei wird das Format der angezeigten Information nicht verändert, es sei denn, daß Zeichen (Bits), die an den äußeren Randbereich angrenzen, jetzt durch den Rand abgedeckt werden. Die Auswahlbits haben folgende Wirkung:

RSEL	Zeilenzahl	CSEL	Spaltenzahl
0	24 Zeilen	0	38 Spalten
1	25 Zeilen	1	40 Spalten

Das RSEL Bit befindet sich in Register 17 ($11) und das CSEL Bit in Register 22 ($16). Normalerweise wird für die Standardanzeige das größere Anzeigefenster benutzt, während das kleinere Fenster meist im Zusammenhang mit dem Scrolling (Bildschirmrollen) benutzt wird.

Bildschirmrollen

Die angezeigten Daten können um bis zu einen Zeichenraum sowohl in horizontaler als auch vertikaler Richtung gerollt werden. Wird es zusammen mit dem schmaleren Anzeigefenster (siehe oben) benutzt, kann man damit einen weicheren Bewegungsübergang der Anzeige erzeugen, da der Systemspeicher nur bei Anzeige einer neuen Zeichenzeile (oder Spalte) auf den neusten Stand gebracht werden muß. Rollen kann man auch zum Zentrieren einer festen Anzeige innerhalb des Anzeigefensters benutzen.

Bits	Register	Funktion
X2, X1, X0	22 ($16)	Horizontale Position
Y2, Y1, Y0	17 ($11)	Vertikale Position

Lichtstift

Der Lichtstifteingang speichert bei fallender Flanke die laufende Bildschirmposition in ein Registerpaar (LPX, LPY). Das X Positionsregister 19 ($13) enthält zum Zeitpunkt des Über-

gangs die 8 MSB der X Position. Da die X Position durch einen 9-Bit Zähler (512 Zustände) definiert ist, steht eine Auflösung von zwei horizontalen Punkten zur Verfügung. Ähnlich wird die Y Position in das Register 20 ($14) zwischengespeichert. Hier genügen 8 Bits, um eine Einzelrasterauflösung innerhalb der sichtbaren Anzeige zu gewährleisten. Der Zwischenspeicher für den Lichtstift kann nur einmal pro Bildaufbau getriggert werden. Darauf folgende Trigger innerhalb desselben Bildschirmaufbaus bleiben ohne Wirkung. Deshalb muß man in Abhängigkeit von den Eigenschaften des Lichtstifts mehrere Proben nehmen, bevor man diesen vom Bildschirm abhebt (im Durchschnitt drei oder mehr).

Rasterregister
Das Rasterregister hat zwei Funktionen. Beim Lesen des Registers 18 ($12) werden die niederwertigen 8 Bit der laufenden Rasterposition angegeben (die MSB-RC8 befinden sich in Register 17 ($11)). Das Rasterregister kann abgefragt werden, um Anzeigeveränderungen außerhalb des sichtbaren Bereichs zu implementieren. Damit läßt sich ein Anzeigeflimmern verhindern. Das sichtbare Anzeigefenster liegt zwischen Raster 51 und Raster 251 ($033-$0FB). Ein Schreibvorgang in die Rasterbits (einschließlich RC8) wird für einen internen Rastervergleich zwischengespeichert. Wenn das laufende Raster mit dem eingeschriebenen Wert übereinstimmt, wird der Rasterinterrupt Zwischenspeicher gesetzt.

Interruptregister

Das Interruptregister zeigt den Status der 4 Interruptquellen an. Ein Interruptzwischenspeicher wird in Register 25 ($19) dann auf „1" gesetzt, wenn eine Interruptquelle ein Interrupt angefordert hat. Die Interruptquellen sind:

Latch Bit	Enable Bit	Gesetzt, wenn
IRST	ERST	gesetzt, wenn (Rasterzähler) = (gespeicherte Rasternummer)
IMDC	EMDC	Gesetzt durch MOB-DATA Kollisionsregister (nur erste Kollision)
IMMC	EMMC	Gesetzt durch MOB-MOB Kollisionsregister (nur erste Kollision)
ILP	ELP	Gesetzt durch negativen Übergang an LP Eingang (einmal pro Bildaufbau)
IRQ		Hoch gesetzt durch Zwischenspeicher gesetzt und zugelassen (Umkehrung von IRQ/Ausgang)

Um eine Interruptanfrage zu erlauben, damit der IRQ/Ausgang auf „0" gesetzt wird, muß das entsprechende Interruptzulassungsbit in Register 26 ($1A) auf „1" gesetzt werden. Wenn ein Interruptzwischenspeicher gesetzt wurde, kann er nur dadurch gelöscht werden, daß eine „1" in den gewünschten Zwischenspeicher in das Interruptregister geschrieben wird. Durch diese Eigenschaft kann man selektiv Videointerrupts handhaben, ohne dafür Software einzusetzen, die an die aktiven Interrupts erinnert.

Dynamische Auffrischung des RAM

In die 6566/6567 Einheiten ist ein dynamischer RAM-Auffrischungskontroller eingebaut. Nach jeder Rasterlinie werden fünf 8-Bit Zeilenadressen aufgefrischt. Durch diese Rate wird

eine maximale Verzögerung von 2,02 ms zwischen den Auffrischungszyklen jeder einzelnen Zeilenadresse innerhalb eines 128 Auffrischungsschemas garantiert. (Die maximale Verzögerung bei einem 256 Adressen Auffrischungsschema beträgt 3,66 ms.) Diese Auffrischung ist für das System vollkommen transparent, da diese während der Phase 1 der System Clock erfolgt. Der 6567 erzeugt sowohl RAS/ als auch CAS/, die normalerweise direkt mit den dynamischen RAMs verbunden sind. RAS/ und CAS/ werden bei jeder Phase 2 und bei jedem Videodatenzugriff (einschließlich Auffrischung) erzeugt, sodaß sich eine externe Erzeugung von Clockpulsen erübrigt.

Reset

Das Reset Bit (RES) in Register 22 ($16) wird nicht für normale Operationen benutzt. Es sollte deshalb bei der Initialisierung des Videobausteins auf „0" gesetzt sein. Ist es auf „1" gesetzt, wird die gesamte Arbeit des Videochip aufgehoben, einschließlich Videoausgabe und Synchronisation, Speicherauffrischung und Systembuszugriff.

Betriebstheorie

Systeminterface

Die 6566/6567 Video-Kontrolleinheiten stehen mit dem Systemdatenbus auf spezielle Weise in Wechselwirkung. Ein 65xx System benutzt den Systembus nur während der Phase 2 (Clock oben) des Zyklus. Die 6566/6567 Einheiten nehmen auf diese Eigenschaft dadurch Rücksicht, daß sie den Systemspeicher normalerweise während der Phase 1 (Clock unten) des Clockzyklus ansprechen. Aus diesem Grund sind solche Operationen wie Holen von Zeichendaten und Speicherauffrischung für den Prozessor vollkommen transparent und verringern nicht den Durchsatz des Prozessors. Die Videobausteine stellen die Interface Kontrollsignale zur Verfügung, die für eine Aufteilung des Bus notwendig sind.

Die Videoeinheiten liefern das Signal AEC (adress enable control), das dazu benutzt wird, die Adressbustreiber des Prozessor zu sperren, um damit der Videoeinheit Zugang zum Adressbus zu gewährleisten. Im aktiven Zustand ist AEC unten. Dadurch kann es direkt an den AEC Eingang der 65xx Familie angeschlossen werden. Das AEC Signal wird normalerweise während der Phase 1 aktiviert, sodaß Prozessoroperationen nicht beeinträchtigt werden. Aufgrund dieser Busaufteilung müssen alle Speicherzugriffe innerhalb eines 1/2 Zyklus vollendet sein. Da die Videobausteine einen 1MHz Clockpuls liefern (der als Systemphase 2 benutzt werden muß), dauert ein Speicherzyklus 500 ns einschließlich Adressaufruf, Datenzugriff und Datenübertragung in die lesende Einheit.

Bestimmte Operationen des 6566/6567 benötigen Daten in einer schnelleren Rate, als sie bei einem Lesevorgang ausschließlich während der Zeit von Phase 1 zur Verfügung stehen. Es handelt sich dabei insbesondere um Operationen wie Zugriff zu den Zeichenzeigern aus der Videomatrix und das Holen von MOB Daten. Aus diesem Grund muß der Prozessor gesperrt werden und auf die Daten muß während der Phase 2 der Clock zugegriffen werden. Man kann dies über ein BA (bus available) Signal erreichen. Normalerweise liegt die BA Leitung

oben, wird aber während der Phase 1 nach unten geschaltet, um damit anzuzeigen, daß der Videobaustein einen Datenzugriff während der Phase 2 vornehmen will. Drei Phase 2 Zeitzyklen stehen dem Prozessor nach einem Herunterschalten von BA zur Verfügung, um jeden laufenden Speicherzugriff zu vollenden. Während der vierten Phase 2, nachdem BA heruntergeschaltet wurde, bleibt das AEC Signal während Phase 2 unten, wenn der Videobaustein Daten holt. Die BA Leitung ist normalerweise an den RDY Eingang eines 65xx Prozessors angeschlossen. Innerhalb des Anzeigefensters werden die Zeichenzeiger jeweils nach 8 Rasterlinien geholt und benötigen daher 40 aufeinanderfolgende Phase 2 Zugriffe, um die Videomatrixzeiger zu holen. Das Holen der MOB Daten benötigt 4 Speicherzugriffe:

Phase	Daten	Bedingung
1	MOB Zeiger	Jedes Raster
2	MOB Byte 1	Jedes Raster, wenn MOB angezeigt wird
1	MOB Byte 2	Jedes Raster, wenn MOB angezeigt wird
2	MOB Byte 3	Jedes Raster, wenn MOB angezeigt wird

Die MOB Zeiger werden während jeder anderen Phase 1 am Ende einer jeden Rasterlinie geholt. Nach Bedarf werden die zusätzlichen Zyklen dazu benutzt, um MOB Daten zu holen. Wiederum werden alle notwendigen Bus Kontrollsignale von den 6566/6567 Einheiten geliefert.

Speicher Interface

Die beiden Versionen des Video Interface Baustein, 6566 und 6567, unterscheiden sich in der Konfiguration des Adreßausgangs. Der 6566 besitzt 13 vollständig dekodierte Adressen, die direkt an den Adreßbus des Systems angeschlossen werden können. Der 6567 besitzt dagegen gemultiplexte Adressen, die direkt an 64K dynamische RAMs angeschlossen werden können. Die niederwertigen Adressbits, A06-A00 liegen an A06-A00 an, während RAS/ heruntergeschaltet wurde. Die höherwertigen Bits, A13-A08, liegen an A05-A00 an, während CAS/ heruntergeschaltet wurde. Bei den Anschlüssen A11-A07 des 6567 handelt es sich um statische Adreßausgänge, die direkt an einen konventionellen 16K (2Kx8) ROM angeschlossen werden können. (Die niederwertigen Adressen benötigen einen externen Zwischenspeicher)

REGISTERPLAN

ADRESSE		DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	BESCHREIBUNG
00	($00)	M0X7	M0X6	M0X5	M0X4	M0X3	M0X2	M0X1	M0X0	MOB 0 X-position
01	($01)	M0Y7	M0Y6	M0Y5	M0Y4	M0Y3	M0Y2	M0Y1	M0Y0	MOB 0 Y-position
02	($02)	M1X7	M1X6	M1X5	M1X4	M1X3	M1X2	M1X1	M1X0	MOB 1 X-position
03	($03)	M1Y7	M1Y6	M1Y5	M1Y4	M1Y3	M1Y2	M1Y1	M1Y0	MOB 1 Y-position
04	($04)	M2X7	M2X6	M2X5	M2X4	M2X3	M2X2	M2X1	M2X0	MOB 2 X-position
05	($05)	M2Y7	M2Y6	M2Y5	M2Y4	M2Y3	M2Y2	M2Y1	M2Y0	MOB 2 Y-position
06	($06)	M3X7	M3X6	M3X5	M3X4	M3X3	M3X2	M3X1	M3X0	MOB 3 X-position
07	($07)	M3Y7	M3Y6	M3Y5	M3Y4	M3Y3	M3Y2	M3Y1	M3Y0	MOB 3 Y-position
08	($08)	M4X7	M4X6	M4X5	M4X4	M4X3	M4X2	M4X1	M4X0	MOB 4 X-position
09	($09)	M4Y7	M4Y6	M4Y5	M4Y4	M4Y3	M4Y2	M4Y1	M4Y0	MOB 4 Y-position
10	($0A)	M5X7	M5X6	M5X5	M5X4	M5X3	M5X2	M5X1	M5X0	MOB 5 X-position
11	($0B)	M5Y7	M5Y6	M5Y5	M5Y4	M5Y3	M5Y2	M5Y1	M5Y0	MOB 5 Y-position
12	($0C)	M6X7	M6X6	M6X5	M6X4	M6X3	M6X2	M6X1	M6X0	MOB 6 X-position
13	($0D)	M6Y7	M6Y6	M6Y5	M6Y4	M6Y3	M6Y2	M6Y1	M6Y0	MOB 6 Y-position
14	($0E)	M7X7	M7X6	M7X5	M7X4	M7X3	M7X2	M7X1	M7X0	MOB 7 X-position
15	($0F)	M7Y7	M7Y6	M7Y5	M7Y4	M7Y3	M7Y2	M7Y1	M7Y0	MOB 7 Y-position
16	($10)	M7X8	M6X8	M5X8	M4X8	M3X8	M2X8	M1X8	M0X8	MSB of X-position
17	(S11)	RC8	ECM	BMM	DEN	RSEL	Y2	Y1	Y0	See text
18	($12)	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	Raster register
19	($13)	LPX8	LPX7	LPX6	LPX5	LPX4	LPX3	LPX2	LPX1	Light Pen X
20	($14)	LPY7	LPY6	LPY5	LPY4	LPY3	LPY2	LPY1	LPY0	Light Pen Y
21	($15)	M7E	M6E	M5E	M4E	M3E	M2E	M1E	M0E	MOB Enable
22	($16)	—	—	RES	MCM	CSEL	X2	X1	X0	See text
23	($17)	M7YE	M6YE	M5YE	M4YE	M3YE	M2YE	M1YE	M0YE	MOB Y-expand
24	($18)	VM13	VM12	VM11	VM10	CB13	CB12	CB11	—	Memory Pointers
25	($19)	IRQ	—	—	—	ILP	IMMC	IMBC	IRST	Interrupt Register
26	($1A)	—	—	—	—	ELP	EMMC	EMBC	ERST	Enable Interrupt

REGISTERPLAN

ADRESSE		DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	BESCHREIBUNG
27	($1B)	M7DP	M6DP	M5DP	M4DP	M3DP	M2DP	M1DP	M0DP	MOB-DATA Priority
28	($1C)	M7MC	M6MC	M5MC	M4MC	M3MC	M2MC	M1MC	M0MC	MOB Multicolor Sel
29	($1D)	M7XE	M6XE	M5XE	M4XE	M3XE	M2XE	M1XE	M0XE	MOB X-expand
30	($1E)	M7M	M6M	M5M	M4M	M3M	M2M	M1M	M0M	MOB-MOB Collision
31	($1F)	M7D	M6D	M5D	M4D	M3D	M2D	M1D	M0D	MOB-DATA Collision
32	($20)	—	—	—	—	EC3	EC2	EC1	EC0	Exterior Color
33	($21)	—	—	—	—	BOC3	BOC2	BOC1	BOC0	Bkgd #0 Color
34	($22)	—	—	—	—	B1C3	B1C2	B1C1	B1C0	Bkgd #1 Color
35	($23)	—	—	—	—	B2C3	B2C2	B2C1	B2C0	Bkgd #2 Color
36	($24)	—	—	—	—	B3C3	B3C2	B3C1	B3C0	Bkgd #3 Color
37	($25)	—	—	—	—	MM03	MM02	MM01	MM00	MOB Multicolor #0
38	($26)	—	—	—	—	MM13	MM12	MM11	MM10	MOB Multicolor #1
39	($27)	—	—	—	—	M0C3	M0C2	M0C1	M0C0	MOB 0 Color
40	($28)	—	—	—	—	M1C3	M1C2	M1C1	M1C0	MOB 1 Color
41	($29)	—	—	—	—	M2C3	M2C2	M2C1	M2C0	MOB 2 Color
42	($2A)	—	—	—	—	M3C3	M3C2	M3C1	M3C0	MOB 3 Color
43	($2B)	—	—	—	—	M4C3	M4C2	M4C1	M4C0	MOB 4 Color
44	($2C)	—	—	—	—	M5C3	M5C2	M5C1	M5C0	MOB 5 Color
45	($2D)	—	—	—	—	M6C3	M6C2	M6C1	M6C0	MOB 6 Color
46	($2E)	—	—	—	—	M7C3	M7C2	M7C1	M7C0	MOB 7 Color

Bemerkung: Ein Strich bedeutet kein Anschluß. Fehlende Anschlüsse werden als „1“ gelesen.

FARBCODES

D4	D3	D1	D0	HEX	DEC	FARBE
0	0	0	0	0	0	SCHWARZ
0	0	0	1	1	1	WEISS
0	0	1	0	2	2	ROT
0	0	1	1	3	3	KOBALTBLAU
0	1	0	0	4	4	PURPURROT
0	1	0	1	5	5	GRÜN
0	1	1	0	6	6	BLAU
0	1	1	1	7	7	GELB
1	0	0	0	8	8	ORANGE
1	0	0	1	9	9	BRAUN
1	0	1	0	A	10	HELLROT
1	0	1	1	B	11	DUNKELGRAU
1	1	0	0	C	12	MITTELGRAU
1	1	0	1	D	13	HELLGRÜN
1	1	1	0	E	14	HELLBLAU
1	1	1	1	F	15	HELLGRAU

6581 Ton Interface Einheit (SID), Bausteinspezifikationen

Aufbau

Beim 6581 Sound Interface Device (SID) handelt es sich um einen Einzelbaustein. Er enthält einen elektronischen dreistimmigen Musiksynthesizer und einen Toneffektgenerator, der zum 65xx und ähnlichen Mikroprozessorfamilien kompatibel ist. Der SID kontrolliert breitbandig und mit hoher Auflösung die Frequenz, die Tonfarbe (Oberwellengehalt) und die Lautstärke. Spezielle Kontrollschaltkreise minimieren den Softwareaufwand und erlauben damit den Einsatz in Arcade Videospielen und preiswerten Musikinstrumenten.

Eigenschaften

- 3 Tonoszillatoren
  Bereich: 0–4 kHz
- 4 Wellenformen pro Oszillator
  Dreieck, Sägezahn
  Variable Pulse, Rauschen
- 3 Amplitudenmodulatoren
  Bereich: 48 dB

- 3 Hüllgeneratoren
  Exponentielles Verhalten
  Anstiegsrate (attack): 2 ms - 8 s
  Abfallrate (decay): 6 ms - 24 s
  Halteniveau (sustain): 0 - max. Lautstärke
  Ausklingrate (release): 6 ms - 24 s
- Oszillatorsynchronisation
- Ringmodulation

Beschreibung

Der 6581 enthält drei Synthesizerstimmen, die sowohl unabhängig voneinander als auch gemeinsam (oder mit externen Tonquellen) benutzt werden können, um komplexe Klänge zu erzeugen. Jede Stimme enthält einen Tonoszillator/Wellenformgenerator, einen Hüllgenerator und einen Amplitudenmodulator. Der Tonoszillator kontrolliert über einen weiten Bereich die Stimmfrequenz. Der Oszillator erzeugt bei der gewählten Frequenz vier Wellenformen mit dem zugehörigen Oberwellenanteil für jede Wellenform. Dadurch läßt sich die Tonfarbe leicht kontrollieren. Die Lautstärke des Oszillators wird vom Amplitudenmodulator kontrolliert, der seinerseits dem Hüllgenerator unterworfen ist. Der Hüllgenerator erzeugt, wenn er getriggert wurde, eine Amplitudenhüllkurve mit programmierbaren Raten für Lautstärkeanstieg und -abfall. Zusätzlich zu den drei Stimmen steht ein programmierbares Filter zur Verfügung, das durch subtraktive Synthese komplexe, dynamische Tonfärbungen erzeugen kann.

Mit dem SID kann der Mikroprozessor den sich verändernden Ausgang des dritten Oszillators und des dritten Hüllgenerators lesen. Diese Ausgaben können als Quelle einer Modulationsinformation benutzt werden, um ein Vibrato zu erzeugen, oder Frequenz/Filter Verschiebungen und ähnliche Effekte. Der dritte Oszillator kann außerdem als Zufallszahlengenerator für Spiele benutzt werden. Zwei A/D Wandler können den SID mit Potentiometern verbinden. Diese können in einem Spiel als Spielpulte oder bei einem Musiksynthesizer als Kontrollregler benutzt werden. Der SID kann externe Tonsignale verarbeiten. Dadurch lassen sich mehrere SID Bausteine hintereinander schalten oder als komplexe polyphone Systeme mischen.

SID Kontrollregister

Im SID gibt es 29 8-Bit Register, die die Tonerzeugung kontrollieren. Diese Register können entweder nur beschrieben (WRITE-only) oder nur gelesen (READ-only) werden und sind unten in Tabelle 1 aufgelistet.

SID Registerbeschreibung

Stimme 1

FREQ LO/FREQ HI (Register 00,01)
Diese Register bilden zusammen eine 16-Bit Zahl, die die Frequenz des Oszillator 1 bestimmt. Die Frequenz ist durch folgende Formel gegeben:

$$F_{out} = (F_n \times F_{clk} / 16777216) \text{ Hz}$$

F_n ist die 16-Bit Zahl in den Frequenzregistern und F_{clk} ist die zum Phase 2 Eingang (Anschluß 6) gehörige System Clock. Für eine Standard 1,0-MHz Clock ergibt sich die Frequenz durch:

$$F_{out} = (F_n \times 0{,}059604645) \text{ Hz}$$

Eine vollständige Wertetabelle zur Erzeugung von acht Oktaven der gleichmäßig temperierten Notenskala, bezogen auf Kammerton a (440 Hz), findet man im Anhang E. Man beachte, daß die Frequenzauflösung von SID für jede Stimmskala ausreicht und einen absatzlosen Übergang von Note zu Note (portamento) ohne unterscheidbare Frequenzsprünge erlaubt.

PW LO/PW HI (Register 02,03)
Diese Register bilden zusammen eine 12-Bit Zahl (Bits 4–7 von PW HI werden nicht benutzt), die die Pulsbreite der Pulswellenform von Oszillator 1 linear kontrolliert. Die Pulsbreite wird durch folgende Gleichung bestimmt:

$$PW_{out} = (PW_n/40{,}95)\%$$

wobei PW_n eine 12-Bit Zahl in den Pulsbreiteregistern darstellt.

Die Auflösung der Pulsbreite erlaubt einen sanften Übergang dieser Breite ohne unterscheidbare Schritte. Man beachte, daß die Pulswellenform für den Oszillator 1 ausgewählt werden muß, damit die Pulsbreiteregister irgendeinen hörbaren Effekt haben. Ein Wert von 0 oder 4095 ($FFF) in den Pulsbreiteregistern erzeugt eine konstante Gleichspannungsausgabe, wohingegen ein Wert von 2048 ($800) eine Rechteckspannung erzeugt.

	ADDRESS					REG #								REG NAME	REG
	A_4	A_3	A_2	A_1	A_0	(HEX)	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Voice 1	TYPE
0	0	0	0	0	0	00	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0	FREQ LO	WRITE-ONLY
1	0	0	0	0	1	01	F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	FREQ HI	WRITE-ONLY
2	0	0	0	1	0	02	PW_7	PW_6	PW_5	PW_4	PW_3	PW_2	PW_1	PW_0	PW-LO	WRITE-ONLY
3	0	0	0	1	1	03	—	—	—	—	PW_{11}	PW_{10}	PW_9	PW_8	PW HI	WRITE-ONLY
4	0	0	1	0	0	04	NOISE				TEST	RING MOD	SYNC	GATE	CONTROL REG	WRITE-ONLY
5	0	0	1	0	1	05	ATK_3	ATK_2	ATK_1	ATK_0	DCY_3	DCY_2	DCY_1	DCY_0	ATTACK/DECAY	WRITE-ONLY
6	0	0	1	1	0	06	STN_3	STN_2	STN_1	STN_0	RLS_3	RLS_2	RLS_1	RLS_0	SUSTAIN/RELEASE	WRITE-ONLY
															Voice 2	
7	0	0	1	1	1	07	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0	FREQ LO	WRITE-ONLY
8	0	1	0	0	0	08	F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	FREQ HI	WRITE-ONLY
9	0	1	0	0	1	09	PW_7	PW_6	PW_5	PW_4	PW_3	PW_2	PW_1	PW_0	PW LO	WRITE-ONLY
10	0	1	0	1	0	0A	—	—	—	—	PW_{11}	PW_{10}	PW_9	PW_8	PW HI	WRITE-ONLY
11	0	1	0	1	1	0B	NOISE				TEST	RING MOD	SYNC	GATE	CONTROL REG	WRITE-ONLY
12	0	1	1	0	0	0C	ATK_3	ATK_2	ATK_1	ATK_0	DCY_3	DCY_2	DCY_1	DCY_0	ATTACK/DECAY	WRITE-ONLY
13	0	1	1	0	1	0D	STN_3	STN_2	STN_1	STN_0	RLS_3	RLS_2	RLS_1	RLS_0	SUSTAIN/RELEASE	WRITE-ONLY
															Voice 3	
14	0	1	1	1	0	0E	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0	FREQ LO	WRITE-ONLY
15	0	1	1	1	1	0F	F_{15}	F_{14}	F_{13}	F_{12}	F_{11}	F_{10}	F_9	F_8	FREQ HI	WRITE-ONLY
16	1	0	0	0	0	10	PW_7	PW_6	PW_5	PW_4	PW_3	PW_2	PW_1	PW_0	PW LO	WRITE-ONLY
17	1	0	0	0	1	11	—	—	—	—	PW_{11}	PW_{10}	PW_9	PW_8	PW HI	WRITE-ONLY
18	1	0	0	1	0	12	NOISE				TEST	RING MOD	SYNC	GATE	CONTROL REG	WRITE-ONLY
19	1	0	0	1	1	13	ATK_3	ATK_2	ATK_1	ATK_0	DCY_3	DCY_2	DCY_1	DCY_0	ATTACK/DECAY	WRITE-ONLY
20	1	0	1	0	0	14	STN_3	STN_2	STN_1	STN_0	RLS_3	RLS_2	RLS_1	RLS_0	SUSTAIN/RELEASE	WRITE-ONLY

															Filter	
21	1	0	1	0	1	15	—	—	—	—	—	FC_2	FC_1	FC_0	FC LO	WRITE-ONLY
22	1	0	1	1	0	16	FC_{10}	FC_9	FC_8	FC_7	FC_6	FC_5	FC_4	FC_3	FC HI	WRITE-ONLY
23	1	0	1	1	1	17	RES_3	RES_2	RES_1	RES_0	FILTEX	FILT 3	FILT 2	FILT 1	RES/FILT	WRITE-ONLY
24	1	1	0	0	0	18	3 OFF	HP	BP	LP	VOL_3	VOL_2	VOL_1	VOL_0	MODE/VOL	WRITE-ONLY
															Misc.	
25	1	1	0	0	1	19	PX_7	PX_6	PX_5	PX_4	PX_3	PX_2	PX_1	PX_0	POT X	READ-ONLY
26	1	1	0	1	0	1A	PY_7	PY_6	PY_5	PY_4	PY_3	PY_2	PY_1	PY_0	POT Y	READ-ONLY
27	1	1	0	1	1	1B	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0	$aOSC_3$/RANDOM	READ-ONLY
28	1	1	1	0	0	1C	E_7	E_6	E_5	E_4	E_3	E_2	E_1	E_0	ENV_3	READ-ONLY

Abbildung I.4

Kontrollregister (Register 04)

Dieses Register enthält 8 Kontrollbits, die verschiedene Einstellungen des Oszillator 1 auswählen.

GATE (Bit 0): Das GATE Bit kontrolliert den Hüllgenerator für Stimme 1. Wenn dieses Bit auf „1" gesetzt ist, wird der Hüllgenerator getriggert und damit der ATTACK/DECAY/SUSTAIN Zyklus eingeleitet. Wird dieses Bit auf „0" zurückgesetzt, beginnt der RELEASE Zyklus. Der Hüllgenerator kontrolliert die Amplitude des Oszillator 1, die am Tonausgang anliegt. Aus diesem Grund muß das GATE Bit (zusammen mit den gewünschten Hüllparametern) für den gewählten Ausgang von Oszillator 1 gesetzt sein, damit dieser hörbar wird. Eine detaillierte Behandlung des Hüllgenerators findet man am Ende dieses Anhangs.

SYNC (Bit 1): Das SYNC Bit synchronisiert, wenn es auf „1" gesetzt ist, die Grundfrequenz von Oszillator 1 mit der Grundfrequenz von Oszillator 3 und erzeugt damit einen „harten Synchronisationseffekt".

Eine Veränderung der Frequenz von Oszillator 1 im Vergleich zu Oszillator 3 erzeugt eine breite Skala komplexer harmonischer Strukturen von Stimme 1 bei der Frequenz von Oszillator 3. Damit eine Synchronisation stattfindet, muß Oszillator 3 auf eine von 0 verschiedene Frequenz gesetzt sein, die vorzugsweise niedriger als die Frequenz von Oszillator 1 liegen sollte. Alle anderen Parameter von Stimme 3 haben keinen Einfluß auf Sync.

RING MOD (Bit 2): Wenn das RING MOD Bit auf „1" gesetzt ist, ersetzt es die Dreieckswellenform als Ausgabe von Oszillator 1 mit einer „Ring-modulierten" Kombination von Oszillator 1 und 3. Eine Veränderung der Frequenz von Oszillator 1 im Vergleich zu Oszillator 3 erzeugt eine breite Skala nichtharmonischer Obertonstrukturen, mit denen man Glocken- oder Gongklänge oder spezielle Effekte erzeugen kann. Damit die Ringmodulation hörbar wird, muß die Dreieckswellenform für den Oszillator 1 gewählt werden und der Oszillator 3 muß auf eine von 0 verschiedene Frequenz eingestellt sein. Alle anderen Parameter der Stimme 3 haben auf die Ringmodulation keinen Einfluß.

TEST (Bit 3): Ist das TEST Bit auf „1" gesetzt, wird der Oszillator 1 zurückgesetzt und auf null gehalten, bis das TEST Bit gelöscht wird. Die Wellenform für Rauschen wird am Ausgang von Oszillator 1 ebenfalls zurückgesetzt und die Ausgabe der Pulswellenform auf einem Gleichspannungsniveau gehalten. Normalerweise wird dieses Bit für Testzwecke benutzt, kann aber auch dazu herangezogen werden, den Oszillator 1 mit externen Ereignissen zu synchronisieren, um dadurch hochkomplexe Wellenformen unter Softwarekontrolle in Echtzeit zu erzeugen.

(Bit 4): Auf „1" gesetzt, wird die Ausgabe einer Dreieckswellenform für Oszillator 1 gewählt. Die Dreieckswellenform enthält wenige harmonische Schwingungen und hat einen weichen, flötenähnlichen Klang.

(Bit 5): Auf „1" gesetzt, wird die Ausgabe einer Sägezahnwellenform für Oszillator 1 gewählt. Die Sägezahnwellenform ist reich an geraden und ungeraden harmonischen Oberschwingungen und hat einen hellen, blechernen Klang.

(Bit 6): Auf „1" gesetzt, wird die Ausgabe einer Pulswellenform für Oszillator 1 gewählt. Der Anteil an harmonischen Oberschwingungen kann bei dieser Wellenform durch das Pulsbreiteregister eingestellt werden, wodurch sich die Tonqualität von einer hellen, hohlen Rechteckschwingung bis zu einem nasalen, flötenden Puls erstrecken kann. Verändert man die Pulsbreite in Echtzeit, erzeugt man einen dynamischen Phaseneffekt, der dem Klang eine gewisse Bewegung hinzufügt. Schnelle Sprünge zwischen unterschiedlichen Pulsbreiten können interessante harmonische Sequenzen erzeugen.

NOISE (Bit 7): Auf „1" gesetzt, wird die Ausgabe einer Rauschwellenform (Noise) für Oszillator 1 gewählt. Bei dieser Ausgabe handelt es sich um ein Zufallssignal, das sich mit der Frequenz von Oszillator 1 verändert. Die Klangqualität kann zwischen einem tiefen Rumpeln bis zu weißem Rauschen variieren, je nachdem, wie die Frequenzregister von Oszillator 1 eingestellt sind. Rauschen wendet man bei der Erzeugung von Geräuschen wie Explosionen, Gewehrschüssen, Düsentriebwerken, Wind, Brandung und anderen frequenzlosen Klängen an, wie etwa auch bei Blechtrommeln und Becken. Eine Veränderung der Oszillatorfrequenz bei eingeschaltetem Rauschen erzeugt einen dramatischen brausenden Effekt.

Eine der Wellenformen muß für den Oszillator 1 gewählt werden, damit das ganze hörbar wird. Es ist jedoch ganz und gar nicht notwendig, Wellenformen abzuwählen, um die Ausgabe von Stimme 1 stumm zu schalten. Die Amplitude von Stimme 1 am Endausgang ist einzig und alleine eine Funktion des Hüllgenerators.

Bemerkung: Die Ausgangswellenformen des Oszillators sind NICHT additiv. Wenn gleichzeitig mehr als eine Ausgangswellenform gewählt wird, wird die resultierende Wellenform durch eine logische AND Operation erzeugt. Obwohl diese Technik dazu benutzt werden kann, neben den vier oben beschriebenen, weitere Wellenformen zu erzeugen, sollte sie mit Vorsicht angewandt werden. Wenn bei eingeschaltetem Rauschen irgendeine andere Wellenform ausgewählt wird, kann sich die Rauschausgabe „aufhängen". Sollte das eintreten, bleibt die Rauschausgabe solange stumm, bis sie durch das TEST Bit zurückgesetzt wird oder bis RES (Anschluß 5) heruntergeschaltet wird.

ATTACK/DECAY (Register 05):

Die Bits 4-7 dieses Registers (ATK0-ATK3) wählen eine von sechzehn Anstiegsraten für den Hüllgenerator der Stimme 1 aus. Die Anstiegsrate bestimmt, wie schnell die Ausgabe von Stimme 1 von null bis zur maximalen Amplitude ansteigt, nachdem der Hüllgenerator getriggert wurde. Die 16 ATTACK Raten sind auf der nächsten Seite aufgelistet. Die Bits 0-3 (DCY0-DCY3) wählen eine von 16 Abfallraten für den Hüllgenerator aus. Der Abfallzyklus folgt dem Anstiegszyklus und die Abfallrate bestimmt, wie schnell die Ausgabe von der maximalen Amplitude auf das ausgewählte Halteniveau fällt. Die 16 DECAY Raten sind ebenfalls auf der nächsten Seite aufgelistet.

SUSTAIN/RELEASE (Register 06):

Die Bits 4-7 dieses Registers (STN0-STN3) wählen eine von 16 Halteniveaus für den Hüllgenerator aus. Der Haltezyklus folgt dem Abfallzyklus und die Ausgabe von Stimme 1 bleibt solange auf der gewählten Halteamplitude, als das GATE Bit gesetzt bleibt. Die Halteniveaus reichen in 16 linearen Schritten von null bis zur maximalen Amplitude. Ein Haltewert von 0 wählt eine Amplitude null aus und ein Haltewert von 15 ($F) wählt die maximale Amplitude. Ein Haltewert von 8 würde die Stimme 1 bei einer Amplitude halten, die der halben maximalen Amplitude entspricht, die durch den Anstiegszyklus erreicht wurde.

Hüllraten

Wert		ATTACK Rate	DECAY/RELEASE Rate
Dez	(Hex)	(Zeit/Zyklus)	(Zeit/Zyklus)
0	0	2 ms	6 ms
1	1	8 ms	24 ms
2	2	16 ms	48 ms
3	3	24 ms	72 ms
4	4	38 ms	114 ms
5	5	56 ms	168 ms
6	6	68 ms	204 ms
7	7	80 ms	240 ms
8	8	100 ms	300 ms
9	9	250 ms	750 ms
10	A	500 ms	1,5 s
11	B	800 ms	2,4 s
12	C	1 s	3 s
13	D	3 s	9 s
14	E	5 s	15 s
15	F	8 s	24 s

Bemerkung: Hüllraten basieren auf einer Phase 2 Clock von 1.0 MHz. Für andere Frequenzen von Phase 2 muß man die angegebene Rate mit 1 MHz/Phase 2 mulitplizieren. Die Raten beziehen sich auf die Zeit, die pro Zyklus zur Verfügung steht. Beispielsweise würde bei einem gegebenen Anstiegswert von 2 der Anstiegszyklus 16 ms benötigen, um von null auf maximale Amplitude zu steigen. Die Abfall/Ausklingraten beziehen sich auf den Zeitbedarf, den diese Zyklen beanspruchen würden, um von der maximalen Amplitude auf null zu fallen.

Die Bits 0-3 (RLS0-RLS3) wählen 1 von 16 Ausklingraten für den Hüllgenerator aus. Der Ausklingzyklus folgt dem Haltezyklus, wenn das GATE Bit auf „0" zurückgesetzt wird. Zu diesem Zeitpunkt fällt die Ausgabe von Stimme 1 mit der gewählten Ausklingrate von der Halteamplitude auf die Amplitude null zurück. Die 16 Ausklingraten (RELEASE) stimmen mit den Abfallraten (DECAY) überein.

Bemerkung: Das Zyklusverhalten des Hüllgenerators kann zu jedem Zeitpunkt über das GATE Bit verändert werden. Der Hüllgenerator kann ohne Einschränkung getriggert und abgeschaltet werden. Wenn beispielsweise das GATE Bit zurückgesetzt wird, bevor der Hüllgenerator den Anstiegszyklus beendet hat, beginnt sofort der Ausklingzyklus, der mit der Amplitude anfängt, die gerade erreicht wurde. Wenn der Hüllgenerator dann wieder getriggert wird (bevor der Ausklingzyklus die Amplitude null erreicht hat) beginnt ein neuer Anstiegszyklus, der mit der Amplitude anfängt, die gerade erreicht wurde. Diese Technik kann über eine Softwarekontrolle in Echtzeit zur Erzeugung komplexer Amplitudenhüllkurven benutzt werden.

Stimme 2

Die Register 07-$0D kontrollieren die Stimme 2 und sind funktionell mit den Registern 00-06 identisch, mit folgenden Ausnahmen:

1. Wenn es ausgewählt wurde, synchronisiert SYNC den Oszillator 2 mit dem Oszillator 1.
2. Wenn es ausgewählt wurde, ersetzt RING MOD die Dreiecksausgabe von Oszillator 2 durch die Ring-modulierte Kombination von Oszillator 2 und 1.

Stimme 3

Die Register $0E-$14 kontrollieren die Stimme 3 und sind funktionell mit den Registern 00-06 identisch mit folgenden Ausnahmen:

1. Wenn es ausgewählt wurde, synchronisiert SYNC den Oszillator 3 mit dem Oszillator 2.
2. Wenn es ausgewählt wurde, ersetzt RING MOD die Dreiecksausgabe von Oszillator 3 durch die Ring-modulierte Kombination von Oszillator 3 und 2.

Eine typische Operation, bezogen auf eine Stimme, besteht darin, die gewünschten Parameter auszuwählen: Frequenz, Wellenform, Effekte (SYNC, RING MOD) und Hüllraten. Dann triggert man die Stimme, wann immer man den Klang wünscht. Der Klang kann für jede beliebige Zeitdauer gehalten und durch Löschen des GATE Bit beendet werden. Jede Stimme kann für sich benutzt werden, mit unabhängigen Parametern und eigener Auslösung, oder gemeinsam, um eine einzige große Stimme zu erzeugen. Werden sie gemeinsam benutzt, führt eine leichte Verstimmung eines jeden Oszillators oder eine Stimmung in musikalischen Intervallen zu einem reichen, lebendigen Klang.

Filter

FC LO/FC HI (Register $15,$16)

Diese Register bilden zusammen eine 11-Bit Zahl (Bits 3-7 von FC LO werden nicht benutzt), die die Kanten- (oder Mitten-) Frequenz des programmierbaren Filters linear kontrollieren. Die ungefähre Kantenfrequenz reicht von 30 Hz bis 12 kHz.

RES/FILT (Register $17)

Die Bits 4-7 dieses Registers (RES0-RES3) kontrollieren die Resonanz des Filters. Bei der Resonanz handelt es sich um einen Erhöhungseffekt, der Frequenzkomponenten bei der Kantenfrequenz des Filters verstärkt, wodurch ein schärferer Klang erreicht wird. 16 Resonanzeinstellungen stehen zur Verfügung, die linear sich von keiner Resonanz (0) bis zu maximaler Resonanz (15 oder $F) erstrecken. Die Bits 0-3 bestimmen, welche Signale durch das Filter geleitet werden:

FILT 1 (Bit 0): Auf „0" gesetzt erscheint Stimme 1 direkt am Tonausgang und wird vom Filter nicht beeinflußt. Auf „1" gesetzt wird Stimme 1 vom Filter bearbeitet und der Oberwellengehalt von Stimme 1 wird entsprechend den ausgewählten Filterparametern verändert.

FILT 2 (Bit 1): Das gleiche wie Bit 0 für Stimme 2.

FILT 3 (Bit 2): Das gleiche wie Bit 0 für Stimme 3.

FILTEX (Bit 3): Dasselbe wie Bit 0 für externe Toneingabe (Anschluß 26).

MODE VOL (Register $18)

Die Bits 4-7 dieses Registers wählen verschiedene Filtermodi und Ausgabemöglichkeiten aus:

LP (Bit 4): Auf „1" gesetzt wird die Tiefpaßausgabe des Filters gewählt und auf den Tonausgang gelegt. Bei einem gegebenen Filtereingangssignal werden alle Frequenzkomponenten, die unterhalb der Filterkantenfrequenz liegen, unverändert durchgelassen, während alle Frequenzkomponenten oberhalb der Kantenfrequenz mit einer Rate von 12 dB/Oktave abgeschwächt werden. Der Tiefpaß Modus erzeugt einen satten Klang.

BP (Bit 5): Dasselbe wie Bit 4, jedoch für die Bandpaßausgabe. Alle Frequenzkomponenten oberhalb und unterhalb der Kantenfrequenz werden mit einer Rate von 6 dB/Oktave abgeschwächt. Der Bandpaß Modus erzeugt einen dünnen, offenen Klang.

HP (Bit 6): Dasselbe wie Bit 4, jedoch für die Hochpaßausgabe. Alle Frequenzen oberhalb der Kantenfrequenz werden unverändert durchgelassen, während alle Frequenzkomponenten unterhalb der Kantenfrequenz mit einer Rate von 12 dB/Oktave abgeschwächt werden. Der Hochpaß Modus erzeugt einen blechernen, schrillen Klang.

3 OFF (Bit 7): Auf „1" gesetzt wird die Ausgabe von Stimme 3 vom direkten Tonkanal abgekoppelt. Setzt man Stimme 3 so, daß sie das Filter nicht durchläuft (FILT 3 = 0) und schaltet 3 OFF an, um Stimme 3 vom Tonausgang fernzuhalten, kann man diese Stimme zu Modulationszwecken benutzen, ohne daß die Stimme selbst am Ausgang hörbar ist.

Bemerkung: Die Filterausgabemodi SIND additiv und unterschiedliche Filtermodi können gleichzeitig angewählt werden. Beispielsweise können sowohl LP als auch HP ausgewählt werden, um das Verhalten eines Notch-Filters zu erzeugen. Damit das Filter einen hörbaren Effekt erzeugt, muß mindestens eine Filterausgabe angewählt sein und mindestens eine Stimme muß durch das Filter geleitet werden. Das Filter stellt wahrscheinlich das wichtigste Element im SID dar, da es die Erzeugung komplexer Tonfarben über subtraktive Synthese erlaubt (das Filter wird dazu benutzt, spezifische Frequenzkomponenten aus einem oberwellenreichen Eingangssignal herauszufiltern). Die besten Resultate erreicht man, wenn man die Kantenfrequenz in Echtzeit verändert.

Bits 0-3: Diese Bits (VOL0-VOL3) wählen 1 von 16 möglichen Lautstärkepegeln für die zusammengesetzte Endausgabe des Tons aus. Die Ausgangslautstärke reicht von keine Ausgabe (0) bis maximale Lautstärke (15 oder $F) in 16 linearen Schritten. Diese Kontrolle kann als statische Lautstärkekontrolle zum Lautstärkeausgleich bei Multichip-Systemen benutzt werden oder dazu, dynamische Lautstärkeeffekte, wie z.B. ein Tremolo, zu erzeugen. Ein von Null verschiedener Lautstärkepegel muß ausgewählt werden, damit der SID überhaupt einen Klang erzeugt.

Verschiedenes

POTX (Register $19)

Mit diesem Register kann der Mikroprozessor die Position des Potentiometers auslesen, das an POTX (Anschluß 24) angeschlossen ist. Die Werte können von 0 bei kleinstem Widerstand bis 255 ($FF) beim größten Widerstand variieren. Der Wert bleibt gültig und wird bei jedem 512. Phase 2 Clockzyklus auf den neuesten Stand gebracht. Im Herstellerdatenblatt findet man unter der Anschlußbeschreibung weitere Informationen zu den Potentiometer- und Kapazitätswerten.

POTY (Register $1A)

Dasselbe wie für POTX für das Potentiometer, das an POTY (Anschluß 23) angeschlossen ist.

OSC 3/RANDOM (Register $1B)

Mit diesem Register kann der Mikroprozessor die oberen 8 Ausgabebits des Oszillator 3 auslesen. Die erzeugte Zahlenart steht in direktem Zusammenhang mit der ausgewählten Wellenform. Wenn die Sägezahnwellenform für den Oszillator 3 gewählt wurde, liefert dieses Register eine von 0 bis 255 ($FF) ansteigende Zahlenserie, die sich mit einer Rate verändert, die durch die Frequenz von Oszillator 3 bestimmt wird. Wird die Dreieckswellenform gewählt,

steigt die Ausgabe von 0 bis 255 und fällt dann wieder bis 0. Wird die Pulswellenform gewählt, springt der Ausgang zwischen 0 und 255. Die Rauschwellenform erzeugt eine Serie von Zufallszahlen. Deshalb kann dieses Register für Spiele als Zufallszahlengenerator benutzt werden. Es gibt eine Vielzahl von Zeitgeber- und Sequenzanwendungen für das OSC 3 Register. Seine Hauptfunktion ist jedoch wahrscheinlich die eines Modulationsgenerators. Die durch dieses Register erzeugten Zahlen können über Software in Echtzeit zum Oszillator, zu den Filterfrequenzregistern oder zu den Pulsbreiteregistern addiert werden. Auf diese Weise lassen sich viele dynamische Effekte erzeugen. Sirenenartige Klänge kann man dadurch erzeugen, daß man den Sägezahnausgang von OSC 3 zu der Frequenzkontrolle eines anderen Oszillators addiert. Synthesizer „Sample and Hold" Effekte lassen sich durch Addition des Rauschausgangs von OSC 3 zu den Filter-Frequenzkontrollregistern produzieren. Ein Vibrato erreicht man, indem man die Frequenz von OSC 3 auf ungefähr 7 Hz einstellt und die OSC 3 Dreiecksausgabe (mit geeigneter Einstellung) zur Frequenzkontrolle eines anderen Oszillators addiert. Durch Veränderung der Frequenz von Oszillator 3 und entsprechende Einstellung der OSC 3 Ausgabe steht eine unbegrenzte Skala von Effekten zur Verfügung. Wird Oszillator 3 normalerweise zur Modulation benutzt, sollte die Tonausgabe von Stimme 3 stumm geschaltet werden (3 OFF = 1).

ENV 3 (Register $1C)

Das gleiche wie OSC 3. Mit diesem Register kann der Mikroprozessor jedoch die Ausgabe des Hüllgenerators von Stimme 3 lesen. Diese Ausgabe kann zur Filterfrequenz addiert werden, um Oberwellenhüllkurven, WAH-WAH und ähnliche Effekte zu produzieren. Phaser-Klänge lassen sich durch Addition dieser Ausgabe zu den Frequenzkontrollregistern eines Oszillators bilden. Der Hüllgenerator von Stimme 3 muß getriggert werden, damit irgendeine Ausgabe von diesem Register erfolgt. Das OSC 3 Register spiegelt jedoch immer die veränderliche Ausgabe des Oszillators wider und wird in keiner Weise durch den Hüllgenerator beeinflußt.

6525 Tri-Port Interface

Aufbau

Das 6525 TRI-PORT INTERFACE (TPI) wurde entworfen, um die Implementation komplexer I/O Operationen in Mikrocomputersystemen zu vereinfachen. Es kombiniert zwei zugeordnete 8-Bit I/O Kanäle mit einem dritten 8-Bit Kanal, der entweder für normale I/O Operationen programmierbar ist oder die Priorität von Unterbrechung/Quittierung kontrolliert. In Abhängigkeit von der gewählten Betriebsart kann der 6525 entweder 24 individuell programmierbare I/O Leitungen zur Verfügung stellen oder 16 I/O Leitungen, 2 Quittungsleitungen und 5 Prioritätsunterbrechungseingänge.

6525 Adressierung

6525 Register (direkte Adressierung)

*000	R0	PRA-Port Register A
001	R1	PRB-Port Register B
010	R2	PRC-Port Register C
011	R3	DDRA-Datenrichtungsregister A
100	R4	DDRB-Datenrichtungsregister B
101	R5	DDRC-Datenrichtungsregister C/Unterbrechungsmaskierungsregister
110	R6	CR-Kontrollregister
111	R7	AIR-Unterbrechung aktiv Register

*Bemerkung: in der Reihenfolge RS2, RS1, RS0

6525 Kontrollregister

CR	CB_1	CB_0	CA_1	CA_0	IE_4	IE_3	IP	MC
AIR				A_4	A_3	A_2	A_1	A_0
DDRC Wenn MC = 1				M_4	M_3	M_2	M_1	M_0
PRC Wenn MC = 1	CB	CA	$\overline{IRQ}$	I_4	I_3	I_2	I_1	I_0

CA, CB - Funktionsbeschreibung

Bei den Leitungen CA und CB handelt es sich um Ausgänge, die auf die gleiche Weise benutzt werden, wie der Ausgang CA_2 und CB_2 des 6520.

CA Ausgabe Betriebsarten

CA_1	CA_0	Betriebsart	Beschreibung
0	0	„Handshake" Lesen	CA wird hochgesetzt bei aktivem Übergang des I3 Interrupt Inputsignals und heruntergesetzt durch eine Operation des Mikroprozessors „Lese A Daten". Dadurch ist eine positive Kontrolle der Datenübertragung von der peripheren Einheit zum Mikroprozessor möglich.
0	1	Puls Ausgabe	CA geht nach unten für IMS nach einer Operation „Lese A Daten". Der Puls kann dazu benutzt werden, einer peripheren Einheit anzuzeigen, daß Daten angenommen wurden.
1	0	Manuelle Ausgabe	CA heruntergeschaltet
1	1	Manuelle Ausgabe	CA heraufgeschaltet

CB Ausgabe Betriebsarten

CB_1	CB_0	Betriebsart	Beschreibung
0	0	„Handshake" Schreiben	CB wird hochgesetzt bei aktivem Übergang des I4 Interrupt Inputsignals und heruntergesetzt durch eine Operation des Mikroprozessors „Schreibe B Daten". Dadurch ist eine positive Kontrolle der Datenübertragung vom Mikroprozessor zur peripheren Einheit möglich.
0	1	Puls Ausgabe	CB geht nach unten für IMS nach einer Operation „Schreibe B Daten". Der Puls kann dazu benutzt werden, einer peripheren Einheit anzuzeigen, daß Daten zur Verfügung stehen.
1	0	Manuelle Ausgabe	CA heruntergeschaltet
1	1	Manuelle Ausgabe	CA heraufgeschaltet

Interrupt Maskierungsregister - Beschreibung

Wenn der Interrupt Modus ausgewählt wurde (MC = 1), wird das Datenrichtungsregister für Kanal C (DDRC) dazu benutzt, die entsprechende Interrupteingabe zuzulassen oder zu sperren. Zum Beispiel: ist M0 = 0, dann ist I0 gesperrt und jedes I0 Interrupt, das im Interrupt Latchregister zwischengespeichert ist, wird nicht in das AIR übertragen und führt deshalb nicht dazu, daß IRQ heruntergeschaltet wird. Das Interrupt Latch kann dadurch gelöscht werden, daß man eine „0" in das entsprechende I Bit in PRC schreibt.

Port Register C - Beschreibung

Das Port Register C (PRC) kann in zwei Betriebsarten arbeiten. Die Betriebsart wird durch das Bit MC in Register CR kontrolliert. Ist MC = 0, arbeitet PRC wie ein Standard I/O Port, der sich genauso wie PRA und PRB verhält. Ist MC = 1, wird das Port Register C für das Handshaking und als Eingang und Ausgang der Prioritätsunterbrechung benutzt.

PRC wenn MC = 0

PC_7	PC_6	PC_5	PC_4	PC_3	PC_2	PC_1	PC_0

PRC wenn MC = 1

CB	CA	$\overline{IRQ}$	I_4	I_3	I_2	I_1	I_0

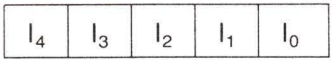
Interrupt Flankenkontrolle

Die Bits IE_4 und IE_3 des Kontrollregisters (CR) werden dazu benutzt, die aktive Flanke festzulegen, die durch das Interrupt Latch erkannt wird.

Wenn IE_4 (IE_3) = 0 ist, wird das I_4 (I_3) Latch durch einen negativen Übergang der I_4 (I_3) Eingabe gesetzt.

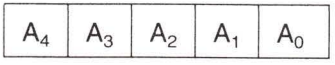
Wenn IE_4 (IE_3) = 1 ist, wird das I_4 (I_3) Latch durch einen positiven Übergang der I_4 (I_3) Eingabe gesetzt.

Alle anderen Interrupt Latches (I_2, I_1, I_0) werden mit einem negativen Übergang der entsprechenden Interrupteingabe gesetzt.



Interrupt Latch Register
Wird bei Lesen von AIR gelöscht. Man benutze folgende Gleichung

$$ILR \leftarrow ILR \oplus AIR$$



Active Interrupt Register
Wird bei Schreiben in AIR gelöscht.

Interrupt Prioritätswahl
IP = 0 Keine Priorität
IP = 1 Interrupts mit Priorität

Funktionsbeschreibung

1. IP = 0 Keine Priorität

Jede Unterbrechungsinformation, die im Interrupt Latch Register (ILR) zwichengespeichert wurde, wird sofort in das Active Interrupt Register (AIR) transferiert und $\overline{IRQ}$ wird nach unten gezogen. Beim Lesen der Unterbrechung wird $\overline{IRQ}$ nach oben zurückgesetzt und die zugehörigen Bits des Interrupt Latch Registers werden durch eine „Exclusive OR" Operation des ILR mit dem AIR gelöscht. Nachdem die entsprechende Unterbrechungsanforderung bedient wurde, löscht ein Schreibvorgang in das AIR das AIR und initiiert eine neue Unterbrechungssequenz, wenn irgendwelche Unterbrechungen während der früheren Unterbrechungsbedienung empfangen wurden. Bei dieser Betriebsart ohne Priorität kann es vorkommen, daß zwei oder mehr Unterbrechungen gleichzeitig auftreten und in das AIR übertragen werden. Falls das passiert, ist es Aufgabe der Software, so etwas zu erkennen und darauf entsprechend zu reagieren.

2. IP = 1 Interrupts mit Priorität

Bei dieser Betriebsart werden die Unterbrechungseingaben nach folgender Reihenfolge mit einer Priorität versehen

$$I_4 > I_3 > I_2 > I_1 > I_0$$

Bei dieser Betriebsart kann immer nur ein Bit des AIR zu irgendeiner Zeit gesetzt werden. Tritt eine Unterbrechung auf, wird diese im Interrupt Latch Register zwischengespeichert, die $\overline{IRQ}$ Leitung nach unten gezogen, und das entsprechende Bit von AIR gesetzt. Um die Arbeitsweise der Unterbrechung mit Priorität vollständig zu verstehen, sollte man sich am einfachsten die folgenden Beispiele anschauen.

A. Der erste Fall ist der einfachste. Eine einzelne Unterbrechung tritt auf, und der Prozessor kann diese vollständig bedienen, bevor eine weitere Unterbrechungsanforderung empfangen wird.

   1. Unterbrechung I_1 wird empfangen.
   2. Bit I_1 wird im Interrupt Latch Register hochgesetzt.

3. $\overline{IRQ}$ wird heruntergezogen.
4. A_1 wird hochgesetzt.
5. Prozessor erkennt $\overline{IRQ}$ und liest AIR, um zu bestimmen, welche Unterbrechung auftauchte.
6. Bit I_1 wird zurückgesetzt und $\overline{IRQ}$ wird auf hoch zurückgesetzt.
7. Prozessor bedient Unterbrechung und signalisiert Ende der Bedienungsroutine durch Schreiben in AIR.
8. A_1 wird nach unten zurückgesetzt und Unterbrechungssequenz ist vollendet.

B. Der zweite Fall tritt ein, wenn eine Unterbrechung empfangen wurde und eine Unterbrechung höherer Priorität eintrifft. (Siehe Bemerkung)

1. Unterbrechung I_1 wird empfangen.
2. Bit I_1 wird im Interrupt Latch Register hochgesetzt.
3. $\overline{IRQ}$ wird heruntergezogen und A_1 hochgesetzt.
4. Prozessor erkennt $\overline{IRQ}$ und liest AIR, um zu bestimmen, welche Unterbrechung auftauchte.
5. Bit I_1 wird zurückgesetzt und $\overline{IRQ}$ wird auf hoch zurückgesetzt.
6. Prozessor beginnt Bedienung von I_1 Interrupt und der I_2 Interrupt wird empfangen.
7. A_2 wird gesetzt, A_1 wird nach unten zurückgesetzt und $\overline{IRQ}$ heruntergezogen.
8. Prozessor hat Bedienung von I_1 Interrupt noch nicht vollendet. Deshalb wird diese Routine automatisch in der 6500 Stapelreihe abgelegt, wenn neues $\overline{IRQ}$ für Unterbrechung I_2 empfangen wurde.
9. Prozessor liest AIR, um Auftreten von I_2 Interrupt zu bestimmen und Bit I_2 des Interrupt Latch wird zurückgesetzt.
10. Prozessor bedient I_2 Interrupt, löscht A_2 durch Beschreiben von AIR und kehrt von Unterbrechung zurück. Rückkehr von Unterbrechung bringt 650x Prozessor dazu, die Bedienung von I_1 Interrupt wieder aufzunehmen.
11. Beim Löschen des A_2 Bit in AIR wird das A1 Bit nicht zu „1" wiederhergestellt. Eine interne Verschaltung sorgt dafür, daß eine Unterbrechung mit niedrigerer Priorität das wieder aufgenommene I_1 nicht mehr unterbrechen kann.

C. Der dritte Fall tritt ein, wenn eine Unterbrechung empfangen wurde und eine Unterbrechung mit niedrigerer Priorität auftaucht.

1. Interrupt I_1 wurde empfangen und zwischengespeichert.
2. $\overline{IRQ}$ wurde nach unten gezogen und A_1 hochgesetzt.
3. Prozessor erkennt $\overline{IRQ}$ und liest AIR, um zu bestimmen, daß I_1 Interrupt auftrat.
4. Prozessorlogik bedient I_1 Interrupt, während der I_0 Interrupt auftrat und zwischengespeichert wurde.
5. Nach Vollendung der I_1 Interruptroutine beschreibt der Prozessor AIR, um A_1 zu löschen und 6525 anzuzeigen, daß Interruptbedienung vollendet.
6. Latch I_0 Interrupt wird in AIR übertragen und $\overline{IRQ}$ heruntergezogen, um neue Unterbrechungssequenz zu beginnen.

Bemerkung: Es wurde darauf hingewiesen, daß der 6525 Prioritätsunterbrechungsinformationen von vorher bedienten Unterbrechungen erhält.

Dies wird durch Benutzung eines Unterbrechungsstapel erreicht. Dieser Stapel wird bei jedem Lesevorgang aus dem AIR mit einer „Push" Operation und bei jedem Schreibvorgang in den AIR mit einer „Pull" Operation bedient. Daher ist es wichtig, keine unnötigen Lese- oder Schreibvorgänge des AIR durchzuführen, da diese zu zusätzlichen und unerwünschten Stapeloperationen führen.

Der einzige Lesevorgang aus dem AIR sollte ausschließlich als Antwort auf eine Unterbrechungsanforderung erfolgen.

Der einzige Schreibvorgang in den AIR sollte erfolgen, um dem 6525 anzuzeigen, daß die Unterbrechungsbedienung vollendet ist.

Glossar

Die Zahlen in Klammern geben das Kapitel an, in dem das Wort oder der Begriff zum ersten Mal benutzt wurde.

Absolute Adresse: (5) Eine Adresse, die jede Stelle im Speicher bezeichnen kann.

Adressbus: (1) Ein Bus, der anzeigt, welcher Teil des Speichers für die nächste Speicheroperation benötigt wird.

Adresse: (1) Die Einheit einer spezifischen Stelle innerhalb des Speichers.

Adressmodus: (5) Die Art, auf die ein Befehl Information innerhalb des Speichers erhält.

Akkumulator: (3) Das A Register; das Register, das für Berechnungen benutzt wird.

Algorithmus: (1) Eine Methode oder Prozedur, eine Berechnungsaufgabe durchzuführen.

Arithmetisch Schieben oder Rotieren: (4) Ein Schieben oder Rotieren, bei dem gewöhnlich das Vorzeichen einer Zahl erhalten bleibt.

Assembler: (2) Ein Programm, das einen Quellcode in einen Objektcode assembliert oder übersetzt.

Assembler Code: (1) Auch Quellcode genannt. Ein Programm, das in einer für den Menschen besser lesbaren Form geschrieben ist. Muß vor der Benutzung übersetzt (assembliert) werden.

Assemblersprache: (1) Ein Satz von Anweisungen oder eine Sprache, in der ein Quellprogramm vor der Assemblierung geschrieben sein muß.

Assemblieren: (1) Der Prozess, der Quellcode in Objektcode umwandelt.

Ausgewählt: (1) Ein Baustein oder eine Einheit, der signalisiert wurde, an einer Datenübertragung teilzunehmen. Wenn der Baustein oder diese Einheit nicht ausgewählt wurde, ignoriert er die Datenoperationen.

Bildschirm-Editierung: (1) Die Möglichkeit, auf dem Bildschirm des Computers Veränderungen vorzunehmen, die entsprechende Veränderungen im Speicher verursachen.

Bildschirmspeicher: (2) Der Teil eines Computers, in dem die Information, die auf dem Bildschirm angezeigt wird, enthalten ist. Eine Veränderung des Bildschirmspeichers verändert die Bildschirmanzeige. Durch Lesen des Bildschirmspeichers erhält man die Information, die auf dem Bildschirm ist.

Binär: (1) Etwas, das zwei mögliche Zustände einnehmen kann. Eine Zahl, die auf Ziffern basiert, von denen eine jede zwei mögliche Zustände hat

Bit: (1) Abkürzung für binary digit: das kleinste Informationselement innerhalb eines Computers.

Bootstrap: (6) Ein Programm, das ein anderes Programm startet.

Breakpoint: (8) Unterbrechungspunkt. Eine Stelle, an der das Programm anhält, um eine Fehlersuche zu ermöglichen.

Bug: (8) „Wanze". Ein Fehler in einem Programm.

Bus: (1) Eine Anzahl von Leitungen, die mehrere Einheiten miteinander verbinden.

Byte: (1) Eine Gruppe von acht Informationsbits. Das normale Maß für Computerspeicher.

Calling Point: (2) Aufrufpunkt. Eine Programmstelle, von der aus eine Unterroutine aufgerufen wird. Die Unterroutine kehrt nach ihrer Beendigung zum Aufrufpunkt zurück.

Datenbus: (1) Ein Bus, der dazu benutzt wird, Daten zwischen dem Speicher und dem Mikroprozessor zu übertragen.

Debugging: (8) Testen eines Programms, um mögliche Fehler aufzudecken.

Dekrementieren: (2) Um einen Wert von 1 verkleinern.

Descriptor: (6) Ein 3-Byte Datensatz, der die Länge und Speicherstelle eines String angibt.

Dezimal: (1) Ein Zahlensystem, das auf einem System von 10 Stellen basiert; das „normale" Zählsystem, das vom Menschen benutzt wird.

Disassembler: (2) Ein Programm, das Objektcode in Assemblercode umwandelt, um Programmuntersuchungen zu ermöglichen.

Disassemblieren: (2) Objektcode in Assemblercode umwandeln. Ähnlich einem LIST in BASIC.

Dynamischer String: (6) Eine Zeichenkette, die im Speicher abgelegt werden muß, nachdem sie empfangen oder berechnet wurde.

Effektive Adresse: (2) Die Adresse, die vom Prozessor benutzt wird, wenn er bei der Durchführung eines Befehls Daten handhabt. Diese kann sich von der Befehlsadresse (oder dem „Operand") durch Indizierung oder indirekte Adressierung unterscheiden.

Ereignisflag: (7) Eine Flag, die anzeigt, daß irgendein Ereignis stattgefunden hat.
File: (8) Eine Ansammlung von Daten, die auf irgendeiner externen Einheit gespeichert ist.
Flag: (3) Flagge oder Fahne. Ein Anzeiger für An/Aus, der irgendeine Bedingung anzeigt.
Floating accumulator: (7) Eine Gruppe von Speicherstellen, die von BASIC dazu benutzt werden, Zahlenberechnungen auszuführen.
Garbage collection: (6) Ein BASIC Vorgang, bei dem aktive Zeichenketten zusammengerückt und inaktive Zeichenketten verworfen werden. Bei einigen Computern kann dieser Vorgang sehr zeitraubend sein.
Inkrementieren: (2) Um einen Wert von 1 vergrößern.
Indexregister: (2) Das X oder Y Register, das zur Veränderung der effektiven Adresse benutzt werden kann.
Indirekte Adresse: (5) Ein Adressierungsschema, bei dem der Befehl die Speicherstelle der aktuell zu benutzenden Adresse enthält; eine Adresse von einer Adresse.
Indizieren: (2) Eine Adresse verändern, indem man den Inhalt eines Indexregisters addiert.
Instruktion: (1) Ein Programmelement, das dem Prozessor mitteilt, was er zu tun hat.
Interrupt: (1) Unterbrechung. Ein Ereignis, das den Prozessor veranlaßt, sein normales Programm zu verlassen, sodaß irgendein anderes Programm, gewöhnlich zeitweilig, die Kontrolle übernimmt.
Interrupt Enable Register: (7) Eine Speicherstelle in einem IA Baustein, die bestimmt, ob ein ausgewähltes Ereignis einen Interrupt auslöst oder nicht.
Interrupt Flag: (7) Unterbrechungsflag. Ein Signal innerhalb des IA, welches anzeigt, daß ein bestimmtes Ereignis die Anforderung stellt, daß eine Unterbrechung stattfindet.
Interrupt Flag Register: (7) Eine Speicherstelle innerhalb des IA, in der Unterbrechungsereignisse festgestellt und auf Wunsch abgeschaltet werden können.
Kanal: (8) Ein Weg, der den Computer mit einer seiner externen Einheiten verbindet.
Kernal: (2) Commodore's Betriebssystem.
Kommentar: (8) Ein Programmelement, das den Computer zu nichts veranlaßt. Wird für den Leser des Programms als Erklärungshilfe benutzt.
Kommutativ: (3) Eine mathematische Operation, die in beiden Richtungen ausgeführt werden kann, d.h. 3+4 ergibt das gleiche Resultat wie 4+3.
Kontrollbus: (1) Ein Bus, der Zeitpunkt und Richtung des Datenflusses zu den verschiedenen angeschlossenen Einheiten anzeigt.
Label: (8) Eine symbolische Adresse, ein Name, der eine Speicherstelle identifiziert.
Latch: (7) Eine Flag, die „einrastet" oder zwischengespeichert wird.
Lesen: (1) Information von einer Einheit erhalten.
Load: (1) Information aus dem Speicher in den Prozessor bringen. Bei einer Load Operation handelt es sich um einen Kopiervorgang. Die Information bleibt auch noch im Speicher.
Logische File Nummer: (8) Die Einheit eines Files, wie sie vom Programmierer benutzt wird.
Logischer Operator: (3) Eine Operation, die einzelne Bits innerhalb eines Byte beeinflußt: AND, ORA und EOR.
Logisch Schieben oder Rotieren: (4) Ein Schiebevorgang, der das Vorzeichen einer Zahl mit Vorzeichen nicht erhält.
Maschinencode: (1) Anweisungen, die in Maschinensprache geschrieben sind.
Maschinensprache: (1) Ein Satz von Befehlen, der es ermöglicht, Anweisungen an den Prozessor zu geben.
Maschinensprachemonitor: (1) Ein Programm, das die Kommunikation mit dem Computer in einer Weise ermöglicht, wie sie zur Maschinenspracheprogrammierung geeignet ist.
Memory: (1) Der Speicher, der vom Computer benutzt wird; jede Stelle ist durch eine Adresse identifiziert.
Memory Map: Ein Speicherplan, der die Belegung des Speichers mit festen Programmen oder die Belegung von Adressen für spezielle Aufgaben darstellt.
Memory mapped: (1) Schaltkreise, die unter Benutzung einer Speicheradresse erreicht werden können, obwohl diese nicht für Speicherzwecke benutzt werden.
Memory Page: (5) Speicherseite. Ein Satz von 256 Speicherstellen, bei denen alle Adressen das gleiche obere Byte haben.
Mikrocomputer: (1) Ein Computersystem, das einen Mikroprozessor, Speicher und Eingabe/Ausgabe Schaltungen enthält. Ein Computer, der unter Benutzung von Mikrochips gebaut ist.

Mikroprozessor: (1) Die zentrale Logik eines Mikrocomputers, die logische und arithmetische Fähigkeiten besitzt. Ein Prozessor, der auf einem Mikrochip gebaut ist.
Monitor: (1) Ein Programm, das dem Benutzer die Kommunikation mit dem Computer erlaubt. Alternativ eine Bildschirmeinheit.
Nichtsymbolischer Assembler: (2) Ein Assembler, bei dem aktuelle Adressen benutzt werden müssen.
Non-maskable Interrupt, NMI: (7) Nichtmaskierbare Unterbrechung. Eine Unterbrechungsart, die nicht gesperrt werden kann.
Objektcode: (1) Das Maschinenspracheprogramm, das in einem Computer laufen kann.
Octothorpe: (2) Manchmal auch Nummernzeichen oder Zeichen für Pfund genannt. Das „ # " Symbol.
Operand: (1) Der Teil einer Anweisung, der dem Opcode folgt. Er zeigt gewöhnlich an, wo die Operation im Speicher ausgeführt werden soll.
Operating System: (1) Betriebssystem. Ein Satz von Programmen in einem Computer, die dafür sorgen, daß allgemeine Arbeiten wie Eingabe/Ausgabe, Zeitgeberfunktionen usw. ausgeführt werden.
Operation Code, Opcode: (1) Der Teil einer Anweisung, die bestimmt, was zu tun ist.
Overflow: (3) Überlauf. Eine Bedingung, die durch eine Rechenoperation verursacht wurde, welche ein Ergebnis erzeugt hat, das für den zur Verfügung stehenden Platz zu groß ist.
Pointer: (6) Zeiger. Eine Adresse im Speicher, gewöhnlich 2 Bytes lang.
Processor Status Word, Status Register: (3) Ein Prozessorregister, das Statusflags enthält.
Pull: (7) Etwas vom Stapel herunternehmen.
Push: (7) Etwas auf dem Stapel ablegen.
Quellcode: (1) Anweisungen, in Assemblersprache geschrieben. Gewöhnlich der erste Code, der vom Programmierer vor einer Assemblierung geschrieben wird.
Random access memory, RAM: (1) Der Teil des Computerspeichers, in dem man Information speichern und abrufen kann.
Read only memory, ROM: (1) Der Teil des Computerspeichers, in dem eine feste Information abgelegt ist. Neue Information kann in einem ROM nicht gespeichert werden. Es ist vorprogrammiert.
Register: (1) Stelle in einem Prozessor, in der Information temporär gehalten werden kann.
Schreiben: (1) Information an eine Einheit senden.
Selbstmodifizierend: (7) Ein Typ von Programm, das sich während des Laufs selbst verändert. Eine seltene und nicht immer als gut zu empfehlende Programmierpraxis.
Speichern: (1) Übertragen von Informationen vom Prozessor in den Speicher. Bei der Speicheroperation handelt es sich um einen Kopiervorgang. Die Information verbleibt auch noch im Prozessor.
Stapel: (7) engl. stack. Ein temporärer Satz von Speicheradressen.
Status: Zustand
Statusregister, Prozessor Statuswort: (3) Innerhalb des Prozessors ein Register, das die Statusflags enthält.
Symbolische Adresse, Label: (7) Ein Name, der eine Speicherstelle identifiziert.
Symbolischer Assembler: (2) Ein Assembler, bei dem symbolische Adressen benutzt werden können. Dieser ist leistungsfähiger als ein nichtsymbolischer Assembler.
Testbare Flag: (3) Eine Flag, die mit Hilfe einer bedingten Verzweigungsanweisung getestet werden kann.
Unterbrechungsquelle: (7) Das spezifische Ereignis, das eine Unterbrechung verursachte. Da viele Dinge eine Unterbrechungsquelle darstellen können, ist es gewöhnlich notwendig, die spezifische Unterbrechungsquelle zu identifizieren.
Unterroutine: (2) Ein Satz von Anweisungen, die durch ein anderes Programm aufgerufen werden können.
Zahl mit Vorzeichen: (3) Eine Zahl, die einen Wert enthält, der entweder positiv oder negativ sein kann.
Zahl ohne Vorzeichen: (3) Eine Zahl, die keinen negativen Wert annehmen kann.
Zero page: (5) Nullseite. Die untersten 256 Speicherstellen. Stellen, deren Adressen mit hexadezimal $00.. beginnen.
Zweierkomplement: (3) Eine Methode, negative Zahlen darzustellen. Bei 1-Byte Zahlen würde $FF den Wert -1 repräsentieren.

Index

CARL HANSER VERLAG
NEUE FACHBÜCHER FÜR COMMODORE-ANWENDER

Fundierte Fachbücher kompetenter Autoren

Butterfield
Maschinenorientierte C64-Programmierung
Butterfield's Lehrbuch der Maschinen- und Assemblersprache für alle Commodore-Computer.
Von James Butterfield. Übersetzt von Dr. Karl F. Weibezahn, Bruchsal. Etwa 300 Seiten. 1985. Kartoniert.

Kwiatkowski
Textverarbeitung auf dem Commodore 64
Eine leicht verständliche Einführung mit Programmbeispielen.
Von Dr. Josef Kwiatkowski, Herne. Etwa 160 Seiten. 1985. Kartoniert.

Onosko
Der Commodore 64
Für Hobby, Schule und Beruf.
Von Tim Onosko. Übersetzt von Ilona und Dr. Josef Kwiatkowski. 356 Seiten. 1984. Kartoniert.

Stephenson
Denksport auf dem Commodore 64
Logik, Strategie, Mathematik.
Von John Stephenson. Übersetzt von Ilona Kwiatkowski, Dortmund. 213 Seiten mit vielen Programmen. 1985. Kartoniert.

Carl Hanser Verlag
Postf. 86 04 20, 8 München 86

Butterfield ist durch seine erfolgreichen Veröffentlichungen zum jetzt schon legendären Pet bis hin zu den heutigen Rechnerfamilien von Commodore zum Inbegriff für alle Anwender der Commodore-Computer geworden.

Dieses Buch führt den Anfänger in die Grundlagen der Maschinen- und Assemblerprogrammierung ein, es beschreibt, was Maschinen- und Assemblersprache ist, wie sie funktioniert und wie man damit programmiert. Vorkenntnisse werden nicht vorausgesetzt, es ist jedoch vorteilhaft, die Grundbegriffe der Programmierung in irgendeiner anderen Sprache zu kennen.

Für das Erlernen der Maschinen- und Assemblersprache gibt es drei gute Gründe: Die Geschwindigkeit dieser Sprachen, ihre unbegrenzten Möglichkeiten, und schließlich stellen sie den Schlüssel zum Verständnis der Maschinenoperationen dar.

ISBN 3-446-14378-5