Der einfache Einstieg in die faszinierende Computertechnik

Buch 1

JUNIOR COMPUTER

Elektor Verlag

Junior-Computer 1

Der einfache Einstieg in die faszinierende Computertechnik

A. Nachtmann
G.H. Nachbar

2. Auflage

ISBN 3-921608-18-X

Elektor Verlag GmbH, 5133 Gangelt 1

2. Auflage

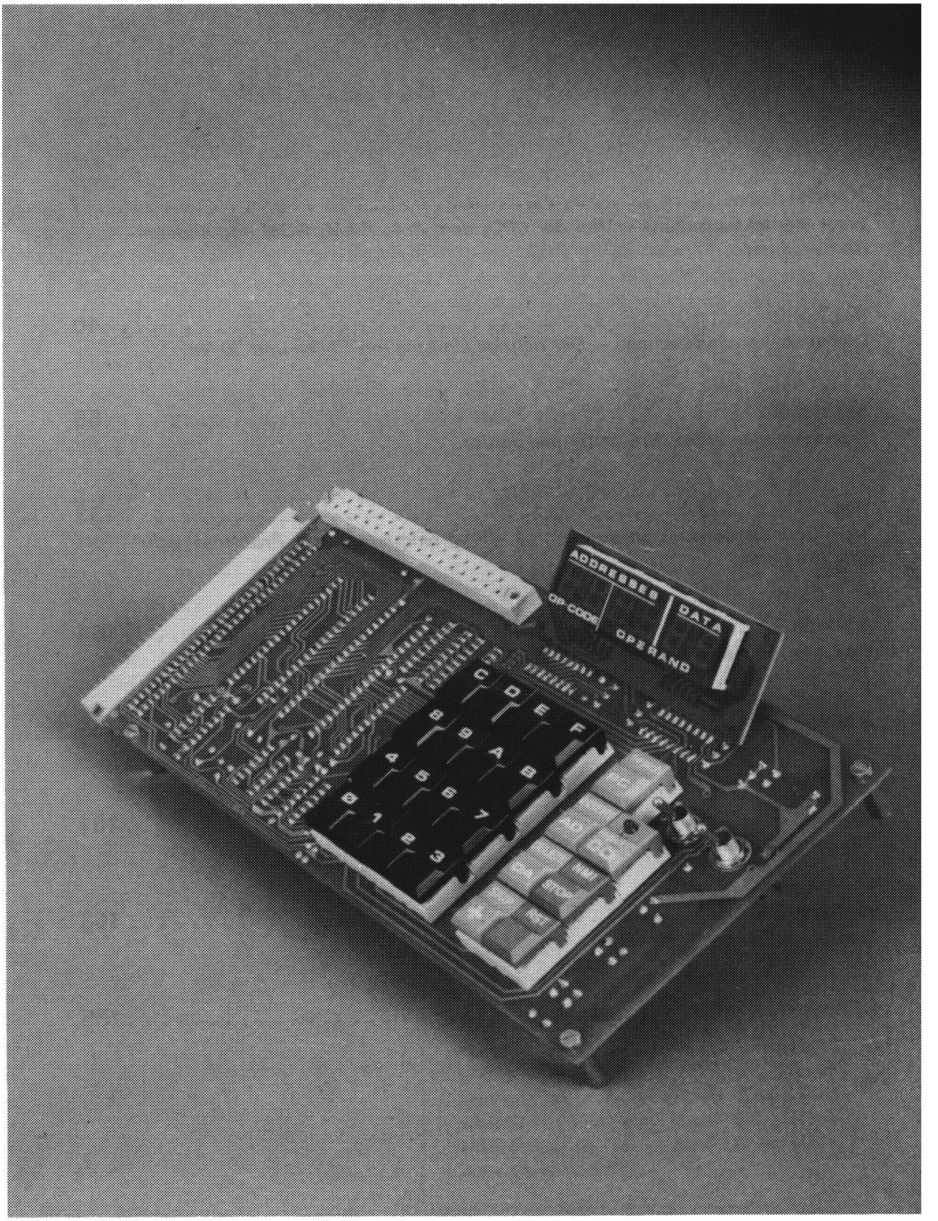
Ein Buch zum Junior-Computer?!

Sie haben sich also entschlossen, mit dem "Junior-Computer" die ersten Schritte in die Welt der Computer zu wagen – oder zunächst einmal diese Seite zu lesen. Wir wollen hier kurz aufzeigen, was Sie erwartet, und was hinter dem Konzept "Junior-Computer" steht.
Na, dann mal los!
Der Junior-Computer ist schon ein durchaus erwachsener Mikrocomputer, denn mit dem modernen 6502-Mikroprozessor bietet er von vornherein alle Möglichkeiten der angenehmen Programmierung und der späteren Erweiterung auf ein großes System "mit allen Schikanen".
Der Junior-Computer ist ein Selbstbau-Mikrocomputer – ein vollständiger Mikrocomputer auf einer Platine. Das spart nicht nur Platz und viele lästige Strippen, sondern auch Geld. Denn dieses Basis-System ist zusammen mit dem Buch alles, was man zum Lernen braucht.
Das Buch besteht aus vier Kapiteln.
Kapitel 1 enthält eine sehr ausführliche Bauanleitung, beginnt aber mit der Einführung in den Junior-Computer, der Beschreibung einzelner Funktionsblöcke.
Kapitel 2 beschäftigt sich schon mit den Anfängen des Programmierens – der Computer-Mathematik, dem Umgang mit "1"en und "0"en.
In Kapitel 3 geht's dann zur Sache. Hier wird der fertig aufgebaute Junior so richtig zum Leben erweckt. "Addresses", "Data", "OP-Code", "Operand", Flußdiagramm – zunächst noch ziemlich geheimnisvolle (Fremd-) Wörter werden im Verlaufe des Kapitels entzaubert.
In Kapitel 4 schließlich: Einige Programme zum Ausprobieren.
Da bekanntlich aller Anfang schwer ist, soll das Junior-Computer-Lernsystem schon jetzt etwas von der Leichtigkeit vermitteln, mit der man sich seinem Junior widmen kann, wenn man erst einmal etwas davon versteht.
In einem zweiten Buch wird der Hobby-Programmierer dann neue Anwendungs-Möglichkeiten seines "alten" Junior-Computers kennenlernen.
Wir hoffen, viel Neugier bei allen Interessenten zu wecken. Es liegt nun in jedermann(frau)s Hand, sie auch zu befriedigen!

Die Autoren

Über die Liefermöglichkeit der im Buch veröffentlichten Platinen gibt die jeweils gültige EPS-Liste (Elektor-Platinen-Service) in der Zeitschrift ELEKTOR Auskunft. Das gilt auch für weitere Informationen über den Elektor-Software-Service (ESS).

Inhalt


ADDRESSES
DATA
OP-CODE
OPERAND
C
D
E
F
8
9
A
B
4
5
6
7
0
1
2
3

1

Erste Kontaktaufnahme

Dieses Buch ist Teil einer mehrbändigen Reihe, deren Inhalt die Beschreibung eines Mikrocomputers für den Selbstbau ist. Das von Elektor auf der Basis des bekannten Prozessors 6502 entwickelte System erhielt den Namen "Junior-Computer" nicht ohne Grund: Es war ein besonderes Anliegen der Autoren, dem im Umgang mit der Computertechnik oft noch wenig vertrauten Hobby-Elektroniker die Schwellenangst zu nehmen, die häufig eine aktive Betätigung auf diesem faszinierenden Gebiet verhindert. Der Weg, der hier beschritten wird, ist konsequent an der Selbstbaupraxis orientiert. Er endet nicht als Sackgasse, sondern führt den Leser zu einem umfassenden Verständnis von Aufbau und Anwendung seines mit eigener Hand erbauten Mikrocomputers.

Bekanntschaft mit einem Computer

Im Prinzip ist ein Computer eine einfache Maschine. Wirft man jedoch zum ersten Mal einen Blick in das Innere eines Computers, so erhält man ein zunächst verwirrendes Bild. Die zahlreichen Bauelemente, Baugruppen und Kabelbäume wecken leicht Zweifel an den eigenen Fähigkeiten, einen Computer selbst zu bauen und später mit ihm zu arbeiten. In diesem Buch werden wir jedoch zeigen, daß es nicht schwer ist, einen Mikrocomputer in eigener Regie zusammenzusetzen, und daß es viel Freude bereitet, ihn zu programmieren und arbeiten zu lassen.

Die Aufgabe eines Computers ist das Ausführen von Befehlen, die der Programmierer zuvor in den Computer eingegeben hat. Bevor er jedoch diese Befehle eingeben kann, müssen sie zuerst in Form eines Programms geschrieben werden. Die Herausforderung, die der Computer an den

Anwender stellt, liegt nicht darin, eine Schaltung zu entwerfen, sondern ein vom Computer ausführbares Programm zu schreiben. Es ist dabei nicht wichtig zu wissen, wie zum Beispiel das Innenleben eines Mikroprozessors aussieht, sondern wichtig ist für den Benutzer nur, wie der Computer auf die verschiedenen Befehle reagiert. Vergleichen wir dies mit einem Auto: Um ein Auto fahren zu können muß man nicht wissen, wie der Motor, das Getriebe oder die Ölpumpe konstruiert sind.
Aus diesem Grund können wir den Junior-Computer als "Black Box" betrachten, als schwarzen Kasten, dessen Inhalt uns bei der Benutzung nicht zu interessieren braucht. Von großem Interesse sind dagegen die Blindeglieder des Computers zur Außenwelt, zu denen das Display und das Tastenfeld gehören. Mit Hilfe dieser Einrichtungen können wir den Computer bequem bedienen,ohne das Innenleben der einzelnen ICs zu kennen.

Der Junior-Computer, eine Black Box

Bevor der Junior-Computer als Black Box vor uns steht, müssen wir ihn zuerst aufbauen. Dieses Kapitel ist deshalb seinem Zusammenbau gewidmet. Einige Leser werden sich nun fragen, ob sie mit dem Bau eines Computers nicht überfordert sind. Diesen Lesern sei gesagt, daß der Junior-Computer in dieser Hinsicht keine allzu hohen Ansprüche stellt. Die Hardware, sein "Organismus", besteht im wesentlichen aus nur elf ICs. Damit dieses knappe Dutzend integrierter Schaltungen auf Anhieb seinen Zweck erfüllt, liefern wir Ihnen eine auf sämtliche Einzelheiten eingehende Baubeschreibung.

Ein bißchen Computerkunde

Das Spielen mit Blöcken oder Bausteinen ist uns seit früher Kindheit vertraut. Warum sollten wir also nicht einen Computer auf ein Blockschema zurückführen? Bild 1 zeigt ein solches Blockschema mit drei Arten von Verbindungen zwischen den einzelnen Blöcken. Ein Computer

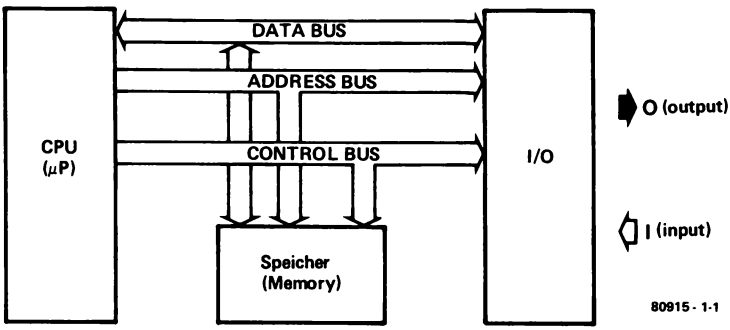


Bild 1. Einfacher geht's nicht: Das elementare Blockschaltbild eines Computers besteht aus drei Blöcken und drei Bussen. Busse sind "Drahtbündel", die die Computerbausteine CPU, Speicher und I/O elektrisch miteinander verbinden.

arbeitet an Hand von Informationen, die durch elektrische Signale dargestellt sind. Diese Informationen werden auch als Daten bezeichnet. Der Informations- oder Datenaustausch zwischen den einzelnen Blöcken findet über den Datenbus statt.
Ohne Außenwelt ist ein Computer eine nutzlose Maschine. Unter Außenwelt verstehen wir ein Tastenfeld (Keyboard), einen Bildschirm, ein 7-Segment-Display, einen Drucker oder ein Kassetteninterface. Als Zwischenstation zwischen Mikrocomputer und Außenwelt dient der Block I/O, eine Abkürzung von Input/Output. Über diesen Block wird der gesamte Datenverkehr von innen nach außen und umgekehrt gesteuert.
Die Daten auf dem Datenbus werden von der CPU bearbeitet. Die CPU oder Central Processing Unit ist das Gehirn des Mikrocomputers. Über die CPU werden alle Aufgaben im Mikrocomputer erledigt. Sobald eine bestimmte Aufgabe ausgeführt ist, sieht der Mikroprozessor im Speicher nach, wie die neue, folgende Aufgabe lautet. Die Aufgaben oder Befehle, die der Mikroprozessor ausführt, sind in Form von digitalen Codes im Speicher des Computers festgelegt. Alle Instruktionen zusammen, die der Computer bei einer bestimmten Anwendung ausführt, bilden das Programm.
Im Speicher des Computers sind also die Daten abgelegt, die der Programmierer zuvor eingegeben hat. Ein Computer ist von sich aus nicht intelligent. Das heißt, er kann keine eigenen Gedanken entwickeln. Aber vorgekaute Gedanken in Form eines Programms kann er mit unglaublicher Geschwindigkeit bearbeiten. Unter vorgekauten Gedanken ist zu verstehen: Der Computer führt nacheinander die Instruktionen aus, die ihm zuvor eingegeben wurden. Im Speicher des Computers sind nicht nur Daten vorhanden, die zuvor eingegeben wurden, sondern auch solche, die den Dialog mit dem Computer erst ermöglichen. Für das Arbeiten eines Computers sind Daten die Grundlage. Diese Daten sind zwar immer digital, können jedoch verschiedene Gesichter haben. Zum einen können diese Daten in Form eines Programms im Speicher abgelegt sein, zum anderen können diese Daten Pulse für die Steuerung eines Druckers oder irgend eines anderen am Computer angeschlossenen Geräts sein. Immer handelt es sich um elektrische Signale, wenn wir in Zukunft von Daten sprechen.
Zwischen der CPU und den zwei Blöcken in Bild 1 herrscht ein reger Datenverkehr. Um den Datenverkehr zwischen diesen Blöcken zu steuern, benötigt man die Adressierung, für die der Adreßbus zuständig ist. Das heißt, die CPU spricht über den Adreßbus denjenigen Block an, mit dem sie Daten austauschen will. Dabei können die Daten von der CPU ausgehen oder von einem der Blöcke zur CPU fließen.
Zum Schluß bleibt noch der Steuerbus oder Controlbus übrig. Dieser Bus ist für die interne Steuerung des Computersystems notwendig. Er enthält unter anderem ein Steuersignal, das die Richtung des Datentransports auf dem Datenbus festlegt. Die Steuersignale auf dem Controlbus gehen zum Teil von der CPU aus oder werden von außen zum Mikroprozessor geführt. Damit die CPU in der richtigen Reihenfolge Befehle oder Instruktionen ausführen kann, liefert sie zwei Taktsignale an den Steuerbus. Diese beiden Taktsignale, 1 MHz "schnell", sind der Herzschlag des Mikrocomputers.

Nun etwas technischer

Vom allgemeinen Blockschema in Bild 1 kommen wir zum Blockschema des Junior-Computers in Bild 2. Zuerst die drei Busse, die Schlagadern des Junior-Computers! Der Adreßbus besteht aus 16 elektrischen Leitungen. Jede Leitung kann unabhängig von den anderen zwei elektrische Zustände annehmen: entweder steht auf der Leitung Spannung oder nicht.

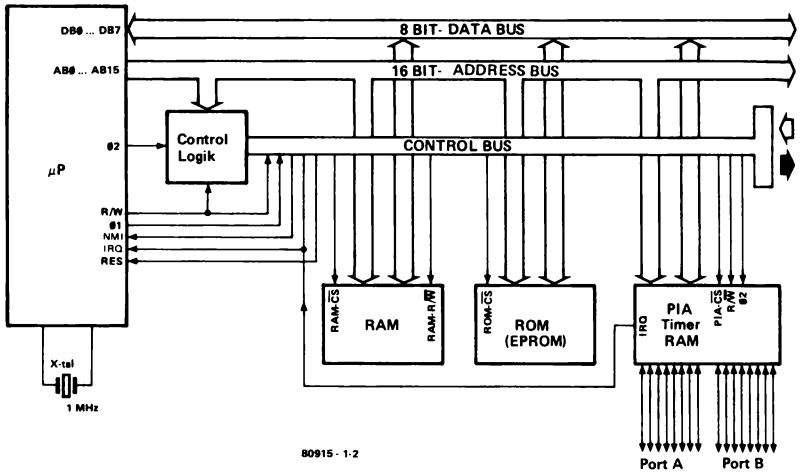


Bild 2. Das detaillierte Blockschaltbild des Junior-Computers. Gegenüber Bild 1 ist der Speicher in zwei Blöcke aufgeteilt: in RAM und ROM. Das RAM ist der Schreib-Lese-Speicher und das ROM der Lesespeicher des Junior-Computers.

Bei 16 Leitungen sind somit 2^{16} oder 65536 verschiedene Zustände möglich. Das sind mehr als 65-tausend verschiedene Plätze, die die CPU im Speicher adressieren kann. Zu jeder Adreßleitung gehört ein Bit. Deshalb spricht man auch von einem 16-bit-Adreßbus. Bit ist die kleinste Einheit einer digitalen Information. Was darunter zu verstehen ist, erklären wir im 2. Kapitel.

Auf dem Datenbus, der aus acht Leitungen besteht, läuft der gesamte Datenverkehr ab. Im Gegensatz zum Adreßbus ist auf dem Datenbus der Datentransport in zwei Richtungen möglich. Deshalb sagt man auch, der Datenbus ist ein bidirektionaler Bus. Selbstverständlich ist dabei, daß der Datenverkehr nicht gleichzeitig in beiden Richtungen fließt, sondern entweder von der CPU weg oder aber zum Prozessor hin.

Die Flußrichtung der Daten auf dem Datenbus legt das Lese/Schreib-Signal (Read/$\overline{\text{Write}}$ = R/$\overline{\text{W}}$-Signal) fest. Dieses Signal befindet sich auf dem Controlbus oder Steuerbus. Das R/$\overline{\text{W}}$ Signal läßt sich mit einem Verkehrszeichen, dem Schild "Einbahnstraße" vergleichen. Liest die CPU Daten aus dem Speicher oder aus der I/O, dann zeigt die Pfeilspitze des Verkehrszeichens "Einbahnstraße" auf die CPU. Das R/$\overline{\text{W}}$-Signal hat dann den Zustand logisch 1. Im anderen Fall, wenn die CPU Daten in den Speicher

oder in die I/O schreibt, zeigt dieses Verkehrszeichen von der CPU weg in Richtung Speicher oder I/O. Das R/$\overline{W}$ Signal hat in diesem Fall den Zustand logisch Ø. Um einen Datentransport in zwei Richtungen auf dem Datenbus zu ermöglichen, benötigt man digitale Bausteine mit sogenannten Tri-State-Ausgängen. Tri-State-Bausteine zeichnen sich dadurch aus, daß ihre Ausgänge nicht zwei, sondern drei Zustände annehmen können:

- keine Spannung vorhanden = log. Ø
- Spannung vorhanden = log. 1
- hochohmiger Zustand

Wegen des hochohmigen Zustands, den die Ausgänge dieser Schaltungen annehmen können, ist es möglich, auf nur einem Datenbus einen Datenverkehr in zwei Richtungen abzuwickeln.

Speicherelemente, die grauen Gehirnzellen des Computers

Ein Computer hat zwei Arten von Speicherelementen: einen Speicher, aus dem er Daten nur lesen kann, und einen Speicher, in dem er Daten sowohl ablegen als auch wieder auslesen kann. Beide Speicherarten sind für das Arbeiten eines Computers unentbehrlich. Die beiden Speicher sind in Bild 2 durch die Blöcke ROM und RAM dargestellt. Erinnern wir uns an Bild 1. Dort sind ROM und RAM zu einem Block, dem Speicher, zusammengefaßt. Warum aber benötigt man zwei verschiedenartige Speicherelemente beim Computer? Um mit einem Computer arbeiten zu können, benötigt man ein Programm, das immer vorhanden sein muß, das Monitor- oder Basisprogramm. Dieses Monitorprogramm ist immer in einem ROM untergebracht. Weiter arbeitet der Computer mit Daten, die veränderlich sind. Dazu gehört das Programm, das der Programmierer in den Computer eingibt (mit Hilfe des Monitorprogramms). Oder häufig muß das Programm korrigiert werden, bevor es richtig läuft. Auch dann müssen Daten im Speicher geändert werden. Daten, die sich in Bruchteilen von Sekunden ändern, kommen über den PIA oder die I/O (Bild 2) in den Computer oder umgekehrt.

Permanente Daten, das sind Daten, die sich während eines Programms nicht ändern, werden aus dem Speicher nur gelesen. Nichtpermanente Daten, das sind Daten die sich ständig ändern, werden sowohl in den Speicher gebracht als auch aus dem Speicher geholt. Wir sprechen dann von "Schreiben" in den Speicher oder "Lesen" aus dem Speicher. Deshalb gibt es im Computer Speicherelemente, aus denen ausschließlich Daten gelesen werden. Ein solches Speicherelement heißt ROM, eine Abkürzung von Read Only Memory. Der Speicher, in dem das Monitorprogramm des Junior-Computers abgelegt ist, hat einen ROM-Charakter, da nur Daten aus diesem Speicher gelesen werden.

Das andere Speicherelement ist das RAM, eine Abkürzung von Random Access Memory. Aus diesem Speicher lassen sich Daten lesen, es lassen sich jedoch auch Daten in ein RAM schreiben. Der Arbeitsspeicher des Junior-Computers hat deshalb einen RAM-Charakter. Zum RAM oder vom RAM erfolgt der Datenverkehr über den Datenbus in zwei Richtungen. Das R/$\overline{W}$-Signal auf dem Controlbus legt dabei fest, ob die CPU aus dem RAM Daten lesen oder in das RAM Daten schreiben möchte. Jetzt dürften die Begriffe Lesen und Schreiben bekannt sein.

Lesen heißt, daß die CPU Daten aus dem Speicher liest. Dabei werden die Daten, die an einem bestimmten Speicherplatz abgelegt sind, in die CPU kopiert. Nach dem Auslesen aus der betreffenden Speicherzelle sind die Daten in der Zelle nicht zerstört oder verlorengegangen. Man kann das mit dem Lesen eines Buchs vergleichen: Nach dem Lesen eines Buchs sind die Buchstaben auch nicht von den Seiten verschwunden! Beim Schreiben von Daten in eine bestimmte Speicherzelle des RAM werden die alten Daten von den neuen überschrieben.
RAMs verlieren ihren Inhalt, wenn ihre Speisespannung abgeschaltet wird. Sollen die Daten im RAM nach Abschalten der Speisespannung nicht verloren gehen, dann können sie auf eine Tonbandkassette oder eine Floppy Disc gespeichert werden. Von diesen peripheren Speichermedien lassen sich dann die Daten bei Bedarf wieder in den Computer zurückholen. Bei der Erweiterung des Junior-Computers gehen wir auf diese beiden Speichermedien noch genau ein.

Nebenbei bemerkt:

Daten, die sich in einem ROM befinden, werden während des Herstellungsprozesses in diesen Speicher gebracht. Man sagt deshalb, ein ROM ist maskenprogrammiert. Das heißt, die Information oder das Programm kommt über eine Herstellungsmaske in das ROM. Eine solche Maske hat Ähnlichkeit mit dem Film eines Platinenentwurfs. Das Monitorprogramm des Junior-Computers ist nicht in einem maskenprogrammierten ROM abgelegt, sondern in einem EPROM. EPROM ist eine Abkürzung von "Erasable, user Programmable ROM". Dieses Speicherelement kann man selbst programmieren mit einem EPROM-Programmer, einer "Dateneinbrennmaschine". EPROMs haben die angenehme Eigenschaft, daß sich die Daten in ihnen mit UV-Licht wieder löschen lassen. Nach dem Löschen kann dieses Speicherelement wieder neu programmiert werden.
Weitere Speicherelemente, die häufig in Computern zu finden sind, heißen PROM und EAROM. Das PROM kann man ebenfalls selbst programmieren. Allerdings ist die Information nach der Programmierung wie beim ROM nicht wieder löschbar. Die Daten im PROM werden von "fusable links" festgelegt, das sind Verbindungen auf dem Siliziumchip, die durch Stromimpulse zerstört werden. Meistens hat eine durchgebrannte Strecke die Bedeutung log. 1 und eine nicht durchgebrannte Strecke log. Ø. Somit lassen sich beliebige Daten in ein PROM einbrennen.
Das EAROM läßt sich wie das EPROM nach der Programmierung wieder löschen. Allerdings geschieht das Löschen nicht mit UV-Licht, sondern mit Stromimpulsen. Nach dem Löschen kann auch das EAROM wieder programmiert werden.

PIA, der I/O-Block des Junior-Computers

Wie wir bereits wissen, wird mit der I/O der Kontakt zwischen Computer und Außenwelt hergestellt. In Bild 2 ist der Block PIA gleichzusetzen mit I/O. PIA ist die Abkürzung von Peripheral Interface Adapter. Vom PIA gehen zwei Busse nach außen: Port A und Port B; sie sind auf einen eigenen Konnektor gelegt.
Neben Port A und B sind noch drei weitere Busse herausgeführt: Der Adreßbus, der Datenbus, und der Controlbus. Diese Busse sind für die

Erweiterung des Junior-Computers gedacht. Eine Erweiterung ist aber eine Vergrößerung der "Innenwelt" des Computers und deshalb noch nicht von Interesse. In unserem Fall besteht die Außenwelt des Junior-Computers aus dem hexadezimalen Keyboard und der sechsstelligen 7-Segment-Anzeige. Diese Außenwelt wird von der CPU mit Hilfe des PIA gesteuert. Die beiden Ports, die aus dem PIA herausführen, sind eine "Verlängerung" des Datenbusses. Deshalb haben die Ports wie der Datenbus eine Breite von 8 bit. Die Selektierung der Ports, die bestimmt, auf welches Port der Datenbus durchgeschaltet wird, erfolgt über den Adreßbus. Jede Portleitung kann unabhängig von den anderen als Eingang oder Ausgang programmiert werden. Wenn eine Portleitung als Eingang programmiert ist, kann die CPU über den Datenbus die Daten auf dieser Leitung lesen. Liegt beispielsweise auf einer als Eingang programmierten Portleitung ein serielles digitales Signal, dann kann die CPU diese serielle Information in eine parallele Information umwandeln. Dazu muß natürlich erst ein Programm geschrieben werden, das die CPU in die Lage versetzt, eine solche Umwandlung zu vollziehen. Im Buch 2, das jetzt noch in weiter Ferne liegt, werden wir lernen, solche Programme zu schreiben.
Ebenso wie sich die Portleitungen unabhängig voneinander als Eingänge programmieren lassen, ist es auch möglich, diese unabhängig voneinander als Ausgänge zu programmieren. Somit lassen sich die Daten auf dem Datenbus "in die Außenwelt schreiben". Damit das Lesen oder Schreiben aus oder in die Ports im Arbeitstakt der CPU erfolgt, sind vom Controlbus an den PIA einige Steuersignale gelegt.

Die CPU, das Aktionszentrum des Computers

Mit einem internen Taktgenerator, zu dem auch der externe 1-MHz-Quarz gehört, steuert die CPU über den Adreßbus, den bidirektionalen Datenbus und den Controlbus das komplette Computersystem. Der Taktgenerator liefert zwei Signale, Φ1 und Φ2, die für das Funktionieren des Computers von besonderer Bedeutung sind. Φ1 ist der Adreßtakt und Φ2 der Datentakt. Bevor ein Datentransport stattfinden kann, muß zuerst irgendeine Speicherzelle im Computersystem adressiert werden. Erst nach dieser Adressierung können die Daten transportiert werden. Der Takt Φ1 ist für die Stabilität der Information auf dem Adreßbus zuständig, während der Takt Φ2 für die Stabilität der Daten auf dem Datenbus verantwortlich ist. Da Φ1 und Φ2 eng miteinander verknüpft sind, spricht man von einem Zwei-Phasen-Takt. Ein wichtiges internes CPU-Register ist der Program Counter "PC" (in Bild 2 nicht gezeichnet!). Der Programmzähler führt den Mikroprozessor über den Adreßbus durch das Programm. Das heißt, daß Instruktionen oder Befehle mit dem Programm-Zähler sequentiell abgearbeitet werden. Adressen, der Grundstoff von Daten im Mikrocomputer, werden entweder direkt oder indirekt vom Programm abgeleitet. Auf alle Fälle findet sich der Mikroprozessor dank dieser Adressen im Programm zurecht.

Darf ich mal stören?

Über drei Signale von Bild 2 müssen wir noch sprechen. Das sind die Signale RES, IRQ und NMI. Beim RES-Signal können wir uns kurz fassen. Mit diesem Signal wird der Junior-Computer gestartet. Das heißt, nach

dem Anlegen der Netzspannung weiß die CPU nicht, an welcher Stelle sie mit dem Abarbeiten des Programms beginnen soll. Mit dem RES-Signal wird ihr jedoch mitgeteilt, daß sie sich um das Monitorprogramm des Junior-Computers kümmern soll. Die Signale IRQ und NMI dienen dazu, die CPU bei der Ausführung eines Programms auf konstruktive Weise zu stören. IRQ ist die Abkürzung von Interrupt ReQuest und NMI von Non Maskable Interrupt. Mit "Stören" ist eine Unterbrechung des laufenden Programmes gemeint, wenn sich der Computer um wichtige Dinge kümmern soll. Die Initiative einer solchen Störung kommt von außen. Dem Computer wird durch einen solchen Interrupt mitgeteilt, daß er sich schleunigst um eine gedrückte Keyboardtaste kümmern soll, daß ihm der Programmierer eine wichtige Mitteilung zu machen hat, oder daß ein peripherer Apparat, zum Beispiel ein Drucker, eine schnelle Bedienung erfordert. Nachdem die CPU einen solchen "Störpuls" an einem der Interrupteingänge empfangen hat, kümmert sie sich sofort um den "Störer". Nachdem dieser Störer bedient wurde, also wieder zufrieden ist, kümmert sich der Prozessor wieder um das Programm. Meistens "klopfen" mehrere Störenfriede zugleich an den Interrupteingängen der CPU an. In diesem Fall bekommt jeder Störenfried eine bestimmte Priorität zugewiesen. Je höher seine Priorität ist, desto schneller wird er behandelt. Ein wichtiger Unterschied zwischen IRQ und NMI ist, daß die CPU einen IRQ ignorieren kann, wenn der Programmierer ihr das erlaubt. Um einen NMI muß sie sich sofort kümmern, egal, ob sie will oder nicht.
Somit haben wir auch Bild 2 besprochen. Noch nicht ganz, denn im PIA ist noch eine weitere interessante Funktion enthalten, ein Timer. Dieser Timer läßt sich über das Programm programmieren; er hat die angenehme Eigenschaft, daß er unabhängig von der CPU läuft. Was heißt das? Häufig müssen beim Datenaustausch zwischen Computer und den angeschlossenen Geräten Zeitverzögerungen erzeugt werden. Denken wir nur an die Zeit, die benötigt wird, um ein Bit zu übertragen. Solche Zeitverzögerungen lassen sich mit dem Timer leicht realisieren, wobei sich die CPU unabhängig vom Timer um das Programm kümmern kann. Der Block "Control Logic" ist ein sog. Adreßdekoder. Die Control Logic erzeugt Chip-Select-Signale. Diese Signale legen fest, ob die CPU mit dem RAM, dem ROM oder dem PIA zusammenarbeiten will. So viel zum Innenleben des Junior-Computers. Jetzt wird es Zeit, den Junior-Computer von außen kennen zu lernen.

Keyboard und Display, die Eingabe/Ausgabe-Einheiten des Junior-Computers

Keyboard und Display sind die Eingabe/Ausgabe-Einheiten des Junior-Computers. Beides ist nötig, um mit dem Computer Kontakt aufzunehmen. Bild 3 zeigt, wie diese Einheit an den Junior-Computer angeschlossen ist. Daraus ist leicht ersichtlich, daß Keyboard wie Display an Port A und Port B angeschlossen sind. Ein hexadezimales Tastenfeld ist die einfachste Möglichkeit, um einen Computer zu füttern. Das hexadezimale Display ist nötig, um dem Programmierer zu zeigen, was er in den Computer getippt hat. Diese beiden hexadezimalen Eingabe- und Ausgabeorgane haben wir gewählt, um den Computer so preiswert wie möglich aufbauen zu können. Sicherlich kann man ohne Schwierigkeiten

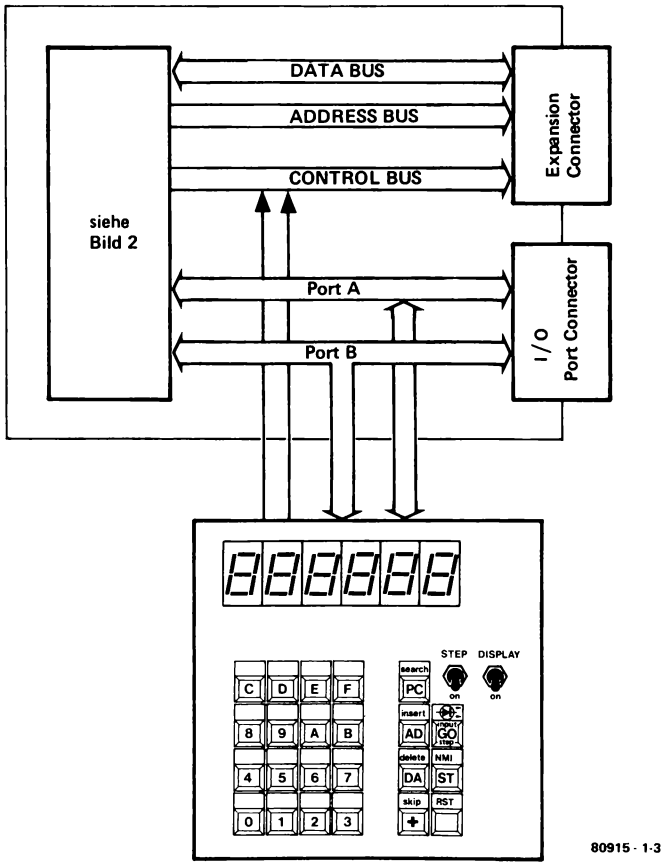


Bild 3. Der Anschluß von Display und Keyboard an die I/O-Leitungen Port A und Port B. Port A ist abwechselnd Eingang und Ausgang, während Port B nur als Ausgang verwendet ist. Die Keyboardtasten ST und RST führen direkt auf den Controlbus und werden vom Monitorprogramm nicht abgefragt.

an den Junior-Computer einen Drucker, ein Videoterminal oder ein anderes peripheres Gerät anschließen, aber dann wird das Projekt wesentlich teurer, und das wollen wir am Anfang vermeiden. In weiteren Büchern zeigen wir noch ausführlich, wie der Junior-Computer auf einfache Weise erweitert werden kann und sich schließlich zum "Senior-Computer" entwickelt. Da neben dem I/O-Konnektor auch ein Ausbreitungs-Konnektor (Expansion Connector) vorhanden ist, steht einer späteren Erweiterung nichts im Weg. Der Vollständigkeit halber sei noch erwähnt, daß der Expansion Connector 64-polig und SC/MP-Bus-kompatibel ist. Die 16 Leitungen von Port A und B sind auf einen 31-poligen Konnektor herausgeführt.

Wie schon gesagt, sind Keyboard und Display an Port A und Port B des peripheren Interface-Adapters PIA angeschlossen. Dabei gilt für Port A ein Zweirichtungsverkehr, für Port B Einrichtungsverkehr. Das heißt, Port A arbeitet einmal als Eingang und zum anderen als Ausgang, während Port B nur als Ausgang fungiert. Warum das so ist, hat folgenden Grund: Port B liefert mit Hilfe des Monitorprogramms die Scanpulse für das Keyboard und die Strobepulse für das Multiplexdisplay. Es muß also nur als Ausgang programmiert sein. Bei Port A ist es etwas anders: Zuerst wird über Port A ein eventuell gedrückter Key (= Taste) in den Computer eingelesen. Anschließend wird über den gleichen Port die 7-Segment-Anzeige bedient. Für das Einlesen eines gedrückten Keys ist Port A als Eingang programmiert. Für die Ansteuerung der Displaysegmente ist Port A als Ausgang programmiert. Bis jetzt können wir diese Einzelheiten noch nicht ganz verstehen, es wird jedoch klar, wie universell sich ein PIA einsetzen läßt.
Vom Keyboard führen noch zwei Verbindungen zum Controlbus. Die Verbindungen kommen von der RST-Taste und von der ST-Taste. Diese beiden Tasten sind Hardware-Tasten, das heißt, sie werden von der CPU nicht periodisch wie die übrigen Tasten des Keyboards abgefragt. Beim Umgang mit dem Junior-Computer werden wir das Keyboard und das Display noch genau kennen lernen, aber das kommt noch in späteren Kapiteln. Zuerst ist die genaue Beschreibung der Elektronik des Junior-Computers wichtig, denn bevor er programmiert werden kann, muß er gebaut werden. Also beschreiben wir das "Fleisch und Blut" des Junior-Computers: Die Hardware.

So ist er zusammengesetzt

Bild 4 zeigt, wie der Junior-Computer zusammengesetzt ist. Das Schaltbild soll jetzt besprochen werden! Beginnen wir mit IC1. Das ist der Mikroprozessor. Erfahrene "Computerfüchse" werden versichern, daß dieser Typ, der 6502, nicht der erste beste Prozessor ist. Diese CPU ist ein wahrer Leckerbissen. Warum? Lassen wir den Prozessor für sich selbst sprechen:

- Wenige, aber sehr wirkungsvolle und vielseitige Befehle (der Computer ist dadurch leicht programmierbar);
- 6502, die CPU mit den meisten Adressierungsarten. Adressierungsarten, die bei anderen Prozessoren erst in der dritten Generation auftauchen, waren bei dieser CPU bereits in der ersten Generation vorhanden;
- Übersichtliche, leicht erlernbare Mnemonics (= Stenoschreibweise für Maschinenbefehle);
- Die am weitesten verbreitete CPU bei Computeramateuren.

Man könnte diese Liste noch fortsetzen, aber es dürfte klar sein, warum wir für den Junior-Computer gerade diesen und keinen anderen Prozessor ausgewählt haben.
Für den Mikroprozessor ist eine geringe externe Beschaltung nötig, damit sein Herz, der Taktgenerator schlagen kann. N1, R1, D1, C1 und der 1-MHz-Quarz sind die Schrittmacher für den internen Taktgenerator. Wie schon erwähnt, schlägt das Herz der CPU 6502 mit einer Taktfrequenz von 1 MHz. Dabei ist deutlich zu erkennen, wie die Taktsignale $\Phi 1$ und $\Phi 2$,

die für das Präparieren des Adreß- und Datenbusses zuständig sind, auf den Controlbus geleitet sind.
Vom Prozessor (IC1) geht auch der Adreßbus aus. Er besteht aus den Leitungen AØ . . . A15. Für den Datenbus mit den Leitungen DØ . . . D7 ist IC1 die "Sende- und Empfangsstation", da auf diesem Bus ein Datenverkehr in zwei Richtungen erfolgt. Auf diesen beiden Bussen liegen Informationen in Form von elektrischen Signalen. Wie? Ganz einfach! Die Adreß- wie die Datenleitungen sind in einer bestimmten Reihenfolge geordnet. Beim Adreßbus gilt die Reihenfolge A15 . . . AØ und für den Datenbus D7 . . . DØ. Jede dieser Leitungen kann Spannung führen oder nicht. Führt eine Adreß- oder Datenleitung Spannung, so ist sie log. 1, führt sie keine Spannung, so ist sie log. Ø. Schreiben wir nun die Nullen und Einsen auf dem Adreßbus (A15 . . . AØ) der Reihenfolge nach von links nach rechts auf ein Blatt Papier, so erhalten wir eine binäre Zahl, die aus 16 Elementen besteht. Dasselbe gilt für den Datenbus. Allerdings sind auf ihm nur Kombinationen aus acht Elementen möglich. Die Zahlen oder genauer gesagt die Null-Eins-Muster auf diesen beiden Bussen stellen einen Code dar. Ein solcher Code auf dem Adreßbus ist identisch mit einer Adresse, und ein Code auf dem Datenbus ist identisch mit Daten, die im Computer verarbeitet werden. Häufig werden die Null-Eins-Muster auf Adreß- und Datenbus als "Wort" bezeichnet. Da der Adreßbus 16 Leitungen hat, lassen sich auf ihm Worte mit einer Breite von 16 bit darstellen. Auf dem Datenbus lassen sich nur Worte mit einer Breite von 8 bit darstellen, da dieser nur acht Leitungen hat. Weil ein Computer eine binäre digitale Maschine ist, versteht er eigentlich nur zwei Zahlen: null und eins oder technisch: Spannung vorhanden, keine Spannung vorhanden.

Die Speicherorganisation des Junior-Computers

Betrachten wir nochmals das Schaltbild des Junior-Computers in Bild 4. Dort sind über den Adreß- und Datenbus die Speicherelemente des Junior-Computers mit dem Mikroprozessor IC1 verbunden. Eines dieser Speicherelemente ist das EPROM IC2, in dem das Monitorprogramm untergebracht ist. Aus dem EPROM lassen sich nur Daten lesen. Den Schreib-Lese-Speicher bilden die beiden RAMs IC4 und IC5. Wie der Name schon sagt, lassen sich aus diesem Speicher Daten lesen oder in ihn Daten hineinschreiben.
Die Daten, die auf dem Datenbus transportiert werden, setzen sich aus 8 bit zusammen. Deshalb müssen auch die Speicherplätze im EPROM und im RAM acht bit breit sein. In Zukunft bezeichnen wir ein 8 bit breites Wort als "Byte". Das EPROM enthält 1024 Speicherplätze, von denen jeder 8 bit, also ein Byte speichern kann. Rundet man 1024 ab, so spricht man von 1 K, wobei K Tausend oder Kilo bedeutet. Alle Bytes im EPROM müssen eindeutig aufgerufen werden können. Mit zehn Adreßleitungen ist das leicht möglich, denn mit AØ . . . A9 sind 2^{10} oder 1024 Speicherzellen adressierbar.
In den RAMs lassen sich nur 1024 halbe Bytes mit den Adreßleitungen AØ . . . A9 adressieren. Deshalb sind für die Speicherung eines ganzen Byte im Schreib-Lese-Speicher zwei ICs nötig: eins für die Speicherung der Datenbits D7 . . . D4 und ein anderes für die Datenbits D3 . . . DØ. Mit IC4 und IC5 wird das technisch realisiert, wobei beide ICs gleich-

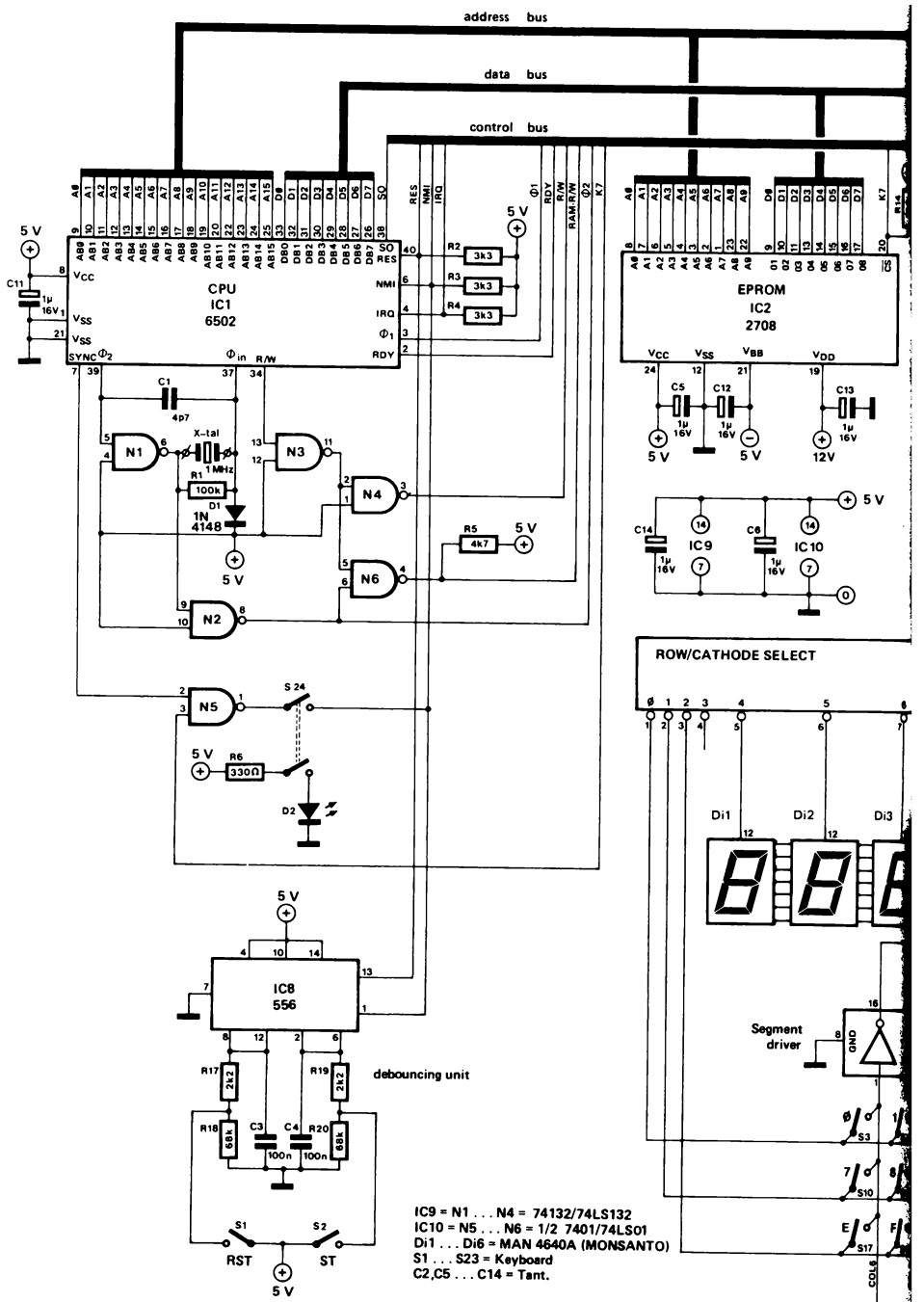
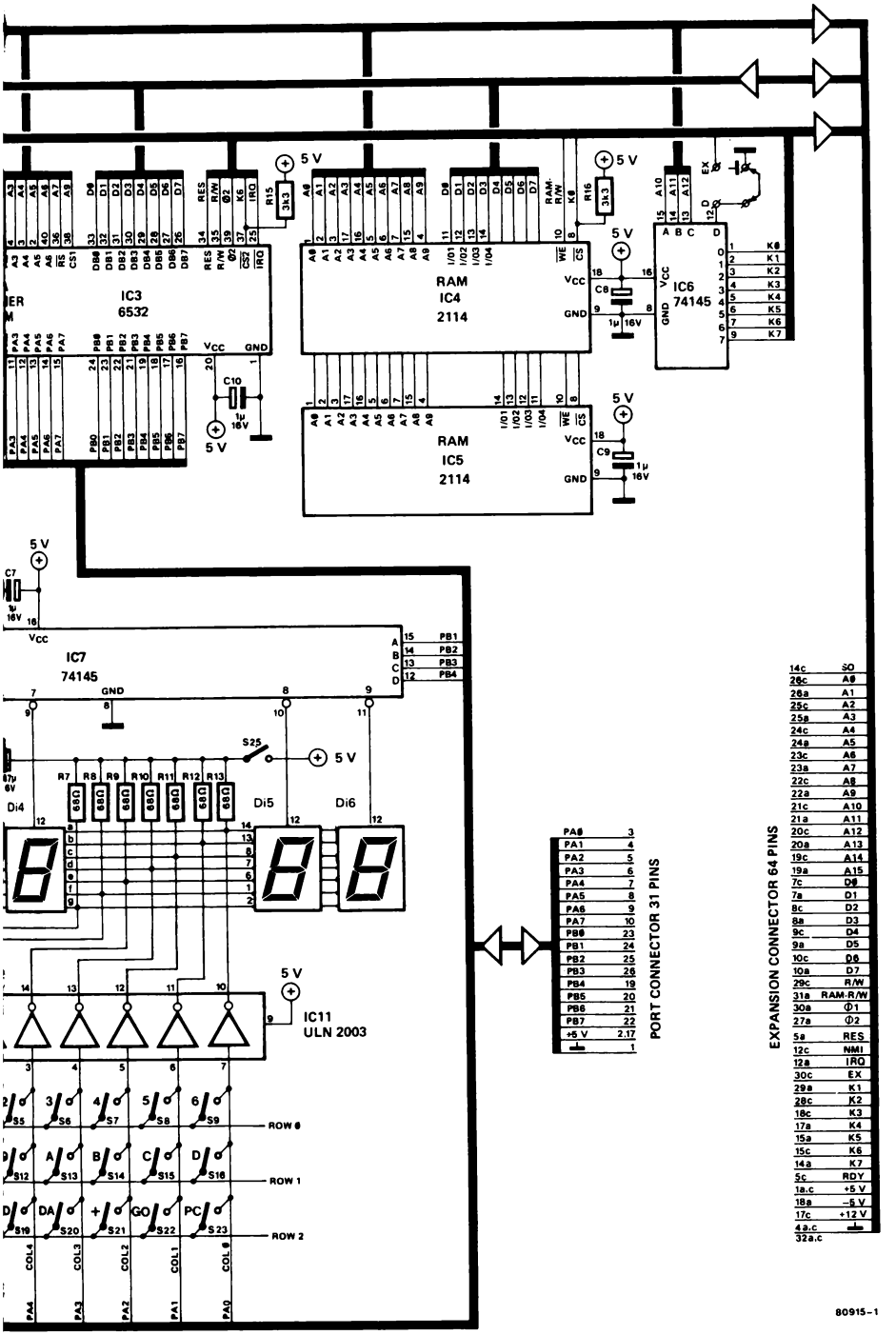


Bild 4. Das Schaltbild des Junior-Computers. Der Übersichtlichkeit wegen sind die Busleitungen zu einem dicken Bündel zusammengefaßt.



PA0	3
PA1	4
PA2	5
PA3	6
PA4	7
PA5	8
PA6	9
PA7	10
PB0	23
PB1	24
PB2	25
PB3	26
PB4	19
PB5	20
PB6	21
PB7	22
+5 V	2,17
⏚	1

14c	SO
26c	A0
26a	A1
25c	A2
25a	A3
24c	A4
24a	A5
23c	A6
23a	A7
22c	A8
22a	A9
21c	A10
21a	A11
20c	A12
20a	A13
19c	A14
19a	A15
7c	D0
7a	D1
8c	D2
8a	D3
9c	D4
9a	D5
10c	D6
10a	D7
29c	R/W
31a	RAM-R/W
30a	Φ1
27a	Φ2
5a	RES
12c	NMI
12a	IRQ
30c	EX
29a	K1
28c	K2
18c	K3
17a	K4
15a	K5
15c	K6
14a	K7
5c	RDY
1a,c	+5 V
18a	−5 V
17c	+12 V
4a,c 32a,c	⏚

zeitig vom Adreßbus adressiert werden.
EPROM und RAM bekommen auch noch vom Steuerbus Signale, sogenannte Selektionssignale. Da beide Speichermedien parallel an den Adreßbus angeschlossen sind (AØ . . . A9), muß zwischen ihnen eine Trennung erfolgen, um eine Doppeladressierung zu vermeiden. Diese Trennung erfolgt über die CS-Signale KØ und K7. Beide Trennungssignale kommen vom Adreßdekoder IC6. KØ ist für das RAM und K7 für das EPROM zuständig. Steht auf einer CS-Leitung Spannung, also log. 1, dann ist der entsprechende Speicher vom Datenbus entkoppelt. Das heißt, die Datenleitungen des Speichers nehmen einen hochohmigen Zustand an. Wird aber eine CS-Leitung log. Ø, dann kann der an ihr angeschlossene Speicher am Datenverkehr, teilnehmen. Man könnte auch sagen, die CPU möchte mit dem betreffenden Speicher einen Datenaustausch vornehmen.
Zehn Adreßleitungen waren nötig, um sowohl im RAM als auch im EPROM 1024 verschiedene Speicherzellen zu adressieren. Da der Adreßbus 16 Leitungen hat, bleiben noch sechs Leitungen übrig. Mit diesen sechs Leitungen lassen sich $2^6 = 64$ Möglichkeiten darstellen. Somit lassen sich 64 Speicherblöcke von jeweils einem KByte adressieren. Das stimmt mit der Rechnung überein, daß sich mit 16 Adreßleitungen 64 KByte adressieren lassen. Wieso? Das nullte KByte ist ebenfalls ein Speicherblock mit 1024 Speicherplätzen. Somit lassen sich über 6 Leitungen $2^6 = 64$ Blöcke von 1 KByte, also 64 KByte adressieren! Das sind immerhin 65536 Speicherzellen.
Mit 64 CS-Leitungen läßt sich der Adreßbereich in 64 1-KByte-Blöcke einteilen, wobei zu jedem KByte-Block die Adreßleitungen AØ . . . A9 und das betreffende CS-Signal gehören. In der Standardausführung des Junior-Computers wird nur von acht CS-Leitungen Gebrauch gemacht. Nämlich von den CS-Signalen KØ . . . K7. Diese CS-Signale erzeugt der Adreßdekoder IC6, dem die folgenden Adreßleitungen A1Ø . . . A12 zugeführt werden. Drei von diesen acht CS-Signalen benötigt der Junior-Computer für die interne Steuerung, die restlichen fünf sind auf den Expansion-Connector für eine spätere Systemerweiterung herausgeführt. Nachdem KØ und K7 für die Selektion zwischen RAM und EPROM Verwendung finden, ist K6 an den PIA (IC3) angeschlossen. Eine Übersicht über die CS-Signale gibt die folgende Tabelle:

A15 . . . A13	A12	A11	A1Ø	A9 . . . AØ	aktiv	Speicherblock:
X	Ø	Ø	Ø	X	KØ	1 K RAM (IC4, IC5)
X	Ø	Ø	1	X	K1	1 K extern
X	Ø	1	Ø	X	K2	1 K extern
X	Ø	1	1	X	K3	1 K extern RAM, ROM
X	1	Ø	Ø	X	K4	1 K extern
X	1	Ø	1	X	K5	1 K extern
X	1	1	Ø	X	K6	RAM in PIA, Ports, Timer (IC3)
X	1	1	1	X	K7	1 K EPROM (IC2)

Diese Tabelle enthält mehrere Spalten, die bestimmten Bits des Adreßbusses zugeordnet sind. Die erste Spalte, die die Adreßbits A13 . . . A15 enthält, ist für den Adreßbereich des Junior-Computers nicht relevant, da

diese Adreßleitungen nicht am Adreßdekoder IC6 angeschlossen sind. Die fünfte Spalte enthält die Adreßbits AØ . . . A9. Diese Adreßbits wählen aus einem KByte-Speicherbereich die adressierten Speicherzellen aus. Auch diese Spalte hat mit der eigentlichen Adreßdekodierung nichts zu tun. Die "X" in der ersten und fünften Spalte stehen für "don't care", was soviel wie belanglos heißt.

Die Spalten A12, A11 und A1Ø sind dagegen für die Adreßdekodierung des Junior-Computers um so wichtiger. Mit diesen drei Adreßbits lassen sich acht Speicherblöcke, die parallel an den Adreßbus AØ . . . A9 angeschlossen sind, voneinander trennen. Die Trennungssignale dieser Speicherblöcke heißen KØ . . . K7. Nicht alle dieser acht Signale finden bei der internen Adreßdekodierung Verwendung. KØ ist auf der Basisplatine des Junior-Computers dem RAM (IC4, IC5) zugeordnet. Wenn diese Leitung log. Ø ist, dann möchte die CPU mit dem RAM-Speicher in Kontakt treten. Das ist der Fall, wenn A12 . . . A1Ø Null ist. Die CS-Leitung K7 ist für das EPROM IC2 zuständig. Immer wenn die CPU aus diesem ROM Daten liest, ist K7 log. Ø. Dieser Fall tritt ein, wenn A12 . . . A1Ø Spannung führen oder 111 sind. Das CS-Signal K6 ist aktiv, wenn die CPU mit dem PIA (IC3) Daten austauschen will. Wenn auf den Adreßleitungen A12 . . . A1Ø das Bitmuster 11Ø steht, wird K6 aktiv. Die übrigen CS-Signale K1 . . . K5 finden bei einer späteren Erweiterung des Junior-Computers Verwendung. Sie dienen dazu, bei dieser Erweiterung EPROMs anzusteuern, die die Programme für die Ansteuerung eines Druckers (TTY), des Elekterminals und für das Arbeiten mit zwei Cassettenrekordern enthalten. Doch das ist in einem folgenden Buch beschrieben. Für die Steuerung des RAMs (IC4, IC5) ist noch ein weiteres Signal nötig: RAM-R/$\overline{W}$. Dieses Steuersignal entscheidet, ob Daten über den Datenbus zum RAM transportiert oder aus diesem gelesen werden. Ist RAM-R/$\overline{W}$ log. 1 und das RAM über die CS-Leitung KØ aktiviert, dann liest die CPU Daten aus einer bestimmten RAM-Zelle. Das Gegenteil tritt ein, wenn RAM-R/$\overline{W}$ log. Ø ist; dann schreibt die CPU Daten in eine RAM-Zelle, die über AØ . . . A9 angewählt ist. Genauer gesagt: die CPU überschreibt die alten Daten in dieser RAM-Zelle. Dieses RAM-R/$\overline{W}$ Signal ist eine Kombination des R/$\overline{W}$-Signales der CPU und des Taktsignales Φ2. Dadurch ist mit Sicherheit das Schreiben von Daten in eine RAM-Zelle ausgeschlossen, solange die Daten auf dem Datenbus nicht stabil stehen.

Einige Steuersignale auf dem Controlbus des Junior-Computers haben wir nun besprochen. Andere wichtige Signale auf diesem Bus wollen wir jetzt kennenlernen. Mit dem Resetsignal werden der Mikroprozessor IC1 und der PIA IC3 in Startposition gebracht. Das geschieht, wenn die RES-Leitung für einige Mikrosekunden log. Ø ist. Diese Leitung ist im Normalzustand log. 1, da sie mit dem Pullup-Widerstand R2 auf das Niveau der Versorgungsspannung gezogen wird. Mit der Taste RST läßt sich der Reset auslösen. Um das Kontaktprellen dieser Taste zu beseitigen, ist IC8 vorgesehen. Dieses IC hat auf seinem Chip zwei Timer. Einer von ihnen entprellt die RST-Taste, der andere die ST-Taste, auf die wir gleich näher eingehen. Die NMI-Leitung gehört ebenfalls zum Controlbus. Diese Leitung ist im Gegensatz zur RES-Leitung flankensensitiv. Das heißt, ein negativer Impuls am NMI-Pin des Mikroprozessors löst einen Interrupt aus. Wie das im einzelnen aussieht, besprechen wir am Ende des 3. Kapitels

in diesem Buch. Wie aus dem Schaltbild (Bild 4) hervorgeht, liegt die NMI-Leitung im Normalzustand über Widerstand R3 auf log. 1. Sowohl mit der Taste ST und dem Ausgang des NAND-Gatters N5 kann auf dieser Leitung eine negative Flanke hervorgerufen werden (wired OR). Mit der ST-Taste läßt sich mit Hilfe des Monitorprogramms ein laufendes Programm unterbrechen, mit Hilfe des Gatters N5 läßt sich der Junior-Computer in "step by step mode" bringen. Ist Schalter S24 geschlossen (ON), so lassen sich Programme schrittweise testen. Der Computer arbeitet dann nur einen einzigen Befehl ab und wartet danach, bis er aufgefordert wird, den folgenden Befehl auszuführen. Dadurch lassen sich eventuelle Fehler in Programmen sehr leicht aufspüren. Da "step by step mode" durch den Monitor im EPROM gesteuert wird, muß ausgeschlossen sein, den Monitor in diesem Betriebszustand zu durchlaufen. Deshalb ist auch das CS-Signal K7 an einen Eingang des Gatters N5 gelegt.
Der eigentliche Programmschritt in "step by step mode" wird vom SYNC-Ausgang der CPU gesteuert. Immer wenn der Mikroprozessor einen Befehl aus dem Speicher holt, geht dieser Ausgang auf log. 1. Das hat einen NMI zur Folge, wodurch der Junior-Computer in eine Warteschleife gezwungen wird.
Auch der IRQ-Eingang des Mikroprozessors ist an eine Leitung des Controlbusses angeschlossen. Diese Leitung wird durch Widerstand R4 auf log. 1 gehalten. Der IRQ-Ausgang des PIA IC3 ist an die IRQ-Leitung angeschlossen. Ein IRQ läßt sich nur dann auslösen, wenn das im Programm ausdrücklich erlaubt ist. Der IRQ-Pin der CPU ist wie der RES-Pin niveausensitiv und muß, um einen IRQ auszulösen, für einige Mikrosekunden log. Ø sein.
Die Φ2-Leitung des Controlbusses ist auch mit IC3, dem PIA verbunden. Das ist notwendig, da sich in dieser Schaltung ein Timer befindet, der sich vom Programm beeinflussen oder besser gesagt programmieren läßt. Diesen Timer, der unabhängig vom Mikroprozessor läuft, steuert das Taktsignal Φ2. Auch die R/$\overline{W}$-Leitung ist mit IC3 verbunden. Das ist nötig, da sich in diesem Universalbaustein ein RAM von 128 Byte sowie verschiedene Register befinden.
Zwei nicht verwendete Pins der CPU sind ebenfalls mit dem Controlbus verbunden: RDY und SO. Will man dynamische RAMs an den Junior-Computer anschließen, dann ist das RDY-Signal wichtig. SO dient für spätere Hardwareerweiterungen. Zusammenfassend liegen auf dem Controlbus folgende Signale:

- R/$\overline{W}$ — Read/Write-Signal
- RAM-R/$\overline{W}$ — Read/Write-Signal für RAM
- KØ . . . K7 — Chip-Select-Signale
- Φ und Φ2 — Taktsignale für Adreß- und Datenbus
- RES — Resetleitung
- IRQ und NMI — Interruptleitungen
- RDY — Steuerung von dynamische RAMs
- SO und EX — Hardwareerweiterung des Computers
  (EX siehe IC6)

Der PIA, der "heiße Draht" zur Außenwelt

Das Schaltbild (Bild 4) enthält einen weiteren wichtigen Computerbau-

stein, den PIA (IC3). Er hat zwei Ports: Port A und Port B. Wie schon zuvor erwähnt, läßt sich auf diese Ports der Datenbus des Junior-Computers durchschalten. Mit welchem Port die CPU zusammenarbeiten möchte, ist vom Programm festgelegt. Die Selektion der Ports erfolgt mit den niederwertigsten Adreßleitungen AØ . . . A3.
Zu jedem Port gehört ein I/O-Register und ein Datenrichtungsregister. Beide Register sind 8 bit breit. In beide lassen sich sowohl Daten schreiben, es lassen sich aus ihnen aber auch Daten lesen. Schreibt die CPU in das Datenrichtungsregister ein bestimmtes Bitmuster, so sind die Ein- und Ausgänge des betreffenden Ports festgelegt: Eine "Ø" im Datenrichtungsregister erklärt die korrespondierende Leitung zum Eingang, eine "1" die betreffende Leitung zum Ausgang. Das R/$\overline{W}$-Signal ist ebenfalls mit dem PIA verbunden. Ist dieses Signal, das der Controlbus liefert, log. 1, so kann die CPU die Daten lesen, die außen am Port anliegen. Unter Daten sind dabei Spannungen zu verstehen, die einen TTL-Pegel haben. Ist die R/$\overline{W}$-Leitung aber log. Ø, so kann die CPU Daten in einen der beiden Ports schreiben.
IC3 enthält auch noch ein 1/8-K-RAM. Zusammen mit den 1024 Bytes in IC4 und IC5 sind 1152 Bytes RAM auf der Basisplatine des Junior-Computers vorhanden. Zur Adressierung dieser 128 Speicherzellen in IC3 dienen die Adreßleitungen AØ . . . A6. Die Adreßleitung A7 ist mit dem Pin $\overline{RS}$ verbunden. Diese Adreßleitung entscheidet, ob die CPU mit dem RAM oder den Ports und dem Timer einen Datenaustausch wünscht. Ist A7 log. Ø, dann ist das RAM angesprochen. Der Timer sowie die Ports sind aufgerufen, wenn A7 log. 1 ist. Um soweit wie nur möglich mit geringen Mitteln eine nahezu lückenlose Adreßdekodierung zu erreichen, sind an den PIA zwei CS-Signale herangeführt. CS2 ist K6 vom Adreßdekoder IC6, während CS1 mit der Adreßleitung A9 verbunden ist. CS2 ist für die Gesamtadressierung von IC3 zuständig, während durch A9 (= CS1) der letzte Teil eines 1K-Speicherblock festgelegt ist. Über den PIA und den Timer läßt sich noch viel sagen. Deshalb haben wir diesem interessanten Baustein in Buch 2 ein ganzes Kapitel gewidmet.

Tastenfeld und Display- die Außenwelt des Junior-Computers

Tastenfeld und Display sind die eingebaute Außenwelt des Junior-Computers. Eingebaut deshalb, weil beides auf der Basisplatine fest verdrahtet ist. Vier Leitungen von Port B und sieben Leitungen von Port A sind nötig, um mit etwas Hardware (IC7 und IC11) die Eingabe-Ausgabe-Einheit zu bedienen. Wie aus dem Schaltbild ersichtlich ist, enthält das Keyboard 23 Tasten, die in einer Matrix angeordnet sind. Diese Matrix hat drei Reihen (ROWØ . . . ROW2) und sieben Spalten (COLØ . . . COL6). 16 Tasten des Keyboards dienen zur Dateneingabe und die restlichen fünf sind Kommandotasten, die eine Steuerfunktion haben. Doch wie mit diesen Tasten gearbeitet wird, erklären wir im Kapitel 3 dieses Buches.
Daten zum Display und Daten vom Keyboard laufen über die sieben Leitungen von Port A. Dabei bedient sich der Junior-Computer der Eigenschaft, mit nur einem Port in zwei Richtungen zu arbeiten. Die Steuerung des Keyboards und des Displays erledigt das Monitorprogramm im EPROM. Es bedient periodisch die Ein-Ausgabe-Einheit. Dasselbe Programm fragt auch ab, ob irgendeine Taste des Keyboards gedrückt ist. Wenn ja, dann

dekodiert das Programm die betätigte Taste. Sogar das Kontaktprellen dieser Tasten wird über Software beseitigt.
Da die Portleitungen nur eine TTL-Last steuern können, benötigt man zur Steuerung des Displays eine Zwischenstation, die die relativ hohen Segmentströme steuern kann (to scan = abtasten). Zum einen ist es der BCD-Dezimal-Umsetzer IC7 und zum anderen der Segmenttreiber IC11. Der BCD-Dezimalumsetzer hat zehn Ausgangsleitungen, die in Abhängigkeit vom Bitmuster auf den vier Eingangsleitungen (A . . . D) log. Ø sind. Die Eingänge des Umsetzers steuern die vier Portleitungen PB1 . . . PB4.
Die drei Reihen der Keyboardmatrix sind mit den ersten drei Ausgängen von IC7 verbunden. Ein Ausgang ist unbenutzt, und die Ausgänge 4 . . . 9 steuern die gemeinsamen Katoden der sechs Displays an. Wird eine dieser Katoden durch einen Ausgang von IC7 auf Massepotential gezogen, dann kann man jedes beliebige Segment im betreffenden Display aufleuchten lassen. Ein Segment leuchtet dann auf, wenn der zu ihm gehörende Puffereingang von IC11 log. Ø ist. Die Segmente der Displays steuert ebenfalls das Monitorprogramm des Junior-Computers. Durch das Bitmuster auf den Leitungen PAØ . . . PA6 werden die Symbole auf dem Display dargestellt.
IC7 steuert auch das Tastenfeld an. Ist eine beliebige Taste der Matrix niedergedrückt, dann erkennt das Programm die betätigte Taste. Daß überhaupt eine Taste betätigt wurde, erkennt das Monitorprogramm daran, daß irgendeine Port-A-Leitung log. Ø ist. Zwei Tasten sind nicht mit der Matrix verbunden: Die RST-Taste und die ST-Taste. Diese beiden Tasten sind über eine Entprellungsvorrichtung (IC8) mit der CPU verbunden. Somit besteht das Tastenfeld aus den Softwaretasten S3 . . . S23 und den Hardwaretasten S1 und S2.
Zwei Schalter sind noch auf der Basisplatine der Junior-Computers vorgesehen: S24 und S25. Mit dem ersten Schalter läßt sich der Computer in "step by step mode" bringen und mit dem zweiten kann man das Display ausschalten. Das ist wichtig, wenn die Port-A-Leitungen PAØ . . . PA6 ein externes Gerät steuern sollen. Einem wilden Flackern der 7-Segment-Anzeigen wird somit entgegengewirkt.
Damit der Junior-Computer arbeiten kann, benötigt er ein Netzteil, das drei Spannungen liefert. In Bild 5 ist eine Standardschaltung dargestellt, die sich aus nur drei ICs und ein paar Kondensatoren nebst Gleichrichter zusammensetzt. Das Netzteil liefert folgende Spannungen:

+ 5 V, Dauerbelastung 1 A
−5 V, Dauerbelastung 100 mA
+12 V, Dauerbelastung 100 mA

Um die Verlustleistung des +5-V-Stabilisators gering zu halten, benötigt man am Eingang eine niedrige Trafospannung. Um einen Standardtrafo mit zwei gleichen Sekundärwicklungen verwenden zu können, ist für den +12 V-Stabilisator eine Spannungsverdopplung vorgesehen. Diese besteht aus den Komponenten C1, C2 sowie D5, D6.
Soviel zur Schaltung des Junior-Computers. Wir haben uns bemüht, nicht zu sehr ins Detail zu gehen, aber auch nicht zu oberflächlich zu sein. Beim Erarbeiten der (Mikro-)Computertechnik sollten Hardware und Software Hand in Hand gehen. Eigentlich ist diese Forderung selbstverständlich.

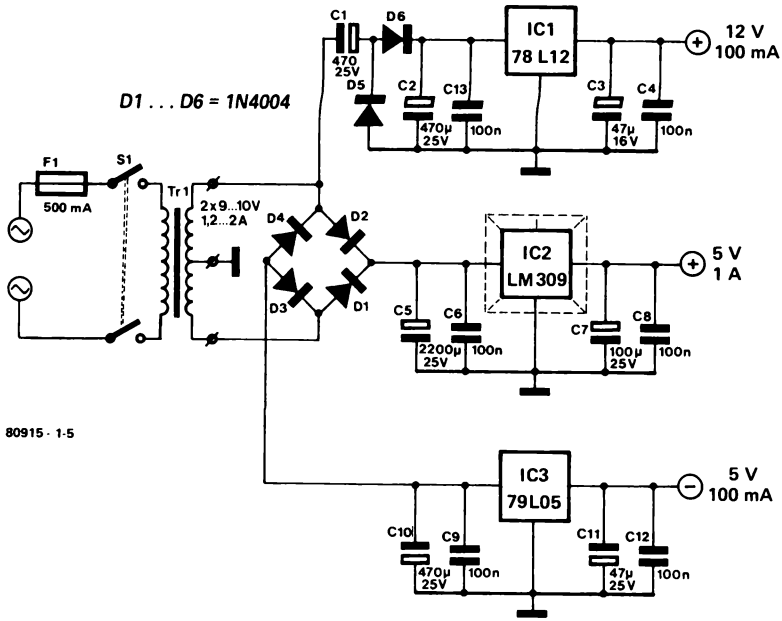


Bild 5. Das Netzteil des Junior-Computers liefert drei stabilisierte Versorgungsspannungen.

Der Bau des Junior-Computers

Jetzt sind wir beim Bau des Junior-Computers angelangt. Es ist leicht möglich, diesen Mikrocomputer selbst zu bauen. Die Aufgaben der verwendeten ICs sind bereits besprochen worden. Nun müssen wir mit diesen ICs die Basis-, die Display- und die Netzteilplatine bestücken. Es ist davon abzuraten, die doppelseitig bedruckte Basisplatine selbst herzustellen. Mehr als 600 Löcher sind zu bohren und durchzukontaktieren. Wie leicht entsteht dabei ein Fehler!

Der Junior-Computer ist ein "single board computer", daß heißt, alle Bauteile sind auf einer einzigen Platine untergebracht. Wie wir aber wissen, setzt sich der Junior-Computer aus drei Platinen zusammen. Warum also "single board computer"? Auf einer dieser drei Platinen ist das Netzteil untergebracht, und dieses zählt nicht zum Computer. Bleiben noch die Basisplatine und die Displayplatine übrig.

Die letzte ist eine kleine Zusatzplatine und fest mit der Basisplatine verbunden. Sicherlich hätten die sechs Displays auf der Basisplatine untergebracht werden können, aber dann wäre diese größer und teurer geworden. Um dieses Problem zu umgehen, haben wir die Displayplatine in einem Winkel von 45° auf die Basisplatine montiert. Das hat auch einen praktischen Grund: Ein schräggestelltes Display ist leichter abzulesen! Nun dürfte klar sein, daß der Junior-Computer ein "single board computer" ist, obwohl er aus zwei Platinen besteht.

Die Basisplatine

Wegen der Abmessungen dieses Buches mußten wir die Basisplatine des Junior-Computers verkleinert abdrucken. Wer trotz Warnung diese Platine selbst herstellen möchte, benötigt die wahren Abmessungen des Printlayouts. Dieses ist in Elektor, Mai 1980, zu finden. Bei einer selbst hergestellten Platine ist es empfehlenswert, alle Durchverbindungen, die von einer Platinenseite zur anderen führen, zu testen. Das läßt sich mit einem Ohmmeter erledigen oder besser mit einem Summer und einem Klingeltrafo oder dem Transformator des Netzteiles. Bild 14a zeigt eine solche akustische Prüfeinrichtung. Bild 14b zeigt, wie eine nicht vorhandene Durchkontaktierung durch ein Drahtstück oder den Anschlußdraht eines Bauteils ersetzt werden kann.

Auf der Oberseite der doppelseitigen Basisplatine befinden sich das Keyboard und das sechsteilige 7-Segment-Display. Auf der Unterseite hängen die CPU, der PIA, der Speicher und die übrigen Komponenten. Den Komponentenaufdruck der Oberseite zeigt Bild 6, den der Unterseite Bild 7. Der Verlauf der Leiterbahnen auf Ober- und Unterseite ist in den Bildern 8 und 9 dargestellt. Lötarbeiten sollten mit einem geeigneten Lötkolben durchgeführt werden. Dazu empfehlen wir einen Lötkolben mit einer Leistung von 20 . . . 25 W und eine Bleistiftlötspitze. Das verwendete Lötzinn sollte nicht dicker als 1 mm sein, da sonst eine genaue Dosierung unmöglich ist.

Beim Bestücken der Platine geht man am besten wie folgt vor: Die Widerstände R1 . . . R2Ø werden in die Basisplatine gesteckt und anschließend festgelötet. Dann werden die überstehenden Drähte so kurz wie möglich mit einem scharfen Seitenschneider abgeschnitten. Demjenigen Leser, der nicht täglich mit dem Widerstandscode umgeht, soll die folgende Farbcode-Tabelle helfen:

100 k: braun-schwarz-gelb-(gold)
3k3: orange-orange-rot-(gold)
4k7: gelb-violett-rot-(gold)
33Ø Ω: orange-orange-braun-(gold)
68 Ω: blau-grau-schwarz-(gold)
2k2: rot-rot-rot-(gold)
68 k: blau-grau-orange-(gold)

Dann wird die Diode D1 eingelötet. Diese Diode sollte kein beliebiger Typ sein, sondern eine 1N4148. Beim Einlöten der Diode ist auf die richtige Polung zu achten, da sonst der Taktoszillator auf dem Prozessorchip nicht anschwingt. Jetzt sind die Kondensatoren an der Reihe. Beim Montieren der Tantalelkos ist ebenfalls auf die richtige Polarität zu achten. Der Pluspol des Elkos entspricht beim Platinenaufdruck einem leeren Balken, der Minuspol einem vollen Balken. Bei manchen Tantalelkos ist der Plusanschluß nicht mit einem "+" gekennzeichnet. Statt dessen befindet sich auf diesem ein Farbpunkt. Sieht man auf den Farbpunkt und zeigen die Anschlußdrähte nach unten, dann ist das rechte Bein der Pluspol des Tantalelkos.

Nachdem die Kondensatoren, die Widerstände und die Diode auf der Rückseite der Basisplatine festgelötet sind, werden auf der Vorderseite die 23 Tasten und die beiden Schalter montiert. Dabei sind die Lötfahnen der Schalter durch isolierten Schaltdraht mit der Platine zu verbinden.

Bei der Montage der Tasten ist darauf zu achten, daß diese senkrecht auf der Platine stehen, damit eine gegenseitige Berührung ausgeschlossen ist.
Dann kommen die ICs auf die Platine. Für IC1 und IC2 sind IC-Sockel mit 40 Pins vorzusehen, für IC2 ein Sockel mit 24 Pins. Die IC-Sockel für diese integrierten Schaltungen sollten vergoldete Kontakte haben. Die übrigen ICs werden direkt in die Platine gelötet. Alterungserscheinungen der Kontakte können sich hier nicht bemerkbar machen. Das EPROM (IC2) sollte auf alle Fälle in einem IC-Sockel sitzen, da dann das Monitorprogramm durch ein anderes Programm ersetzt werden kann.
Nun kommen der 1-MHz-Quarz und die LED in der GO-Taste (S22) auf die Platine. Bei der Montage der LED ist auf die richtige Polung zu achten. Der längere Anschluß dieser Diode ist der Pluspol (das Dreieck im Platinenaufdruck). Will man Geld sparen, so braucht der 64-polige Erweiterungskonnektor noch nicht auf die Platine montiert zu werden. Er ist erst bei der Erweiterung des Junior-Computers notwendig. Die Versorgungsspannung wird dann direkt auf die Lötinseln dieses Konnektors gelegt. Die Anschlüsse der drei Versorgungsspannungen auf dem Erweiterungskonnektor sind:
+5 V: Pin 1a oder 1c
Masse = 0 V: Pin 4a oder 4c oder 32a oder 32c
−5 V: Pin 18a
+12 V: Pin 17c
Bei Verwendung eines Konnektors lassen sich diese Anschlüsse leicht wiederfinden, da auf der Unterseite die Nummern der Anschlüsse eingeprägt sind. Bis zu einem Meter darf die Länge der vier Anschlußdrähte zwischen Netzteil und Junior-Computer betragen.
Zuletzt werden der 31-polige Portkonnektor und eine Drahtbrücke auf die Platine gelötet. Die Drahtbrücke liegt zwischen den Punkten ⊥ und D.
Nachdem alle Bauteile auf der Basisplatine festgelötet sind, befestigen wir auf der Unterseite der Basisplatine mit M3-Schrauben sechs Alu-Abstandsbolzen mit Innengewinde. Diese Abstandsbolzen sollten eine Länge von 10 mm haben. Zwei Bolzen werden unter den Portkonnektor montiert. Die restlichen vier kommen in die Ecken der Basisplatine und zwischen das aufgeteilte Tastenfeld. Dadurch erhält die Platine eine hohe mechanische Stabilität, und selbst ein starker Druck auf das Tastenfeld kann die Platine nicht verbiegen.
Die Basisplatine ist jetzt elektrisch und mechanisch zusammengebaut. Bevor wir die anderen Platinen aufbauen, sollte die Basisplatine einer genauen Kontrolle unterzogen werden. Dabei ist darauf zu achten, daß alle Bauteile am richtigen Platz sitzen und die Polarität der Kondensatoren und Dioden mit dem Platinenaufdruck übereinstimmt.

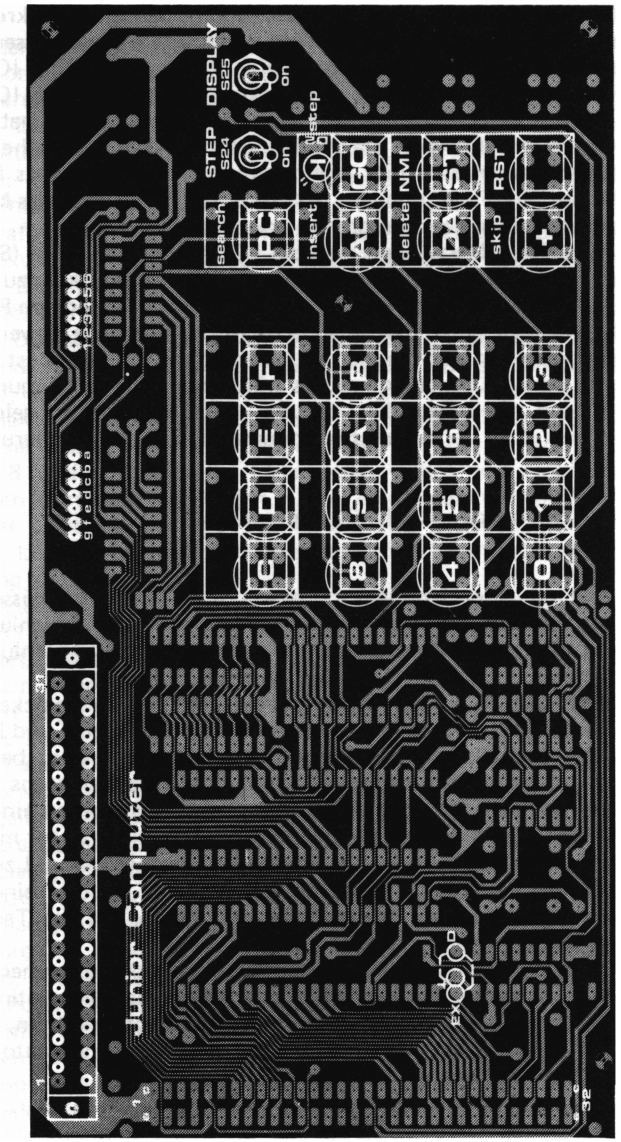


Bild 6. Oberseite der doppelseitigen Basisplatine.

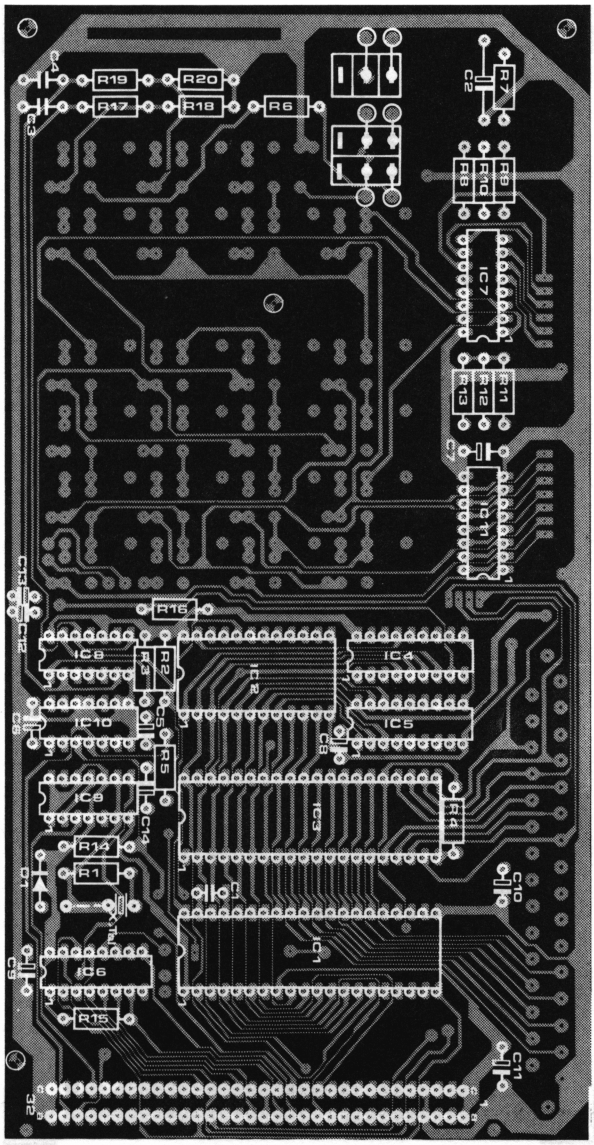


Bild 7. Unterseite der Basisplatine. Auf dieser Seite befinden sich die Widerstände, Kondensatoren, Dioden, ICs und der 1-MHz-Quarz.

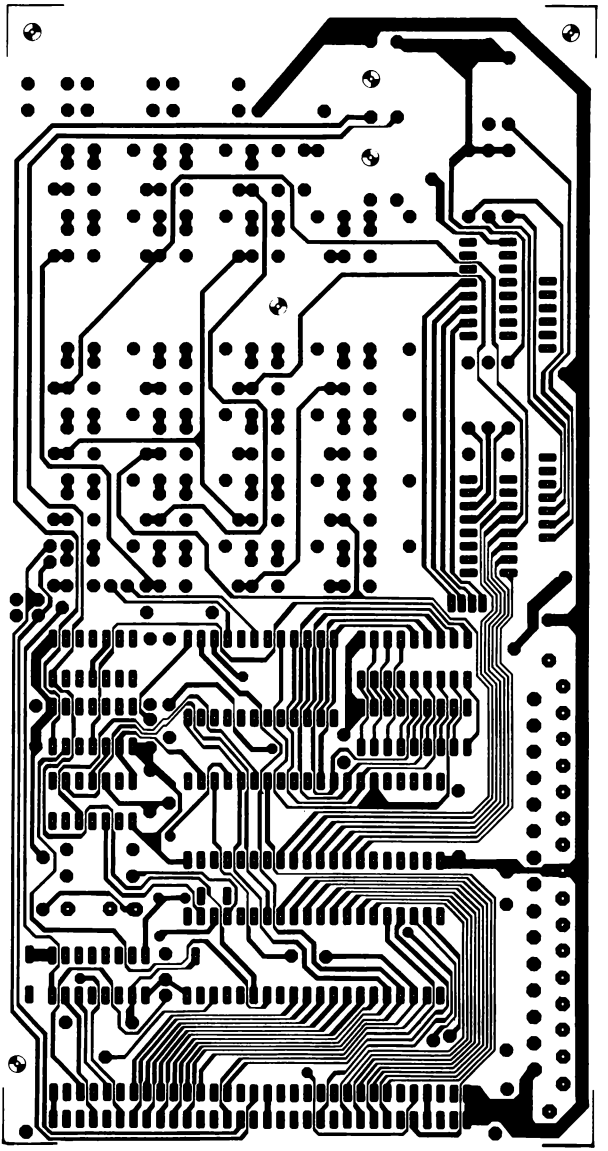


Bild 8. Verlauf der Leiterbahnen auf der Oberseite der Basisplatine (verkleinerter Maßstab!)

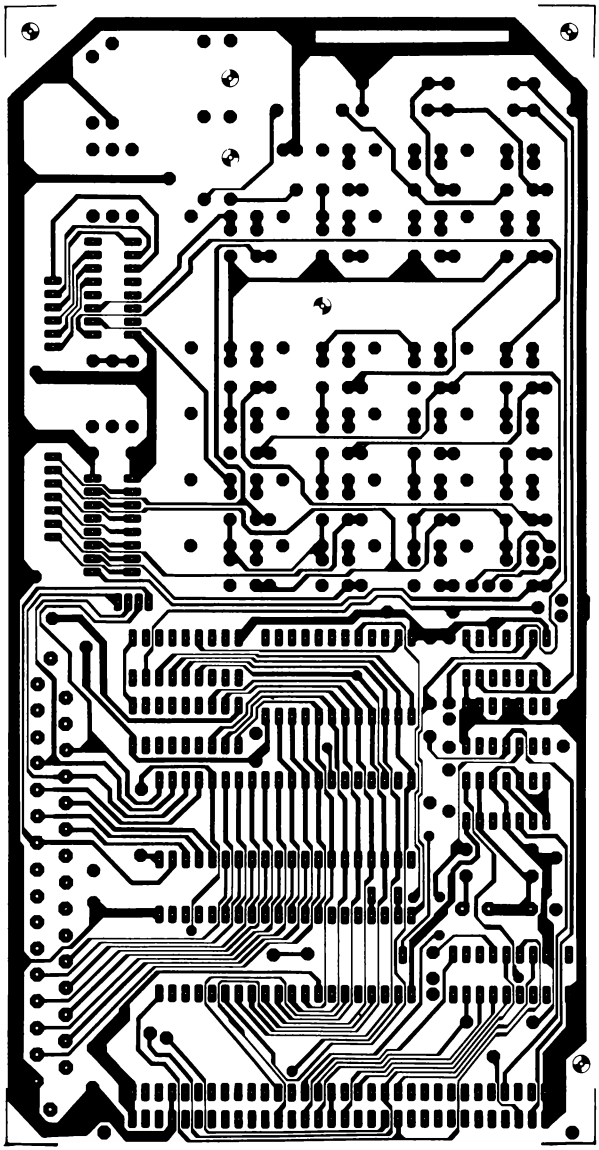


Bild 9. Verlauf der Leiterbahnen auf der Unterseite der Basisplatine.

Stückliste für die Basisplatine

Die Schaltung des Junior-Computers ist in Bild 4 dargestellt. Den Platinenentwurf und den Komponentenaufdruck zeigen die Bilder 6 . . . 8.

Widerstände:

R1 = 100 k
R2,R3,R4,R14,R15,R16 = 3k3
R5 = 4k7
R6 = 330 Ω
R7 . . . R13 = 68 Ω
R17,R19 = 2k2
R18,R20 = 68 k

Kondensatoren:

C1 = 4p7 keramisch
C2 = 47 µ/6 V Tantal
C3,C4 = 100 n MKH
C5 . . . C14 = 1 µ/35 V Tantal

Halbleiter:

IC1 = 6502 (Rockwell)
IC2 = 2708
IC3 = 6532 (Rockwell)
IC4,IC5 = 2114
IC6,IC7 = 74145
IC8 = 556
IC9 = 74LS132
IC10 = 74LS01, 7401
IC11 = ULN2003 (Sprague)
D1 = 1N4148

Diverses:

XTAL = Quarz 1 MHz
S1 . . . S21,S23 = Digitast (Shadow)
S22 = Digitast mit LED (Shadow)
S24 = doppelpoliger Kipp-(Um-)Schalter
S25 = einpoliger Kipp-(Um-)Schalter
*Federleiste 31polig C42334-A56-A63 (Siemens) DIN 41617
Messerleiste 64polig C42334-A191-A502 (Siemens) DIN 41612
* (in dieser Ausbaustufe noch nicht erforderlich!)
IC-Sockel, 24-Pins
IC-Sockel, 40-Pins

Die Displayplatine

Der Komponentenaufdruck und der Verlauf der Kupferbahnen ist in den Bildern 10 und 11 dargestellt. Da die Leiterbahnen auf der Displayplatine sehr dünn sind, müssen die Lötarbeiten mit größter Sorgfalt ausgeführt werden. Auf dieser Platine müssen nur sechs Displays und 13 Drähte montiert werden. Am besten beginnt man mit den Drähten, die die Verbindung mit der Basisplatine herstellen. Dazu eignet sich abisolierter 0,8-mm-Montagedraht, der in 20 mm lange Stücke geschnitten wird. Diese 13 Drahtstücke kommen in die vorgesehenen Löcher "a . . . g" und "1 . . . 6" und werden dann festgelötet. Nachdem diese Drähte befestigt sind, richtet man sie mit einer Justierzange aus. Dann kommen die Displays direkt auf die Platine und werden ohne Sockel festgelötet. Nach diesen Arbeiten verbinden wir die Displayplatine mit der Basisplatine des Junior-Computers und löten sie auf dieser fest. Dann justieren wir die Displayplatine so, daß sie einen Winkel von 45° zur Basisplatine einnimmt. Es ist nicht unbedingt notwendig, die Displayplatine in der beschriebenen Weise auf der Basisplatine des Junior-Computers zu montieren. Kommt zum Beispiel der Computer in ein Gehäuse, dann läßt sich die Displayplatine in beliebigem Abstand von der Basisplatine montieren. Die 13 Verbindungen zwischen den Platinen lassen sich dann durch ein mehradriges Fachkabel (flat cable) herstellen.

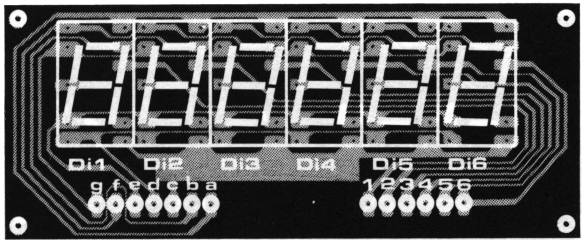


Bild 10. Komponentenaufdruck der Displayplatine.

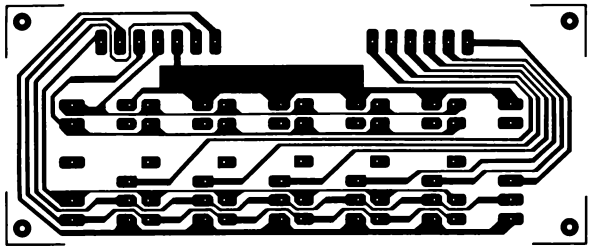


Bild 11. Verlauf der Kupferbahnen auf der Displayplatine.

Stückliste für die Displayplatine

Den Komponentenaufdruck und den Platinenentwurf zeigen die Bilder 10 und 11.

Halbleiter:

Di1,Di2,Di3,Di4,Di5, Di6 = MAN4640A, gemeinsame Katode, (Monsanto)

Die Netzteilplatine

Der Komponentenaufdruck und der Verlauf der Kupferbahnen ist in den Bildern 12 und 13 dargestellt. Beim Aufbau dieser Platine sind keinerlei Schwierigkeiten zu erwarten. Doch ist dabei auf die richtige Polarität der Dioden und Elkos zu achten. Bei der Montage des +5-V-Stabilisators machen wir von einem Fingerkühlkörper Gebrauch. Bevor IC2 festgelötet wird, muß es mit zwei M3-Schrauben auf der Platine befestigt werden. Am besten verwendet man einen Kühlkörper, der mit den Bohrungen für ein TO3-Gehäuse versehen ist.

Nachdem die Netzteilplatine bestückt ist, schließen wir an ihr den Netztransformator an. Die Primärseite des Trafos ist nur noch mit dem Netzschalter S1 und der Sicherung F1 zu verbinden. Damit sind alle Bestückungsarbeiten am Junior-Computer abgeschlossen.

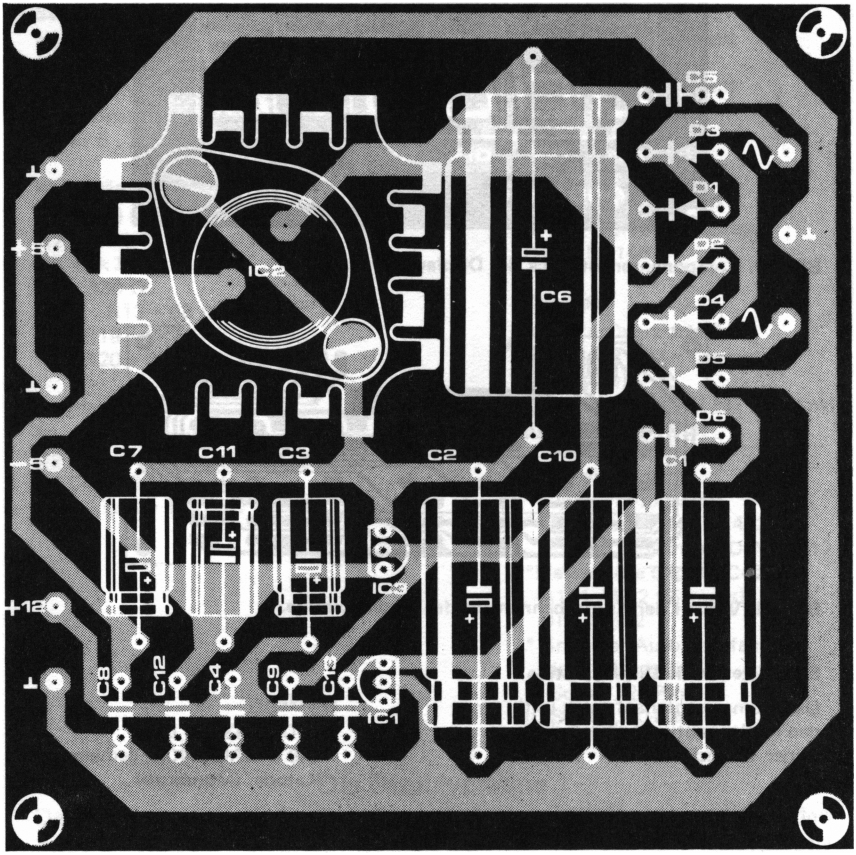


Bild 12. Komponentenaufdruck der Netzteilplatine.

Stückliste für die Netzteilplatine

Das Schaltbild des Netzteils ist in Bild 5 dargestellt. Bild 12 und 13 zeigen den Komponentenaufdruck und den Platinenentwurf.

Kondensatoren:

C1,C2,C10 = 470 µ/25 V
C3,C11 = 47 µ/25 V
C4,C5,C8,C9,C12,C13 = 100 n MKH
C6 = 2200 µ/25 V
C7 = 100 µ/25 V

Halbleiter:

IC1 = 78L12ACP (5%)
IC2 = LM309K + Fingerkühlkörper für TO-3-Gehäuse
IC3 = 79L05 ACP (5%)
D1 . . . D6 = 1N4004

Diverses:

Tr1 = Netztrafo sek. 2 x 9 . . . 10 V/1,2 . . . 2 A
S1 = Netzschalter, doppelpolig.
F1 = Sicherung 500 mA, träge, mit Halter (extern!)

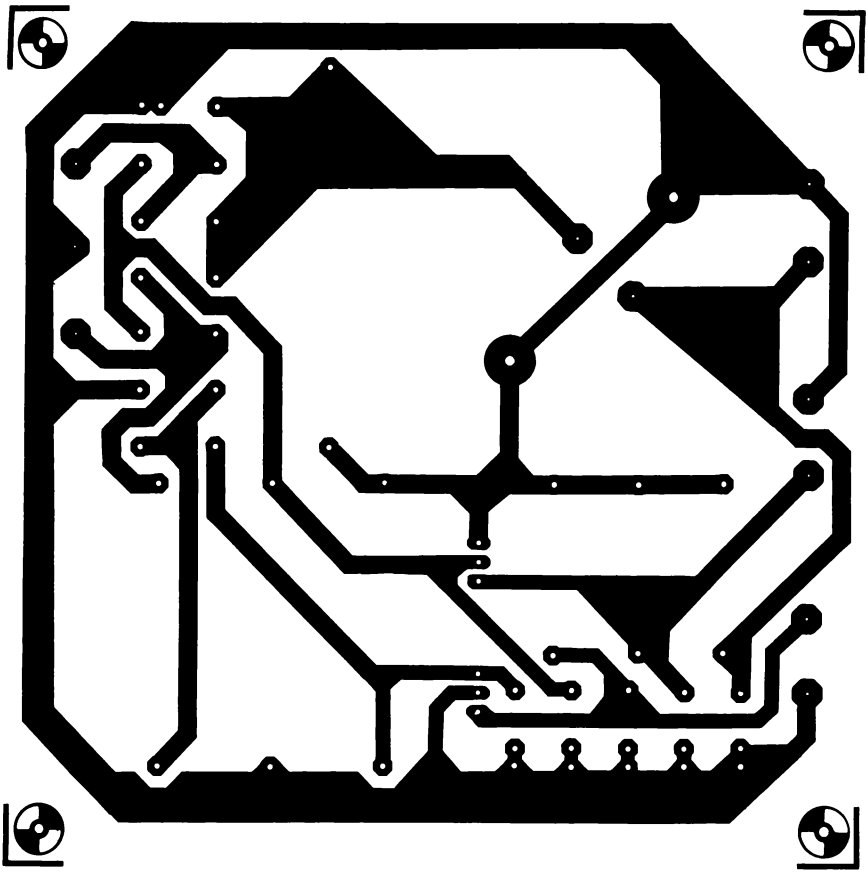
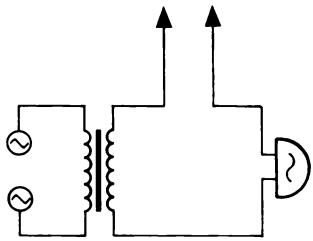


Bild 13. Verlauf der Kupferbahnen auf der Netzteilplatine.

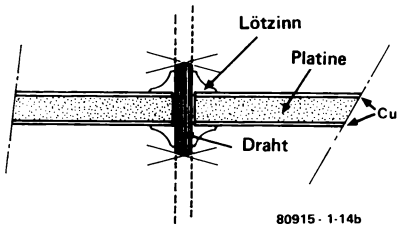
Und jetzt wird's spannend!

Jetzt nehmen wir den Junior-Computer in Betrieb. Bevor wir das Netzteil mit der Computerplatine verbinden, führen wir vorsichtshalber ein paar Messungen durch. Dazu wird das Netzteil mit S1 eingeschaltet, und mit einen Multimeter überprüfen wir die drei Versorgungsspannungen +5 V, −5 V und +12 V. Diese Spannungen müssen in einem Toleranzbereich von ± 5% liegen. Ist das nicht der Fall, dann ist das Netzteil nochmals zu untersuchen. Normalerweise kann bei dieser Art von Netzteil nichts schief gehen. Ist das Netzteil in Ordnung, dann verbinden wir es mit dem Junior-Computer. Dabei ist nochmals die Richtigkeit der vier Anschlüsse zu überprüfen. Falsch angeschlossene Versorgungsspannungen bekommen der "Gesundheit" des Junior-Computers schlecht.



80915 - 1-14a

Bild 14a. Das Testen der Durchverbindungen auf der doppelseitigen Basisplatine kann mit einem Ohmmeter erfolgen. Eine elegantere Prüfmethode ist das "akustische" Durchmessen dieser Verbindungen: Man benötigt dazu einen Trafo und einen Summer. Diese Prüfmethode sollte nur dann angewendet werden, wenn die Platine noch nicht bestückt ist.



80915 - 1-14b

Bild 14b. Hier wird gezeigt, wie man bei einer selbstgefertigten Basisplatine eine Durchverbindung mit einer Drahtbrücke oder durch den Anschlußdraht eines Bauteils herstellen kann.

Nachdem die Basisplatine mit dem *ausgeschalteten* Netzteil verbunden ist, schalten wir den "step by step mode" aus, das heißt, der Schalter S24 steht in der Position "OFF". Dann bringen wir den Displayschalter S25 in die Position "ON" und schalten das Netzteil wieder ein. Das Display bleibt dunkel. Keine Panik! Das ist ganz normal, wenn an den Junior-Computer die Versorgungsspannung angelegt wird. Erst muß die RST-Taste betätigt werden, bevor das Display aufleuchtet. Wenn das Display aufleuchtet, dann zeigt es willkürliche hexadezimale Zahlen an. Was unter einer hexadezimalen Zahl zu verstehen ist, beschreiben wir im folgenden Kapitel dieses Buchs. Bild 15 zeigt, in welcher Weise die Segmente der Displays aufleuchten können. Leuchten alle sechs Displays nach dem Loslassen der RST-Taste auf, dann funktioniert die Hardware des Junior-Computers.

Was tun, wenn das Display dunkel bleibt?

Wir sind der Meinung, daß nur in seltenen Fällen das Display dunkel bleibt. Die Platinen wurden mit der größten Sorgfalt entwickelt, und die

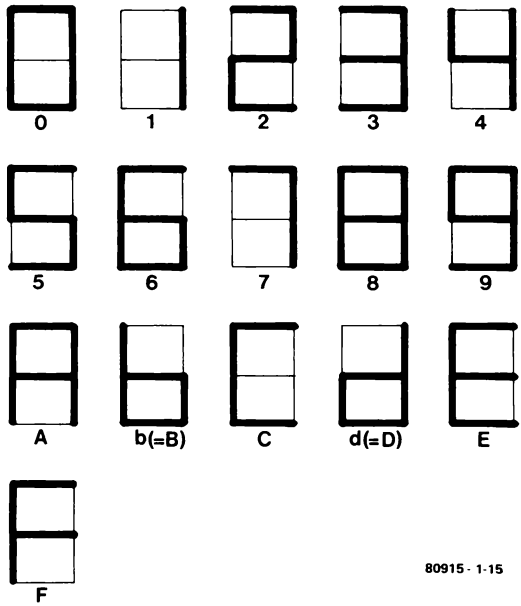


Bild 15. Nach Betätigen der RST-Taste erscheinen auf dem Display des Junior-Computers sechs willkürliche hexadezimale Zeichen. Hier sind alle Zeichen zusammengestellt, die auf dem Display aufleuchten dürfen.

Hardware hat eine Testzeit von fast zwei Jahren hinter sich. Aber dennoch kann der Junior-Computer in Einzelfällen nicht arbeiten.
Angenommen das Netzteil arbeitet und der Junior-Computer gibt trotzdem keinen "Pieps" von sich. Folgende allgemeine Ursachen können daran schuld sein:

- Beim Einlöten von Bauteilen auf der Basis- oder Displayplatine ist ein unerlaubter Kurzschluß entstanden. Wer wenig Erfahrung mit dem Lötkolben hat, sollte deshalb alle Lötstellen des Junior-Computers überprüfen.
- Schlechte Lötverbindungen sind ebenfalls ein Grund, daß der Computer nicht arbeitet. Im Zweifelsfall werden kalte Lötstellen nochmals mit Zinn nachgelötet.
- Schlechte Kontakte zwischen Konnektor-, Feder- und -Stiftleiste. Bei bereits verwendeten Konnektoren kann diese Möglichkeit leicht zutreffen. Auch zwischen IC und IC-Sockel sind schlechte Kontakte möglich. Oft wird beim Eindrücken des ICs in den Sockel ein Pin verbogen. Das kommt häufiger vor, als man denkt!
- Falsch gepolte Dioden und Elkos können ebenfalls der Grund sein, daß der Computer nicht arbeitet. Auch die Anschlüsse der drei Versorgungsspannungen müssen an den richtigen Expansion-Konnektor-Pins angelötet sein. Soviel zu den allgemeinen Fehlermöglichkeiten. Nun geben wir noch ein paar spezielle Tips, um eventuelle Fehler aufzuspüren:

– Mit einem Multimeter messen wir die Spannung zwischen Pin 13 und Pin 7 von IC8. Diese Spannung muß ungefähr 5 V betragen. Betätigt man die RST-Taste, dann muß diese Spannung ca. 0,5 V sein. Ist das nicht der Fall, dann können folgende Bauteile defekt sein: Der doppelte Timer IC8, der Pullup-Widerstand R2, oder die Taste RST.
– Ist das alles in Ordnung, dann überprüfen wir mit einem Ohmmeter, ob Pin 12 von IC6 an Masse liegt. Ist das nicht der Fall, dann ist die Drahtbrücke auf der Basisplatine falsch oder nicht eingelötet.
– Auch der Taktgenerator kann leicht überprüft werden. Mit einem Oszilloskop überprüfen wir die Spannungen an den Expansion-Konnektor-Pins 30a und 27a. Dort liegen die rechteckförmigen Signale $\Phi 1$ und $\Phi 2$. Die Frequenz muß 1 MHz ($\hat{=}$ 1 μs) sein und die Amplitude sollte einen Wert von 3 . . . 5 V haben. Mit einem Zweistrahl-Oszilloskop läßt sich leicht feststellen, daß sich diese beiden Taktsignale nicht überlappen. Sind auf dem Bildschirm keine rechteckförmigen Spannungen zu sehen, dann steht der Taktgenerator still. Als verdächtige Bauteile kommen dann C1, IC9, D1 oder der Quarz in Frage.

Sollten dennoch Schwierigkeiten bei der Hardware des Junior-Computers auftreten, dann können Sie schriftlich oder telefonisch mit uns Kontakt aufnehmen. Wir stehen Ihnen gern mit Rat und Tat zur Seite!

Ein Gehäuse für den Junior?

Eigentlich gibt es über ein Gehäuse für den Junior-Computer nicht viel zu sagen. Will man eins kaufen oder selbst bauen, so ist einem geräumigen Gehäuse der Vorzug zu geben. Das Gehäuse soll folgende Computerbausteine aufnehmen können:

– den Junior-Computer
– das Elekterminal mit ASCII Keyboard
– fünf Eurokarten, gesteckt auf eine Busplatine
– mehrere Steckverbindungen zum Anschluß von zwei Cassettenrekordern, zwei Floppy's eines Druckers usw.
– Platz für ein "dickes" Computernetzteil mit Trafo (Bei Erweiterungen findet der SC/MP-Trafo Einsatz. Seine Daten sind: 10 V/6,5 A; 15 V/1 A)

Auch bei künftigen Erweiterungen müssen das Tastenfeld und das Display des Junior-Computers von außen bedient werden können. Das heißt, es wird mit dem hexadezimalen Keyboard auf der Basisplatine, dem 6-stelligen 7-Segment-Display sowie einem ASCII-Keyboard und einem Drucker zusammengearbeitet. Wer einen Metallfoliendrucker an den Junior-Computer anschließen möchte, kann diesen ebenfalls mit ins Gehäuse einbauen. Es kann jeder beliebige Metallfoliendrucker (z.B. DATAMEGA-Drucker) verwendet werden, Hauptsache der Druckkopf hat sieben Nadeln.

Wer kein Gehäuse haben möchte, kann auf eine Möglichkeit zurückgreifen, die billig ist und sich im Elektor-Labor bewährt hat:

Der Junior-Computer wird samt Netzteil und Trafo auf einer 5 mm dicken Plexiglassscheibe montiert. Diese Scheibe hat die Abmessungen von 25 cm x 30 cm. Das Netzteil kommt in ein eigenes Plexiglasgehäuse, damit ein Kontakt mit der Netzspannung ausgeschlossen ist.

Das war die Hardware des Junior-Computers. In den folgenden Kapiteln beschäftigen wir uns ausschließlich mit dem Programmieren dieser schönen Maschine. Mit diesem Abenteuer können wir jetzt endlich beginnen!

EPROMs werden im Handel unprogrammiert vertrieben. Ein nicht-programmiertes EPROM ist für den Junior-Computer unbrauchbar. Erst wenn mit Hilfe einer Programmiermaschine (EPROM-Programmer) das Monitorprogramm des Junior-Computers in das EPROM "eingebrannt" ist, kann es in den IC-Sockel eingesetzt werden. Wenn Sie selbst einen EPROM-Programmer besitzen, dann können Sie mit Hilfe des hexadezimalen "Monitor Dump" am Ende dieses Buches das EPROM selbst programmieren. Viele Händler haben von uns ein EPROM bekommen, das den Monitor enthält. Davon können auch Sie eine Kopie anfertigen lassen. Es gibt auch noch eine andere Möglichkeit, wie Sie den Monitor in Ihr EPROM bekommen: Der Elektor-EPROM-Service! Wie funktioniert's? Schicken Sie die (E)PROMs sorgfältig verpackt (in Aluminiumfolie oder Schaumstoff) an Elektor unter Angabe des gewünschten Programms. Die Bestellnummer des entsprechenden Programms und die entstehenden Kosten (zuzüglich 2,50 DM Verpackung und Rückporto) entnehmen Sie der jeweils gültigen EPS/ESS-Liste der aktuellen Elektor-Ausgabe. Wir weisen nochmals darauf hin, daß der Junior-Computer nur dann arbeiten kann, wenn das EPROM richtig programmiert wurde.

2

Digital denken und rechnen

"Das kannst Du anhand Deiner 10 Finger abzählen". Ein Ausspruch, den sicherlich jeder schon einmal gehört oder getan hat. Das Abzählen an den 10 Fingern ist nur möglich, weil wir beim Rechnen das Dezimalsystem benutzen.
Der (Junior-)Computer muß sich auch mit Berechnungen und den entsprechenden Daten befassen. Doch fehlen zum Zählen die 10 Finger. Er muß sich mit zwei begnügen: Jedes digitale System arbeitet mit dem binären Zahlensystem. Davon handelt das folgende Kapitel.

Woran denken die meisten Menschen, wenn sie die Zahl 1984 hören? Vielleicht an den Zukunftsroman "1984" von George Orwell? Andere wiederum – zum Beispiel Mathematiker – zerlegen diese Zahl in Zehnerpotenzen ($1 \times 10^3 + 9 \times 10^2 + 8 \times 10^1 + 4 \times 10^0 = 1984$). Computer arbeiten mit Zahlen, die uns dezimal denkenden Menschen ungeläufig sind. Deshalb wollen wir in diesem Kapitel lernen, wie man mit binären und hexadezimalen Zahlen – kurz Hexalzahlen – mit denen der Junior-Computer arbeitet, umgehen muß. Dieses Kapitel ist unumgänglich, wenn man das Programmieren lernen will.
Das Dezimal- oder Zehnersystem ist sicherlich das bekannteste und am meisten benutzte Zahlensystem. Dieses Zahlensystem baut auf die Zahl 10 auf. Man sagt auch, 10 ist die Basis des Dezimalsystems. Das Dezimalsystem verwendet die Zahlensymbole 1; 2; 3; 4; 5; 6; 7; 8; 9; 0. Es werden also insgesamt zehn Symbole zur Darstellung von beliebigen Zahlen aneinandergefügt. Ein Digitalrechner kann – wie schon erwähnt – nur Information in Form von Zahlen verarbeiten. Soll ein Computer beispielsweise eine Spannung, einen Strom oder eine Frequenz überwachen, so ist das nur möglich, wenn wir einer Spannung von 5 Volt die Zahl 5, einer Spannung von 6 Volt die Zahl 6 usw. zuordnen. Eine solche Zuordnung ist technisch durchaus realisierbar, aber kompliziert und teuer. Viel einfacher ist es, wenn man sich auf zwei Zustände beschränkt und diesen dann die Zahlen

Ø und 1 zuordnet. Technisch ist das mit Schaltern sehr einfach realisierbar, indem man wie folgt vorgeht:

"Ø" = log. Ø = kein Strom, keine Spannung, Schalter offen, Logikzustand "Low" oder "L", Lampe leuchtet nicht (bei positiver Logik).

"1" = log. 1 = Strom bzw. Spannung eingeschaltet, Schalter geschlossen, Logikzustand "High" oder "H", Lampe leuchtet.

Moderne Digitalrechner sowie der Junior-Computer arbeiten ausnahmslos nach diesem Prinzip. Ein Rechner bedient sich also eines Zahlensystemes, das mit nur zwei Zahlen auskommt: Ø und 1, dem binären Zahlensystem. Für eine binäre Zahl gelten selbstverständlich dieselben Rechenregeln wie für eine dezimale Zahl:

$$1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 .$$

Der einzige Unterschied ist, daß das binäre Zahlensystem die Basis zwei, dagegen das dezimale die Basis zehn hat. Wird beim Zählen im dezimalen Zahlensystem der Zahlenvorrat einer Stelle überschritten (Ergebnis größer als 9) so entsteht ein Übertrag auf die nächste Stelle. Beim Binärsystem entsteht bereits ein Übertrag, wenn die Zahl 1 überschritten wird. Bevor wir die Zahlentheorie eines Mikrocomputers weiter erörtern, sollen noch die Begriffe Bit, Byte und digitale Codes erwähnt werden.

Bit, Byte und digitale Codes

Bit ist die Abkürzung von "binary digit", was soviel wie Binärziffer bedeutet. Die Grundeinheit einer Information in der Digitaltechnik ist das Bit. Ein bekanntes Speicherelement, das D-Flipflop, kann zum Beispiel ein Bit speichern. Eine Bitfolge wie ØlØ1 kann als Dezimalzahl interpretiert werden: $ØlØ1 = Ø \times 2^3 + 1 \times 2^2 + Ø \times 2^1 + 1 \times 2^Ø = Ø + 4 + Ø + 1 = 5_{1Ø}$. Um nun die verschiedenen Zahlen in den verschiedenen Zahlensystemen zu unterscheiden, wird die Basis des Zahlensystemes —wie in diesem Beispiel— als Index geschrieben. Also $ØlØ1_2 = 5_{10}$ (sprich: Null-eins-null-eins im Binärsystem entspricht 5 im Zehnersystem). Der Bitfolge ØlØØØlØ1 entspricht der Buchstabe "E" im ASCII-Code. Mit diesem Code werden wir uns noch später auseinandersetzen. Faßt man nun mehrere Bits zu einer Einheit zusammen, erhält man ein Wort, das mehrere Bit lang ist. Das Binärwort $ØlØ1_2$ ist vier Bit lang, sowie das Wort ELEKTOR sieben Buchstaben "lang" ist. Ein Binärwort, das sich aus acht Bit zusammensetzt oder acht Bit lang ist, heißt Byte.

Im Teil 1 haben wir von der Speicherorganisation gesprochen. Das RAM des Junior-Computers kann Worte speichern, die vier Bit lang sind. Um acht Bit, also ein Byte zu speichern, benötigten wir deshalb zwei solcher RAM-ICs. Im EPROM dagegen sind Worte mit einer Länge von acht Bit gespeichert. Jedes Wort, das aus diesem Speicherelement ausgelesen wird, entspricht einem Byte. Ein digitaler Code setzt sich ebenfalls wie das Wort aus einzelnen Bits zusammen, also aus einer Folge von Nullen und Einsen. Digitale Codes stellen die Daten so dar, daß ein Computer sie versteht. Will der Programmierer in einen Computer die Zahl 9_{10} eingeben, muß er diese Zahl erst so codieren, daß sie der Computer verstehen kann.

Es gibt verschiedene Arten von digitalen Codes:

– Codes mit denen die dezimale Ziffern 0 . . . 9 binär verschlüsselt werden, zum Beispiel BCD-Code, Gray-Code.

– Codes, die in Digitalschaltungen Verwendung finden, um zum Beispiel vorwärts oder rückwärts zu zählen, falls ein bestimmtes Ereignis eintrifft.
– Codes, mit denen man Dezimalziffern, Buchstaben oder Operationen in Binärform darstellen kann: ASCII-Code, Baudot-Code.
– Der Code, aus dem sich der Befehlssatz des Mikroprozessors 6502 zusammensetzt. Die Abkürzungen der erwähnten Codes haben folgende Bedeutung:

BCD = Binär codierter Dezimalcode = Binary Coded Decimal
ASCII = Amerikanischer Standard Code für den Austausch von Informationen = American Standard Code for Information Intercharge.

Binäre, dezimale und hexadezimale Zahlen.

Wenn wir mit Zahlen umgehen, so sind wir gewöhnt, im Dezimalsystem zu rechnen. Die einzelnen Ziffern einer Zahl haben positionsabhängige Gewichte. Deshalb kann ein und dieselbe Zahl in den verschiedenen Zahlensystemen verschieden groß sein. Nehmen wir das gewohnte Zehnersystem, so bedeutet die Zahl $1984_{10} = 1 \times 10^3 + 9 \times 10^2 + 8 \times 10^1 + 4 \times 10^0$. Das haben wir anfangs schon erwähnt. Legt man aber der Zahl 1984 ein Zahlensystem mit der Basis 16 zugrunde, so erhält man:
$1984_{16} = 1 \times 16^3 + 9 \times 16^2 + 8 \times 16^1 + 4 \times 16^0 = 6532_{10}$. Ein Zahlensystem mit der Basis 16 ist das hexadezimale Zahlensystem oder kurz Hexalsystem. Nun müssen wir lernen, wie man eine bestimmte Zahl in ein anderes Zahlensystem transformiert. Dazu stellen wir nochmals die Zahlensysteme mit ihren Symbolen für die Ziffern zusammen:
– Das Binärsystem: 0; 1.
– Das Dezimalsystem: 0; 1; 2; 3; 4; 5, 6; 7; 8; 9.
– Das Hexalsystem: 0; 1; 2; 3; 4; 5, 6; 7; 8; 9; A; B; C; D; E; F.

Aus dieser Zusammenstellung ist ersichtlich, daß das hexadezimale System die Zahlensymbole 0 . . . 9 wie das dezimale System verwendet. Um für Zahlen, die größer als 9 sind, nur eine Ziffer verwenden zu müssen, werden die Zahlensymbole A . . . F eingeführt. Bild 1 zeigt anschaulich, wie sich eine binäre, eine dezimale und eine hexadezimale Zahlenreihe aufbaut.
Die Rechenregeln, wie wir sie vom Zehnersystem her gewohnt sind, lassen sich auf alle Zahlensysteme übertragen. In diesem Abschnitt werden die vier Grundrechenarten – Addition, Subtraktion, Multiplikation und Division – in binären Zahlensystem beschrieben.

Die binäre Addition

Bei der Addition im Zehnersystem entsteht bekanntlich immer dann ein Übertrag auf die nächste Stelle, wenn die Zahl 10 überschritten wird. Das folgende Additionsbeispiel soll diesen Sachverhalt ins Gedächtnis zurückrufen.

129_{10}	1. Zahl
$+243_{10}$	2. Zahl
1	Übertrag
372	

Werden nun zwei binäre Zahlen addiert, so entsteht immer dann ein Übertrag von einer Stelle auf die andere, wenn die Summe einer Spalte

binär	dezimal	hexa-dezimal	nibble
0	0	0	0000
1	1	1	0001
10	2	2	0010
11	3	3	0011
100	4	4	0100
101	5	5	0101
110	6	6	0110
111	7	7	0111
1000	8	8	1000
1001	9	9	1001
1010	10	A	1010
1011	11	B	1011
1100	12	C	1100
1101	13	D	1101
1110	14	E	1110
1111	15	F	1111
10000	16	10	
10001	17	11	
.	.	.	
.	.	.	
.	.	.	
.	.	.	

Bild 1. In dieser Zahlentafel sind binäre, dezimale und hexadezimale Zahlen gegenübergestellt. Die hexadezimale Schreibweise ist eine Stenoschrift für eine Binärzahl.

größer oder gleich 2 ist. An Hand eines Beispieles soll das demonstriert werden:

```
   101101  1. Binärzahl
+    10101  2. Binärzahl
  1111  1   Übertrag
-------------------
  1000010
```

In der Computertechnik bezeichnet man diesen Übertrag als Carry, was gleichbedeutend mit dem deutschen Wort Übertrag ist.

Die binäre Subtraktion

Bei der Subtraktion wird der Minuend (= die Zahl, von der eine andere abgezogen wird) um die Größe des Subtrahenden (= die Zahl, die abgezogen wird) vermindert. Das Ergebnis ist die Differenz. Im Binärsystem läuft die Subtraktion genauso ab, wie im vertrauten Dezimalsystem. Subtrahieren wir anfangs zwei dezimale Zahlen voneinander:

```
  1984 Minuend
-  199 Subtrahend
    11 Entlehnung
-----------------
  1785 Differenz
```

Um bei der niedrigsten Stelle die Zahl 9 von der Zahl 4 abziehen zu können, wurde eine Entlehnung von der nächsthöheren Stelle gemacht. Bei binären Zahlen läuft die Subtraktion genauso ab:

```
  11001001  Minuend
–  1001000  Subtrahend
           Entlehnung
----------
  10000001  Differenz
```

In diesem Beispiel ist der Minuend pro Stelle größer oder gleich dem Subtrahenden. Eine Entlehnung von der nächsthöheren Stelle war deshalb nicht erforderlich. Im folgenden Beispiel trifft das nicht zu und eine Entlehnung muß stattfinden.

```
  11000001  Minuend
–  1111110  Subtrahend
  111111    Entlehnung
----------
  01000011  Differenz
```

Die binäre Multiplikation

Auch die binäre Multiplikation wird nach denselben Regeln durchgeführt, wie die dezimale Multiplikation. Dabei werden Produkte mit den einzelnen Stellen des Multiplikators (= die Zahl, mit der eine andere multipliziert wird) gebildet und anschließend stellenrichtig addiert. Gegenüber der dezimalen Multiplikation bringt die binäre Vereinfachungen mit sich. Da die Stellen des Multiplikators nur die Zahlenwerte Null und Eins annehmen können, muß der Multiplikand (= die Zahl, die mit dem Multiplikator multipliziert wird) nur mit Null und Eins multipliziert werden. Bei der binären Multiplikation kommen als Teiloperationen nur die Addition und Verschiebung vor. Das ist wichtig zu wissen, wenn später Programme entwickelt werden. Bevor wir eine binäre Multiplikation an Hand eines Beispieles durchführen, rufen wir uns die dezimale Multiplikation ins Gedächtnis zurück.

```
233 x 147
----------
  233
  932
+ 1631
  11      Übertrag
----------
 34251
```

Und nun die binäre Multiplikation:

```
1011 x 1010
-----------
  1011
  0000
  1011
+ 0000
  1         Übertrag
-----------
 1101110
```

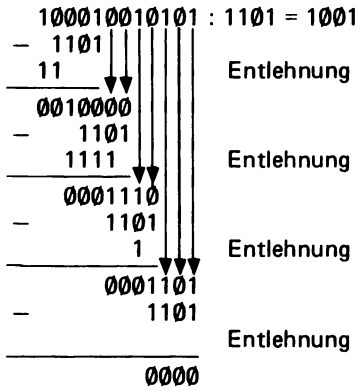
Die binäre Division

Die binäre Division bedient sich desselben Algorithmus wie die dezimale Division. Die einzelnen Operationen der binären Division sind Verschiebung und Subtraktion. Sehen wir uns anfangs den Ablauf einer dezimalen Division an:

$2091_{10} : 17_{10} = 123_{10}$

```
 - 17
 ----
  39
 - 34
 ----
   51
 -  51
 -----
   00
```

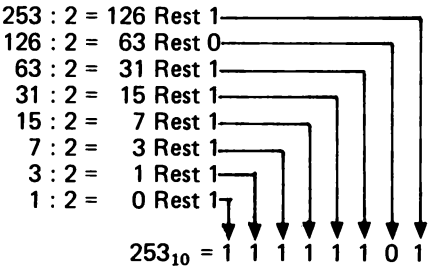
Da bei der binären Division die Stellen des Dividenden (= die Zahl, die geteilt wird) und des Divisors (= die Zahl, durch die geteilt wird) nur Null oder Eins sein können, ergibt sich wieder eine Vereinfachung.



Umwandlung einer dezimalen Zahl in eine binäre Zahl

Häufig muß eine dezimale Zahl in eine binäre umgewandelt werden. Die Dezimalzahl wird dabei in Zweierpotenzen gegliedert, das heißt solange wie nur möglich durch zwei dividiert:

$253_{10} = ?_2$



Umwandlung einer Hexalzahl in eine binäre Zahl

Die Umwandlung einer Hexalzahl in eine binäre Zahl geht in der umgekehrten Reihenfolge vor sich. Dazu sieht man in Bild 1 nach, welche Binärzahlen den hexadezimalen entsprechen.

$$12\ BF = ?_2$$

Umwandlung:

1 2 B F

lt. Bild 1: ØØØ1 ØØ1Ø 1Ø11 1111

Zieht man die Vierergruppen wieder zu einer Zahl zusammen so erhält man:

$$12\ BF = ØØØ1ØØ1Ø1Ø111111_2$$
$$= 1ØØ1Ø1Ø111111_2$$

Umwandlung einer Hexalzahl in eine dezimale Zahl

Bei dieser Umwandlung wird die hexadezimale Zahl als Summe von 16-ner Potenzen im Zehnersystem angeschrieben.
Betrachten wir z.B. die Hexalzahl ABBA:

$$ABBA = A \times (16_{10})^3 + B \times (16_{10})^2 + B \times (16_{10})^1 + A \times (16_{10})^0$$

Die hexadezimalen Zahlensymbole A und B werden lt. Bild 1 in Dezimalzahlen umgewandelt. $A = 10_{10}$ und $B = 11_{10}$. Nun lassen wir die Indizes wegen der Übersichtlichkeit weg und können neu ansetzen:

$$ABBA = A \times 16^3 + B \times 16^2 + B \times 16^1 + A \times 16^0$$
$$= 10 \times 16^3 + 11 \times 16^2 + 11 \times 16^1 + 10 \times 16^0$$
$$= 43962_{10}$$

Umwandlung einer binären Zahl in eine Hexalzahl

Die Zahl $1ØØ1ØØ1Ø_2$ soll in eine Hexalzahl umgewandelt werden. Das Problem läßt sich wie folgt darstellen:

$$1ØØ1ØØ1Ø_2 = ?_{16}$$

Hexalzahlen werden sowohl durch eine "16" im Index oder durch ein Dollarzeichen gekennzeichnet. Der erste Schritt, das Problem zu lösen, besteht darin, daß die binäre Zahl in Vierergruppen aufgeteilt wird; von rechts beginnend:

$$1ØØ1ØØ1Ø = 1ØØ1\ ØØ1Ø = 92_{16} = \$\ 92$$

9 2

Dann sehen wir in Bild 1 nach, welchen hexadezimalen Wert die Vierergruppe hat. 1ØØ1 hat den hexadezimalen Wert $ 9 und ØØ1Ø den Wert $ 2. Diese Ziffern – zu einer Zahl zusammengesetzt – ergeben die hexadezimale Zahl $ 92 oder 92_{16}.
Nun soll die binäre Zahl $1111Ø1Ø1Ø_2$ in eine hexadezimale Zahl umgewandelt werden. Zuerst trennen wir wieder in Vierergruppen von rechts nach links:

ØØØ1 111Ø 1Ø1Ø = $ 1EA

1 E A

Bei der hochwertigen Vierergruppe werden die fehlenden Binärziffern der Vierergruppe durch Nullen ergänzt (unterstrichene Nullen).

Zweierkomplement Arithmetik (Two's Complement Arithmetic)

Bis jetzt wurden nur Zahlen mit einem positiven Vorzeichen betrachtet. Außerdem wurde nicht berücksichtigt, daß jeder Rechner eine bestimmte Wortlänge hat. Der Junior-Computer hat, wie wir wissen, eine Wortlänge von 8 Bit oder 1 Byte. Er kann also Zahlen von $00000000_2 \ldots 11111111_2$ oder $00 \ldots FF$ oder $0_{10} \ldots 255_{10}$ verarbeiten. Wenn dieser Zahlenvorrat nicht ausreicht, dann kann man durch Aneinanderreihen von mehreren Bytes Zahlen beliebiger Wortlänge darstellen. Im Programm muß dann allerdings dieser Sachverhalt berücksichtigt werden.

Positive Binärzahlen lassen sich auf einem Zahlenstrahl darstellen. Bild 2 zeigt einen Zahlenstrahl, auf dem ein Teil der Binärzahlen von 00000000 ... 11111111 aufgetragen ist. Der Übersichtlichkeit wegen sind auch noch die äquivalenten dezimalen Zahlen eingezeichnet. Jeder positiven Binärzahl entspricht ein Punkt auf dieser Geraden.
Negative Binärzahlen lassen sich wie negative Dezimalzahlen durch ein Minuszeichen kennzeichnen. Im Hinblick auf den Rechner ist diese Art der Zahlendarstellung schwierig und vom Schaltungsaufwand unökonomisch. Beim Digitalrechner, wie dem Junior-Computer, werden negative Zahlen durch das Zweierkomplement (Two's Complement) dargestellt.

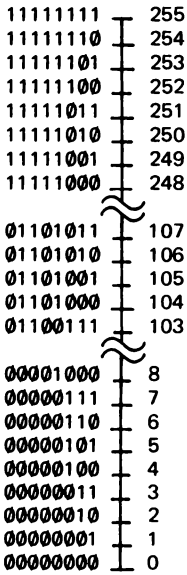


Bild 2. Die CPU 6502 ist ein 8-Bit Prozessor. Mit acht Bits lassen sich 256 Zahlen auf einem Zahlenstrahl darstellen.

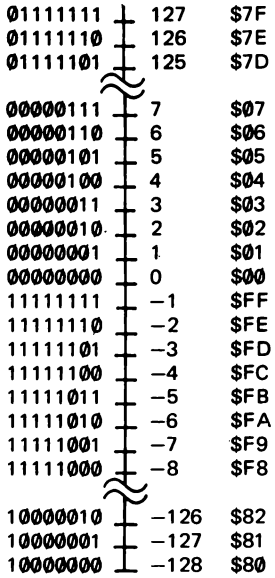


Bild 3. Zahlen mit einer Länge von acht Bits (oder einer beliebigen Wortlänge) lassen sich mit der Zweierkomplement-Notierung anschreiben. Dabei ist das hochwertigste Bit das Vorzeichenbit. Ist das höchstwertige Bit Ø, dann handelt es sich um eine positive Zahl, ist es dagegen 1, dann ist die Binärzahl negativ. Die Binärzahlen sind mit den korrespondierenden Dezimal- und Hexadezimalzahlen auf einem Zahlenstrahl dargestellt. Das $-Zeichen kennzeichnet eine hexadezimale Zahl.

Diese Darstellungsart einer negativen Binärzahl (Bild 3) wollen wir nun näher betrachten. Was ist ein Komplement? Darunter versteht man die Ergänzung zur größten gleichstelligen Zahl. Das Komplement der Dezimalzahl 1984 läßt sich wie folgt berechnen:

$A = 1984_{10}$
$\overline{A}$ = ? Komplement
$\overline{A} = 9999 - 1984 = 8015$

Das Komplement der Zahl 1984_{10} ist also 8015_{10}. Dasselbe wenden wir nun auf Binärzahlen an:

$B = 01110101$
$\overline{B}$ = ?

```
   11111111   größte gleichstellige Zahl
 − 01110101   Binärzahl "B"
   ---------- Übertrag
 ------------------------------
   10001010   Einerkomplement "B̄"
```

Sieht man sich das Ergebnis der Subtraktion an, so bemerkt man, daß das Komplement eine bitweise Invertierung der Binärzahl ist. Bei der Subtraktion der Binärzahl zur größten gleichstelligen Zahl erfolgte immer eine

Ergänzung zu 1. Deshalb wird "$\overline{B}$" auch als Einerkomplement der Binärzahl "B" bezeichnet. Addiert man eine Binärzahl zu ihrem Einerkomplement, so hat das Ergebnis immer die Form 11 . . . 11. Um das zu beweisen nehmen wir wieder das vorige Zahlenbeispiel:

$$\begin{array}{rl} & 01110101 \quad \text{"B"} \\ + & 10001010 \quad \text{"}\overline{B}\text{"} \\ \hline & 11111111 \quad \text{"B + }\overline{B}\text{"} \\ + & \qquad 1 \\ \hline & 100000000 \quad \text{"B + }\overline{B}\text{ + 1"} \end{array}$$

Das Ergebnis der ersten Addition "B + $\overline{B}$" hat also tatsächlich die Form 11 . . . 11. Addiert man zu diesem Ergebnis nochmals die Zahl 1, so erhält man eine Binärzahl in der Form 100000000. Abgesehen von dem Übertrag in die neunte Stelle der Binärzahl ist das Ergebnis Null.
Für eine 8 bit breite Binärzahl — wie in diesem Beispiel — kann also angeschrieben werden:

$$B + \overline{B} + 1 = 0 \text{ oder}$$
$$\overline{B} + 1 = -B$$

Somit ist $\overline{B}$ + 1 eine Darstellungsweise für die negative Binärzahl −B, oder anders ausgedrückt: Addiert man zu einer beliebigen Binärzahl B ihr Zweierkomplement $\overline{B}$ + 1, so ist das Ergebnis immer Null. Folgendes Beispiel wird diesen Sachverhalt demonstrieren:

01110101	Binärzahl "B"
10001010	Einerkomplement "$\overline{B}$"
10001011	Zweierkomplement "$\overline{B}$" + 1" = −B

Wird nun −B zu +B addiert, so erhält man:

$$\begin{array}{rll} & 01110101 & +B \\ + & 10001011 & -B \\ \hline & 100000000 & +B + (-B) \end{array}$$

wobei der Übertrag in die neunte Stelle nicht berücksichtigt werden soll. Wenn nun das Zweierkomplement eine Darstellungsweise von negativen Binärzahlen ist, so muß sich auch eine Subtraktion von Binärzahlen auf eine Addition zurückführen lassen. Bekanntlich läßt sich die Differenz $121_{10} - 68_{10} = 53_{10}$ auch als Summe anschreiben: $121_{10} + (-68_{10}) = 53_{10}$. Dasselbe übertragen wir nun auf die äquivalenten Binärzahlen:

$$\begin{array}{rll} 121_{10} & = 01111001_2 & \\ 68_{10} & = 01000100_2 & \\ -68_{10} & = 10111011_2 & \text{Einerkomplement} \\ & + \qquad 1_2 & \\ \hline & 10111100_2 & \text{Zweierkomplement} \end{array}$$

$$\begin{array}{rr} 121_{10} & 01111001_2 \\ -\ 68_{10} & +\ 10111100_2 \\ \hline 53_{10} & 100110101_2 \end{array}$$

Abgesehen von dem Übertrag in die 9. Stelle bei der binären Addition lautet das richtige Ergebnis:

$$53_{10} = 00110101_2.$$

Ein weiteres Beispiel soll dazu beitragen, diese "graue" Zahlentheorie verständlich zu machen. Zwei Binärzahlen unterschiedlicher Länge sollen voneinander abgezogen werden: 01110111 – 0111 = ?
Zunächst beide Zahlen auf gleiche Länge bringen, indem die höherwertigen Stellen durch Nullen ergänzt werden. Dann wird das Zweierkomplement der mit dem Minuszeichen versehenen Binärzahl gebildet.

$$\begin{array}{rll} & 00000111_2 & \text{Binärzahl} \\ & 11111000_2 & \text{Einerkomplement} \\ + & \qquad 1_2 & \\ \hline & 11111001_2 & \text{Zweierkomplement} \end{array}$$

Jetzt addieren wir das Zweierkomplement der mit dem Minuszeichen versehenen Binärzahl zur ersten positiven Binärzahl:

$$\begin{array}{rl} & 01110111_2 \\ + & 11111001_2 \\ \hline & 101110000_2 \end{array}$$

Der Übertrag auf die 9. Stelle bleibt wieder unberücksichtigt und wir erhalten als richtiges Ergebnis 01110000_2.
Negative Binärzahlen lassen sich mit dem Zweierkomplement auch auf dem Zahlenstrahl darstellen wie in Bild 3. Die Tabelle in Bild 4 gibt darüber Aufschluß, wie negative Binärzahlen negativen Dezimal- oder Hexalzahlen zugeordnet sind. Damit ist das Kapitel der "trockenen" Zahlentheorie abgeschlossen, und wir können uns nun dem Programmieren zuwenden.

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

⊕

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

⊖

Bild 4. Der Zahlenstrahl von Bild 3 ist hier in Tabellenform angegeben. Mit dieser Tabelle lassen sich dezimale Zahlen leicht in hexadezimale Zahlen umrechnen. Die Tabelle ist in zwei Teile aufgeteilt: Der obere Teil gilt für positive und der untere für negative Zahlen in Zweierkomplement-Darstellung.

Beispiel: 59_{10} = $3B
-92_{10} = $A4
$27 = 39_{10}
$F8 = -8_{10}

Da beim Programmieren das binäre und hexadezimale Rechnen von Bedeutung ist, führen Sie vorher noch folgende Übungen durch:

1. *Wie ist die hexadezimale Schreibweise von %01011111?*
2. *Wie ist die hexadezimale Schreibweise von 11111?*
3. *Wie ist die hexadezimale Schreibweise von %10100111?*
4. *Wie ist die hexadezimale Schreibweise von %110101010?*
5. *Wie ist die hexadezimale Schreibweise von %10?*
6. *Wie ist die hexadezimale Schreibweise von %001011?*
7. *Welche Binärzahl entspricht der Hexalzahl 132?*
8. *Welche Binärzahl entspricht der Hexalzahl A014?*
9. *Welche Binärzahl entspricht der Hexalzahl 0356?*
10. *Welche Binärzahl entspricht der Hexalzahl A561?*
11. *Welche Binärzahl entspricht der Hexalzahl ABBA?*
12. *Wieviel ist %01001111 + 11000111?*
13. *Wieviel ist 1110011 + 11111111?*
14. *Wieviel ist 11111111 + 1?*
15. *Wieviel ist 11110000 + 1111?*
16. *Wieviel ist %10101010 + %10101?*
17. *Wieviel ist %01110100 – 1101?*
18. *Wieviel ist 11110000 – 1111?*
19. *Wieviel ist %10111000 – 10000001?*
20. *Wieviel ist %10101111 – %10101111?*
21. *Wieviel ist 100 – 11111011?*
22. *Wieviel ist 11110001 x %01111?*
23. *Wieviel ist %101 x 11111111?*
24. *Wieviel ist %1010 x %1010?*
25. *Wieviel ist 11 x 11111111?*
26. *Wieviel ist 4 x ABBA?*
27. *Wieviel ist %10000000101 : 111?*
28. *Wieviel ist %110100000000 : 1101?*
29. *Wieviel ist %1001101111011001 0 : 1001110?*
30. *Wieviel ist $B9A0 : $0B?*

Lösungen

1. *2F*
2. *1F*
3. *A7*
4. *1AA*
5. *2*
6. *0B*
7. *%000100110010*
8. *%1010000000010100*
9. *%000000110101011 0*
10. *%1010010101100001*
11. *%1010101110111010*
12. *%100010110*
13. *%101110010*
14. *%100000000*
15. *11111111*

16. 11111111
17. 01100111
18. 11100001
19. 00110111
20. 0
21. 100001001 (–247 im 9 Bit Zweierkomplement)
22. 111000011111
23. 100111111011
24. 1100100
25. 101111101
26. 1010101110111001000 = $2AEE8
27. 100100011
28. 100000000
29. 1111111111
30. 100001110000 = S10E0

Vorbereitung auf das folgende Kapitel

Beim Entwurf eines Programms bedient sich der Programmierer gewöhnlich einer Flow Chart (Flußdiagramm). Diese ist das "Schaltbild" eines Programms. Bild 5 zeigt die wichtigsten Symbole eines Flußdiagramms.
Die "Beginn"- und "Ende"-Marken eines Programms bezeichnet man auch als Label. Sie können jedoch auch an jeder beliebigen Programmstelle als Programm-Überschriften auftreten.

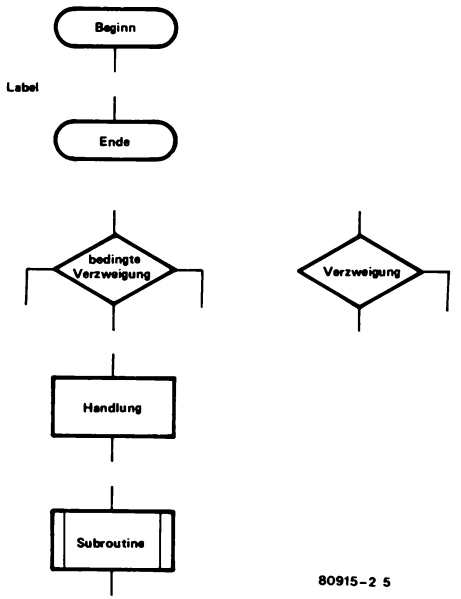


Bild 5. Symbole, die in Flußdiagrammen verwendet werden. Sie sind die Bauteile einer Flow Chart.

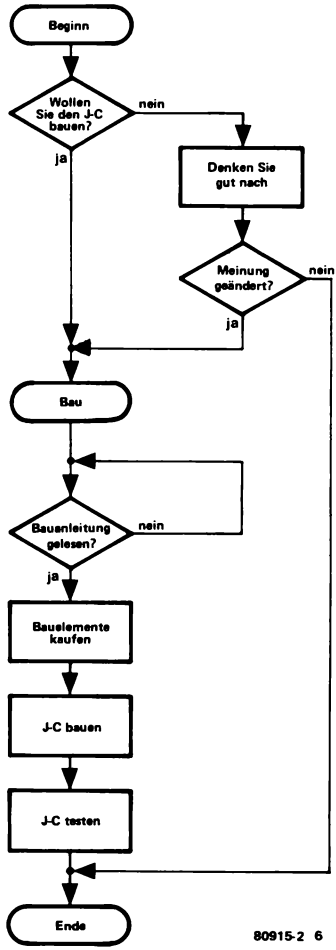


Bild 6. Das globale oder grobe Flußdiagramm für den Bau des Junior-Computers.

Dann gibt es noch die Rauten, die die Symbole für Programmverzweigungen sind. In Abhängigkeit einer oder mehrerer Bedingungen verzweigt von ihnen aus das Programm. Handlungen, die das Programm ausführen soll, stehen in Blöcken. Subroutinen stehen ebenfalls in Blöcken, die an beiden Seiten mit zwei senkrechten Strichen versehen sind.
Mit diesen wenigen Symbolen kann bereits ein Flußdiagramm gezeichnet werden. Bild 6 zeigt das globale oder grobe Flußdiagramm vom Bau des Junior-Computers. Dieses Basis-Flußdiagramm ist der Ausgangspunkt eines jeden Programmes. Die detaillierte Flow Chart für den Bau des

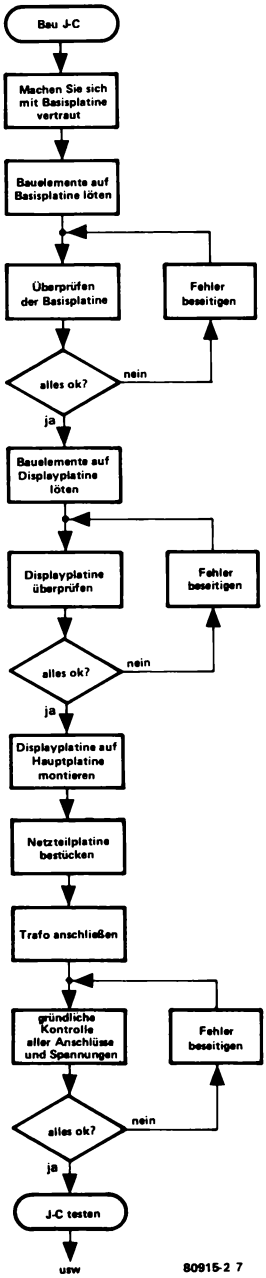


Bild 7. Das verfeinerte Flußdiagramm für den Bau des Junior-Computers. Es ist von der globalen Flow Chart in Bild 6 abgeleitet.

Junior-Computer zeigt Bild 7. Sogar dieses Programm läßt sich noch verfeinern: Bild 8 zeigt, wie ein einzelnes Bauteil (C12) auf die Platine montiert und festgelötet wird. Das Verfahren, von einer groben Flow Chart zu einer detaillierten Flow Chart zu kommen, werden wir in Kapitel 3 stets anwenden.

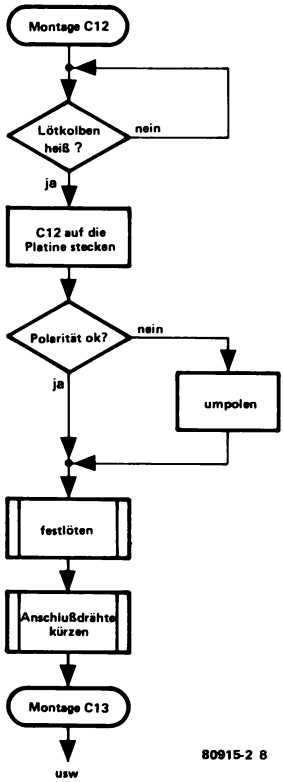


Bild 8. Detaillierte Flow Chart für die Montage und das Festlöten des Kondensators C12 auf der Basisplatine des Junior-Computers.

3

Programmieren

Nachdem der Junior-Computer gebaut ist und wir uns im 2. Kapitel genügend Basisinformation angeeignet haben, zeigen wir nun, wie sich der Junior-Computer programmieren läßt. Wie schreibt man ein Programm, wie reagiert der Computer auf dieses Programm oder wozu läßt sich der Junior-Computer gebrauchen? Diese Fragen beantworten wir ausführlich in diesem Kapitel.

So steht er nun vor uns, der Junior-Computer, mit seinem Display, seinen 23 Tasten und zwei Schaltern auf der Basisplatine. Wie er sich uns präsentiert, ist in Bild 1a gezeichnet. Nach dem Einschalten des Netzteiles und dem Drücken der RST-Taste leuchten sechs Displays auf. Diese zeigen willkürliche hexadezimale Zahlen. Das Drücken auf die RST-Taste hatte zur Folge, daß sich der Computer um das Monitorprogramm im EPROM kümmert. Durch diesen Monitor wird der Computer über das Keyboard ansprechbar. Die Folge davon ist, daß der Junior-Computer periodisch nachsieht, ob irgendeine Taste im Tastenfeld betätigt wurde. Handelt es sich um die Tasten Ø . . . F, dann wird die gedrückte Taste ins Display gebracht.
Die vier linken Displays zeigen hexadezimal die Adressen an. Im Prinzip sind alle Adressen von ØØØØ . . . FFFF möglich. Die zwei rechten Displays geben den Inhalt des Speicherplatzes an, der durch die Adresse auf dem Display angesprochen wird.
Nehmen wir einmal an, wir möchten bestimmte Speicherzellen im RAM mit Daten laden. Zum Beispiel in die Speicherzelle mit der Adresse Ø2ØØ die Daten 18, in die folgende A9 und so weiter. Wir gehen dabei wie folgt vor:

				Adresse	Daten	Display			Befehle
RST				xxxx	xx				
AD				xxxx	xx				
Ø	2	Ø	Ø	Ø2ØØ	xx				
DA				Ø2ØØ	xx				
		1	8	Ø2ØØ	18	Ø2ØØ	1	8	CLC
+		A	9	Ø2Ø1	A9	Ø2Ø1	A	9	LDA #
+		Ø	3	Ø2Ø2	Ø3	Ø2Ø2	Ø	3	
+		6	9	Ø2Ø3	69	Ø2Ø3	6	9	ADC #
+		Ø	7	Ø2Ø4	Ø7	Ø2Ø4	Ø	7	
+		8	D	Ø2Ø5	8D	Ø2Ø5	8	D	STA–
+		Ø	Ø	Ø2Ø6	ØØ	Ø2Ø6	Ø	Ø	
+		Ø	3	Ø2Ø7	Ø3	Ø2Ø7	Ø	3	
+		4	C	Ø2Ø8	4C	Ø2Ø8	4	C	JMP–
+		Ø	Ø	Ø2Ø9	ØØ	Ø2Ø9	Ø	Ø	
+		Ø	3	Ø2ØA	Ø3	Ø2ØA	Ø	3	
AD						Ø2ØB	X	X	
1	A	7	A	1A7A	xx	Ø2ØC	X	X	
DA				1A7A	xx	1A79	X	X	
		Ø	Ø	1A7A	ØØ	1A7A	Ø	Ø	
+		1	C	1A7B	1C	1A7B	1	C	
						1A7C	X	X	

Was haben wir nun im Einzelnen gemacht? Zuerst haben wir durch das Drücken der RST-Taste das Monitorprogramm aufgerufen. Das Adreß- und Datendisplay zeigt willkürliche hexadezimale Zahlen an. Durch ein "X" (X = don't care = belanglos) haben wir das im Adreß- und Datendisplay angedeutet. Durch Drücken der AD-Taste teilen wir dem Junior-Computer mit, daß wir ihm mit den Tasten Ø . . . F eine Adresse eingegeben möchten. Diese Adresse ist Ø2ØØ. Die Tasten Ø, 2 und Ø (2x) werden betätigt und die gewünschte Adresse steht auf den vier Adreßdisplays. Dann drücken wir auf die DA-Taste. Wir teilen somit dem Computer mit, daß wir mit den Tasten Ø . . . F Daten an der angezeigten Adresse ablegen möchten. Die ersten einzugebenden Daten sind 18. Nach dem Drücken der Tasten 1 und 8 stehen in der Speicherzelle mit der Adresse Ø2ØØ die Daten 18. Wenn wir vom Laden einer Speicherzelle sprechen, so ist das nicht ganz richtig: In Wirklichkeit haben wir die alten Daten in der betreffenden Speicherzelle durch neue überschrieben. Die Speicherzelle war ja nicht leer!
Durch Drücken der +/– Taste wird die Adresse im Display um Eins erhöht. Die Daten, die wir jetzt eingeben, werden nun bei dieser neuen Adresse abgelegt. Es ist dabei nicht nötig, die Datentaste aufs neue zu drücken. Der Junior-Computer merkt sich, ob zuletzt die DA- oder die AD-Taste gedrückt wurde. Will man eine neue Adresse eingeben (z.B. 1A7A) dann muß zuerst die Adreßtaste gedrückt werden, um dem Computer mitzu-

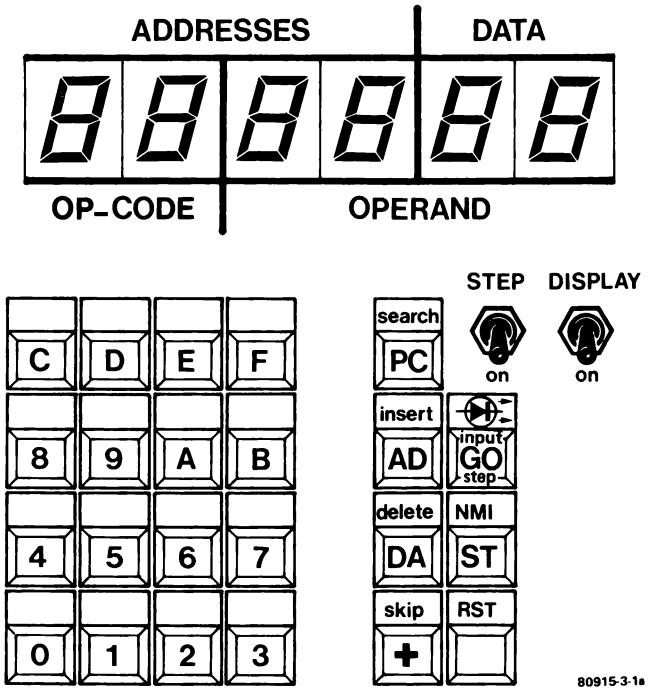


Bild 1a. So präsentieren sich Keyboard und Display des Junior-Computers dem Anwender. Neben den 16 hexadezimalen Datentasten sind noch sieben Funktionstasten und zwei Funktionsschalter vorhanden. Die Funktionstasten sind mit Doppelfunktionen belegt. Die Aufschrift PC, AD, DA, +, GO, ST und RST ist in diesem Buch für den Programmierer relevant. Im Buch Junior-Computer 2 gilt beim Start des Editors die Aufschrift SEARCH, INSERT, DELETE, SKIP und INPUT. Der Editor ist ebenfalls Bestandteil des Monitorprogramms.
Auch das Display hat eine Doppelfunktion. Hier wird es nur als Adreß-Datendisplay verwendet.

teilen, daß die eingegebenen Daten als Adressen interpretiert werden sollen.
Was haben wir nun gelernt, nachdem wir an bestimmten Adressen ein paar Daten abgelegt haben? Eine Menge! Denn die meisten Tasten des Keyboards sind uns jetzt vertraut:

- Die Tasten Ø . . . F dienen zur Eingabe von Adressen und zum Laden von Daten.
- Die AD-Taste. Durch diese Taste teilen wir dem Junior-Computer mit, daß wir mit den Tasten Ø . . . F Adressen eingeben möchten.
- Die DA-Taste. Durch das Drücken dieser Taste teilen wir dem Junior-Computer mit, daß wir mit den Tasten Ø . . . F Daten in bestimmten Speicherzellen des RAM ablegen möchten.
- Die +/– Taste. Das Drücken dieser Taste hat zur Folge, daß die im

Display angezeigte Adresse um Eins erhöht wird. Die Adreß- oder Daten-Mode wird dabei nicht geändert.

– Die RST-Taste. Durch diese Taste rufen wir das Monitorprogramm des Junior-Computers auf. Erst nach dem Drücken dieser Taste ist der Junior-Computer über das Keyboard ansprechbar. Diese Taste setzt den Computer automatisch in Adreß-Mode.

Bei der Eingabe von Adressen und Daten werden die gedrückten Tasten von rechts nach links ins Display geschoben. Beim Eingeben der Adresse 1A7A sieht das folgendermaßen aus: Nach 1: XXX1; nach A: XX1A; nach 7: X1A7; und zum Schluß nach A: 1A7A.

Beim Laden der Daten an den verschiedenen Adressen haben wir nicht willkürliche Daten eingegeben, sondern ein kleines Programm geladen. Ein Programm, das zwei hexadezimale Zahlen addiert. Ohne allzusehr ins Detail zu gehen zeigen wir, warum diese paar hexadezimalen Zahlen ein Programm sind, das der Junior-Computer verstehen kann:

Auf Adresse Ø2ØØ steht 18. Das ist der Code für die Instruktion CLC, CLear Carry. Nachdem die CPU diese Instruktion ausgeführt hat, kommt sie zur der folgenden Adresse Ø2Ø1. Dort steht A9, der Code für LDA, LoaD Accumulator immediate. Laden womit? Das steht auf der folgenden Adresse Ø2Ø2. Der Accu wird also mit Ø3 geladen. Auf der Adresse Ø2Ø3 steht 69. Das ist der Code für die ADC-Instruktion, ADd memory to Accumulator with Carry. Das heißt: Zähle (addiere) zum Inhalt des Accumulators die Zahl, die der ADC-Instruktion folgt. Der Inhalt der Speicherzelle mit der Adresse Ø2Ø4 ist Ø7. Der Inhalt des Accus ist Ø3. Der Inhalt des Accus nach der Ausführung der ADC-Instruktion ist ØA: denn Ø3 plus Ø7 ergibt die hexadezimale Zahl ØA. Jetzt sind wir neugierig, was mit diesem Accuinhalt geschieht.

Auf der Adresse Ø2Ø5 steht 8D. Das ist der Code für die STA-Instruktion, STore Accumulator in Memory. Anders ausgedrückt: Lege den Inhalt des Accumulators in einer bestimmten Speicherzelle ab. In welcher Speicherzelle? Die Adresse dieser Speicherzelle steht auf den folgenden Adressen Ø2Ø6 und Ø2Ø7. Zuerst steht das niederwertige Adreßbyte der Speicherzelle, in die der Inhalt des Accus geschrieben werden soll, dann das hochwertige Adreßbyte. Die Speicherzelle in die der Inhalt des Accus geschrieben wird, hat also die Adresse Ø3ØØ. Nach der Ausführung dieser STA-Instruktion kommt die CPU zu der Adresse Ø2Ø8. Dort steht 4C, das ist der Code für JMP (JuMP): springe zu einer bestimmten Adresse. Zu welcher Adresse soll gesprungen werden? Das steht in den beiden Speicherzellen, die der JMP-Instruktion folgen: Ø2Ø9 und Ø2ØA. In diesen Speicherzellen steht das nieder- und hochwertige Adreßbyte der Adresse, zu der gesprungen werden soll: Ø3ØØ. Dort finden wir das Resultat der vorangegangenen Addition vor: ØA. ØA kann eine hexadezimale Zahl sein, aber auch eine Instruktion! Die CPU interpretiert den Inhalt der Speicherzelle Ø3ØØ als eine Instruktion: ØA ist der Code von ASL-A = Arithmetic Shift Left Accumulator, d.h. schiebe den Inhalt des Accumulators um eine Bitposition nach links.

Das "wie und warum" dieses Programm abläuft, steht an dieser Stelle noch nicht zur Diskussion. Aber was wir daraus ersehen können ist, daß Instruktionen und Daten, die zu diesen Instruktionen gehören, hintereinander (sequenziell) im Speicher des Computers abgelegt sind.

Ein Instruktion ist in Form von hexadezimalen Zahlen im Speicher abgelegt. Eine Instruktion ist aber wiederum von Daten abhängig, welche die Instruktion genau beschreiben. Erinnern wir uns an die STA-Instruktion: 8D. In welcher Speicherzelle soll der Inhalt des Accus abgelegt werden? Die Daten dazu befinden sich in den beiden Speicherzellen, die der STA-Instruktion folgen: Ø3ØØ.

Daraus lernen wir, daß Daten und Maschinenbefehle gleichberechtigt in einem und demselben Speicher abgelegt sind. Es wird also kein separater Speicher für die Befehle und die Daten benötigt! Diese Idee hatte zum ersten Mal in den vierziger Jahren ein gewisser Herr Neumann; Maschinenbefehle und die zu ihnen gehörenden Daten liegen gleichberechtigt und sequenziell im Speicher des Mikrocomputers. Der Computer arbeitet sich nach dem "Neumann-Zyklus" durch das Programm. Von der CPU aus gesehen, läuft dieser Zyklus folgendermaßen ab:

- Kode des momentanen Maschinenbefehles holen (zum Beispiel 8D = STA).
- Maschinenbefehl dekodieren (es sollen Daten gespeichert, *nicht* geladen, manipuliert oder zu einer neuen Adresse gesprungen werden. Die Information steht im Kode 8D).
- Operand holen. Unter Operand sind Daten zu verstehen, die den Maschinenbefehl näher bezeichnen. (Im Beispiel ist der Maschinenbefehl 8D = STA. Wohin sollen die Daten gespeichert werden? Der Operand, das sind die beiden Bytes, die dem Maschinenbefehl folgen, geben darüber Auskunft: die Daten sollen in der Speicherzelle Ø3ØØ abgelegt werden).
- Die Instruktion ausführen. Unter einer Instruktion versteht man den Maschinenbefehl mit dem dazugehörenden Operanden. Während die Instruktion ausgeführt wird, sieht der Mikroprozessor nach, an welcher Adresse er den folgenden Maschinenbefehl findet. Dieser Befehl wird dann wieder wie zuvor beschrieben abgearbeitet. Nach dem "Neumann-Zyklus" arbeitet sich also die CPU durch das Programm.

Das Programmiermodell der 6502-CPU

Bevor wir tiefer die Programmiertechnik der 6502-CPU oder des Junior-Computers eingehen, geben wir erst eine Übersicht von der internen Registerstruktur des Mikroprozessors. Der Inhalt der internen CPU-Register ist nämlich durch den Programmierer über das Programm beeinflußbar. Und bevor man diese Register beeinflussen kann, muß bekannt sein, was es zu beeinflussen gibt. Alle internen Register der 6502-CPU sind in Bild 1b dargestellt.

Der Accumulator "A" (*Accu*) ist ein 8-Bit-Register. Er wird als Zwischenstation benötigt, wenn Daten aus den Speicher geholt werden, um sie anschließend in eine andere Speicherzelle zu transportieren. Die meisten Operationen, die ein Programm ausführt, laufen über den Accumulator.

Eine weitere CPU-Einheit ist die ALU, eine Abkürzung von Arithmetic Logic Unit. Diese Einheit arbeitet eng mit dem Accumulator zusammen. Technisch gesehen hat die ALU zwei 8-Bit Eingänge und einen 8-Bit Ausgang. Die Daten, die über Befehle den beiden ALU-Eingängen zugeleitet werden, verarbeitet diese Einheit, das Ergebnis dieser Operationen steht am Schluß im Accu. Wie der Name ALU schon sagt, werden in dieser

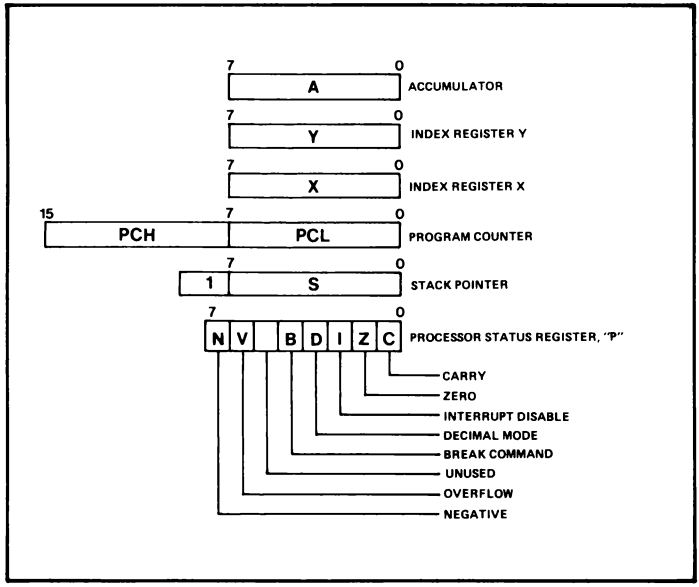


Bild 1b. Die CPU-Register des 6502-Mikroprozessors. Der Inhalt dieser Register läßt sich vom Programm beeinflussen.

Einheit sowohl arithmetische wie logische Operationen durchgeführt. Eine arithmetische Operation ist zum Beispiel eine Addition oder Subtraktion. Logische Operationen sind z.B. UND-, ODER-, EXCLUSIV ODER-Verknüpfungen.
Ein weiteres internes CPU-Register ist das *Prozessor Status* "P". Dieses Register gibt in Form von Flags die Zustände von verschiedenen Operationen an. Flags (flag = Fahne) sind mit D-Flipflops zu vergleichen. Flags werden nicht nur als Ergebnis von Operationen gesetzt oder zurückgesetzt, sondern lassen sich auch durch Befehle beeinflussen.
Ein Flag im Prozessor Status Register ist das Carry Flag "C". Es wird gesetzt (log. 1), wenn zum Beispiel bei der Addition von zwei Zahlen ein Übertrag vom achten auf das neunte Bit erfolgt. Ist bei dieser Addition kein Übertrag notwendig, wird das Carry Flag im P-Register zurückgesetzt. Ein weiteres Flag ist das N-Flag. Diese Flag wird gesetzt, wenn das Ergebnis einer Operation negativ ist; im anderen Fall wird es zurückgesetzt. Ähnlich ist es bei dem Z-Flag im P-Register. Es gibt an, ob das Ergebnis einer Operation Null ist oder nicht. Das Prozessor Status Register werden wir an anderer Stelle noch ausführlich besprechen, aber seine Funktionsweise dürfte nun in groben Zügen klar geworden sein. Die 6502-CPU hat Befehle, durch die sich die Flags im P-Register testen lassen. Der Programmierer kann somit modifizierend oder korrigierend auf das Programm einwirken.

Der Programm Zähler oder *Program Counter "PC"* ist ein 16-Bit Register. Er ist aufgeteilt in die Register PCL (L = Low) und PCH (H = High). Dieser Zähler führt den Mikroprozessor durch das Programm. Erinnern wir uns an den "Neumann-Zyklus"! Der Inhalt des Programmzählers ist immer eine Adresse. PC zeigt zuerst auf einen Maschinenbefehl, der sich irgendwo im Speicher befindet. Hat dieser Befehl einen oder mehrere Operanden, werden diese ebenfalls vom Programmzähler adressiert und über den Datenbus in die CPU geholt.
Noch zwei weitere Register sind in der 6502-CPU enthalten: Das X- und das Y-Register. Der Inhalt beider Register läßt sich durch Instruktionen beeinflussen. Das X- und das Y-Register können wir wie den Accu als ein Zwischenspeicher ansehen, sie sind aber auch für verschiedene Adressierungsarten vorgesehen. Bei der indizierten und indirekten Adressierung kommen wir im Detail auf diese Register, genannt Index-Register, zu sprechen.
Auch *den Stack Pointer "S"* werden wir an anderer Stelle genau erklären. Mit diesem Pointer ist es möglich, auf den Stack beim Sprung in Unterprogramme Adressen abzulegen, die bei der Rückkehr aus dem Unterprogramm wieder benötigt werden. Und noch viele andere Möglichkeiten erschließen sich dem Programmierer durch diesen Pointer.

Das Adressierungs- und Befehlsrepertoir

Dreizehn verschiedene Adressierungsarten stehen dem Programmierer des Junior-Computers zur Verfügung. Diese Vielzahl von Adressierungsarten sind der starke Punkt der 6502-CPU. Das Instruction Set oder Befehlsrepertoire ist nicht besonders groß: 56 Befehle. Die Kombination von einer beschränkten Anzahl "kräftiger" Befehle mit einer großen Skala an Adressierungsarten sorgen für eine optimale Programmierfreundlichkeit. Die meisten CPU-Hersteller anderer Prozessoren schließen sich diesem 6502-Trend an. Warum? Das ist leicht zu beantworten:
Wenig Instruktionen heißt, der Programmierer muß sich nicht viele Maschinenbefehle merken. Alle Maschinenbefehle sind durch einen hexadezimalen Code und drei große Buchstaben gekennzeichnet. Diese drei großen Buchstaben beschreiben in Stenoschreibweise den Maschinenbefehl und werden als "Mnemonics" bezeichnet. Mnemonic, zu deutsch Mnemonik, bedeutet Gedächtniskunst. Die Gedächtniskunst besteht darin, hexadezimale Zahlen, die Maschinenbefehle darstellen, stenoartig in Worten zu beschreiben. Die wenigen, aber "kräftigen" Maschinenbefehle der 6502-CPU lassen sich zusammen mit einer breiten Adressierungspalette optimal und einfach einsetzen.

Immediate Addressing

(unmittelbare Adressierung)

Immediate Addressing bezieht sich auf die internen CPU-Register. Welches Register angesprochen ist, darüber geben die Mnemonics Auskunft. Instruktionen dieser Adressierungsart bestehen aus zwei Bytes. Das erste Byte repräsentiert den Maschinenbefehl oder OP-Code, das zweite Byte sind beliebige Daten. Den unmittelbaren Charakter dieser Instruktionen symbolisiert das Zeichen ("railroad crossing": #). Beispiele für solche Instruktionen sind:

LDA # 7A: lade den Accu mit 7A oder
LDX # 3B: lade das X-Register mit 3B oder
LDY # 2F: lade das Y-Register mit 2F.
Ein weiteres Beispiel für diese Adressierungsart ist der Befehl ADC. Die ausführliche Schreibweise für ADC ist: A+M+C→A. Das heißt, addiere zum Inhalt des Accumulators den Inhalt der Speicherzelle, die unmittelbar dem ADC-Befehl folgt. Addiere dazu auch noch das C-Flag. Vor einer Addition muß dieses Flag immer Null gemacht werden. Das läßt sich mit dem CLC-Befehl (CLear Carry) erreichen. Sehen wir uns ein solches Additionsprogramm näher an:

```
CLC       clear carry (C=Ø)
LDA # 13  lade den Accu mit 13
ADC # Ø8  addiere dazu Ø8
BRK       stop, sobald die Addition beendet ist.
```

Der BRK-Befehl (BRK=BReaK) am Schluß des Programms ist nötig, um den Computer nach Ausführung der Addition zu stoppen. Wie übertragen wir nun dieses Programm auf den Junior-Computer, um es anschließend auszuführen? Zuerst suchen wir die Befehls-Codes der einzelnen Maschinenbefehle aus der Tabelle am Ende dieses Buches (Instruction Set). Wir finden: CLC = 18; LDA # = A9; ADC # = 69; BRK = ØØ. Als Startadresse wählen wir Ø1ØØ. Dann gehen wir wie folgt vor:
(Computer einschalten, Display-Schalter "ON", Step-Schalter "OFF")

				Adresse	Daten	
RST	AD			xxxx	xx	
Ø	1	Ø	Ø	Ø1ØØ	xx	
DA		1	8	Ø1ØØ	18	CLC
+		A	9	Ø1Ø1	A9	LDA #
+		1	3	Ø1Ø2	13	
+		6	9	Ø1Ø3	69	ADC #
+		Ø	8	Ø1Ø4	Ø8	
+		Ø	Ø	Ø1Ø5	ØØ	BRK
AD				Ø1Ø5	ØØ	
1	A	7	E	1A7E	xx	
DA		0	0	1A7E	ØØ	
+		1	C	1A7F	1C	
AD						
Ø	1	Ø	Ø	Ø1ØØ	18	
GO				Ø1Ø7	xx	Programmstart
AD				Ø1Ø7	xx	
Ø	Ø	F	3	ØØF3	1B	Resultat

Ein paar Dinge müssen wir noch erklären. Warum der plötzliche Sprung von Adresse Ø1Ø5 zur Adresse 1A7A? Und warum steht das Resultat auf der Adresse ØØF3? Nach der BRK-Instruktion auf der Adresse Ø1Ø5 ist

das eigentliche Programm beendet. Was danach folgt, sind Prozeduren, die mit dem Monitorprogramm des Junior-Computers zu tun haben. Nach dem Abarbeiten der BRK-Instruktion kümmert sich der Mikroprozessor um die Adressen 1A7E und 1A7F. In diesen beiden Speicherplätzen ist die Startadresse (1CØØ) einer Save Routine abgelegt. Diese Save Routine ist ein Teil des Monitors und speichert alle internen CPU-Register in bestimmten RAM-Zellen ab, wenn die CPU auf einen BRK-Befehl stößt. Die jeweiligen Speicherzellen haben folgende Adressen:

ØØEF: Inhalt von PCL
ØØFØ: Inhalt von PCH
ØØF1: Inhalt des P-Registers
ØØF2: Inhalt des Stack Pointers "S"
ØØF3: Inhalt des Accus "A"
ØØF4: Inhalt des Y-Registers
ØØF5: Inhalt des X-Registers.

In der Speicherzelle mit der Adresse ØØF3 wird also der Inhalt des Accumulators abgelegt. Deshalb steht auch in dieser Speicherzelle das Ergebnis der Addition: 1B. Doch bevor wir in dieser Speicherzelle das Ergebnis nachsehen können, muß das Additionsprogramm gestartet werden. Dazu geben wir die Startadresse ØlØØ ein und starten mit der GO-Taste das Programm.

Wie sich mit dem Junior-Computer Zahlen addieren lassen, läßt sich auch eine Subtraktion ausführen. Die Schreibweise für diesen Befehl ist SBC, was heißt A-M-$\overline{C}$ → A. Oder: Ziehe vom Inhalt des Accus den Inhalt der Speicherzelle ab, die unmittelbar dem SBC-Befehl folgt. Ziehe weiter davon das negierte C-Flag ab. Das C-Flag ist negiert, da sich der Mikroprozessor bei der Subtraktion des Zweierkomplements bedient (siehe 2. Kapitel). Das hat zur Folge, daß vor einer Subtraktion das C-Flag gesetzt werden muß. Mit dem SEC-Befehl (SEt Cary) läßt sich diese Forderung erfüllen. Ein Subtraktionsprogramm sieht dann wie folgt aus:

```
SEC      set carry
LDA 13   lade den Accu mit 13
SBC Ø8   subtrahiere davon Ø8
BRK      stop, sobald die Subtraktion beendet ist.
```

Am Ende des Buches holen wir wieder aus dem Instruction Set die OP-Codes für die verwendeten Befehle: SEC = 38; LDA = A9; SBC = E9; BRK = ØØ. Als Startadresse wählen wir ØlØØ und geben das Programm wie folgt in den Junior-Computer ein:

				Adresse	Daten	
RST	AD			xxxx	xx	
Ø	1	Ø	Ø	ØlØØ	xx	
DA		3	8	ØlØØ	38	SEC
+		A	9	Ø1Ø1	A9	LDA #
+		1	3	Ø1Ø2	13	
+		E	9	Ø1Ø3	E9	SBC #
+		Ø	8	Ø1Ø4	Ø8	
+		Ø	Ø	Ø1Ø5	ØØ	BRK

AD				Ø1Ø5	ØØ	siehe Bemerkung
1	A	7	E	1A7E	xx	
DA		Ø	Ø	1A7E	ØØ	
+		1	C	1A7F	1C	
AD						
Ø	1	Ø	Ø	Ø1ØØ	38	
GO				Ø1Ø7	xx	Programmstart
AD				Ø1Ø7	xx	
Ø	Ø	F	3	ØØF3	ØB	Resultat

Bemerkung: Das Laden von Daten in die Speicherzellen 1A7E und 1A7F ist nur dann wichtig, wenn der Junior-Computer vor dem Laden des Subtraktionsprogrammes ausgeschaltet war. In der Speicherzelle ØØF3 finden wir wieder das Ergebnis der Subtraktion: ØB. Denn 13 (dezimal 19) minus Ø8 (dezimal Ø8) ist ØB (dezimal 11).

Bisher verwendeten wir nur die arithmetischen Befehle ADC und SBC. Die 6502-CPU hat jedoch auch eine Anzahl von logischen Befehlen. Mit diesen Befehlen lassen sich AND-, OR- und Exclusiv OR- Verknüpfungen durchführen. Eine OR-Verknüpfung läßt sich mit dem Befehl ORA durchführen. Die ausführliche Schreibweise dieses Befehles ist:
A $\vee$ M → A. Der OP-Code dieses Befehles ist Ø9. Ein Programm soll diese Instruktion erklären:

```
LDA #AA   lade den Accu mit AA
ORA #ØF   führe Bit für Bit eine OR-Verknüpfung durch
BRK       stop, wenn OR-Verknüpfung durchgeführt ist.
```

AA ist binär: 1Ø1Ø1Ø1Ø
ØF ist binär: ØØØØ1111
Resultat: 1Ø1Ø1111 (AF; steht im Accu)

Mit Bits gleicher Wertigkeit wird also eine OR-Funktion durchgeführt. Bild 2a verdeutlicht das, indem die ODER-Verknüpfung mit Hardware-Gattern durchgeführt wird. Eine andere logische Operation ist die AND-Verknüpfung. Der Befehl, der diese Verknüpfung durchführt, heißt AND. Seine Funktion ist: A $\wedge$ M → A. Der OP-Code von AND ist 29. Ein Programm soll die Wirkungsweise dieses Befehles verdeutlichen:

```
LDA #AA   lade den Accu mit AA
AND #ØF   führe Bit für Bit eine AND-Verknüpfung aus
BRK       stop, wenn AND-Verknüpfung durchgeführt ist.
```

AA ist binär: 1Ø1Ø1Ø1Ø
ØF ist binär: ØØØØ1111
Resultat: ØØØØ1Ø1Ø (ØA; steht im Accu)

Mit Bits gleicher Wertigkeit wird eine AND-Verknüpfung durchgeführt. Bild 2b gibt die Hardwarelösung mit AND-Gattern wieder.

Die dritte logische Operation ist die Exclusive OR-Verknüpfung. Die Funktion ist: A $\veebar$ M → A. Die Schreibweise ist EOR mit dem OP-Code 49. Ein Programm soll auch diese Verknüpfung verdeutlichen:

```
LDA #AA   lade den Accumulator mit AA
EOR #ØF   führe Bit für Bit eine EOR-Verknüpfung durch
BRK       stop, wenn EOR-Verknüpfung durchgeführt ist.
```

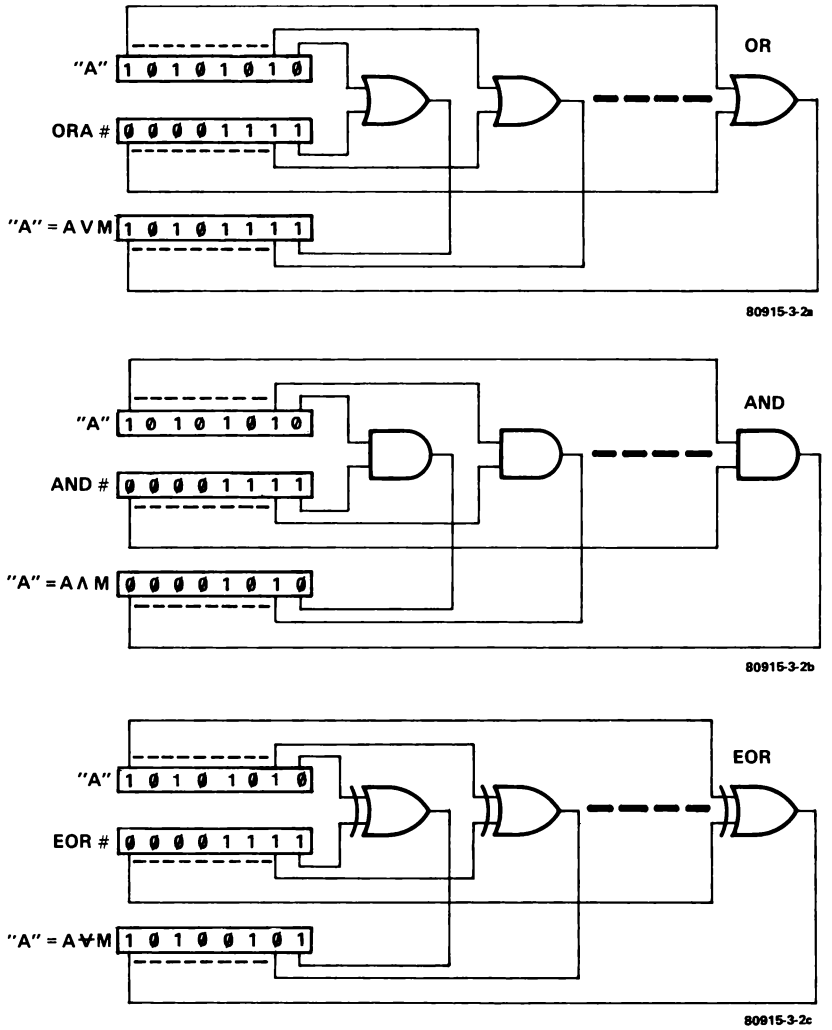


Bild 2. Die logische Ausführung der Instruktionen ORA, AND und EOR ist hier symbolisch mit Hardware-Gattern ausgeführt.

AA ist binär: 10101010
0F ist binär: 00001111
Resultat: 10100101 (A5; steht im Accu)
Mit Bits gleicher Wertigkeit wird eine Exclusive OR-Verknüpfung durchgeführt. Diese drei Beispiele geben wir nun in den Junior-Computer ein. Die Tasten sind wie folgt zu drücken:

```
RST   AD                    xxxx   xx
 Ø    1    Ø    Ø          ØlØØ   xx
DA         A    9          ØlØØ   A9    LDA#      Beginn 1
 +         A    A          Ø1Ø1   AA
 +         Ø    9          Ø1Ø2   Ø9    ORA#
 +         Ø    F          Ø1Ø3   ØF
 +         Ø    Ø          Ø1Ø4   ØØ    BRK       Ende 1
 +         A    9          Ø1Ø5   A9    LDA#      Beginn 2
 +         A    A          Ø1Ø6   AA
 +         2    9          Ø1Ø7   29    AND#
 +         Ø    F          Ø1Ø8   ØF
 +         Ø    Ø          Ø1Ø9   ØØ    BRK       Ende 2
 +         A    9          Ø1ØA   A9    LDA#      Beginn 3
 +         A    A          Ø1ØB   AA
 +         4    9          Ø1ØC   49    EOR#
 +         Ø    F          Ø1ØD   ØF
 +         Ø    Ø          Ø1ØE   ØØ    BRK       Ende 3
AD
 Ø    1    Ø    Ø          ØlØØ   A9    Startadresse 1
GO                         Ø1Ø6   AA    Stop 1
AD
 Ø    Ø    F    3          ØØF3   AF    Resultat 1
AD
 Ø    1    Ø    5          Ø1Ø5   A9    Startadresse 2
GO                         Ø1ØB   AA    Stop 2
AD
 Ø    Ø    F    3          ØØF3   ØA    Resultat 2
AD
 Ø    1    Ø    A          Ø1ØA   A9    Startadresse 3
GO                         Ø11Ø   xx    Stop 3
AD
 Ø    Ø    F    3          ØØF3   A5    Resultat 3
```

Da wir jedes der drei Programme mit der BRK-Instruktion beenden, lassen sich alle hintereinander schreiben. Es ist jedoch darauf zu achten, daß vor jedem Programmstart mit der GO-Taste die richtige Startadresse eingegeben wird, da die BRK-Instruktion einen "Bremsweg" von zwei Adressen benötigt. Das hängt mit der internen Hardware-Struktur der 6502-CPU zusammen.
Die soeben besprochenen logischen Instruktionen haben in Programmen eine wichtige Bedeutung. Sie werden benötigt, um bestimmte Bits in einem Byte zu beeinflussen, daß heißt, Bits setzen, Bits zurücksetzen oder Bits zu invertieren. Zusammenfassend läßt sich sagen:

AND-Instruktion: Bits werden mit ihr zurückgesetzt (log. 0)
OR-Instruktion: Bits werden mit ihr gesetzt (log. 1)
EOR-Instruktion: Bits werden mit ihr invertiert.
Soviel zu Befehlen mit Immediate addressing. Acht Befehle haben wir bereits kennen gelernt: LDA #, LDX #, LDY #, ADC #, SBC #, ORA #, AND #, EOR #. Dann noch die Befehle CLC, SEC, BRK, die wir noch bei einem anderen Adressierungsverfahren, dem Implied Addressing, genau besprechen werden.

Absolute Addressing

Absolute oder direkte Addressierung

Bei Immediate Addressing werden die Daten, die unmittelbar dem OP-Code folgen, in ein internes Register der CPU geladen. Diese Register waren der Accu, das X- und das Y-Register. Es ist jedoch auch möglich in diese Register Daten zu laden, die an einer bestimmten Adresse abgelegt sind. Instruktionen mit Absolute Addressing sind drei Bytes lang und sind wie folgt aufgebaut:
OP-Code, ADL und ADH. ADL und ADH sind zwei Adreßbytes, aus denen Daten geholt, oder in die Daten geschrieben werden. Das folgende Programm soll zeigen, was damit gemeint ist:

```
LDA-0100 lade den Accu mit dem Inhalt der Speicherzelle 0100
STA-015A speichere den Accu-Inhalt in die Speicherzelle 015A
LDA-0101 lade den Accu mit dem Inhalt der Speicherzelle 0101
STA-015B speichere den Accu-Inhalt in die Speicherzelle 015B
LDA-0102 lade den Accu mit dem Inhalt der Speicherzelle 0102
STA-015C speichere den Accu-Inhalt in die Speicherzelle 015C
LDA-0103 lade den Accu mit dem Inhalt der Speicherzelle 0103
STA-015D speichere den Accu-Inhalt in die Speicherzelle 015D
LDA-0104 lade den Accu mit dem Inhalt der Speicherzelle 0104
STA-015E speichere den Accu-Inhalt in die Speicherzelle 015E
```

Nebenbei bemerkt: Der Strich zwischen OP-Code und Adresse gibt den absoluten Adressierungscharakter an. Was macht dieses Programm? Der Inhalt der Speicherzellen mit den Adressen 0100 . . . 0105 wird in die Speicherzellen 015A . . . 015E kopiert. Die Zwischenstation für die Lade- und Schreibeoperationen ist der Accu. Nachdem der Inhalt der Speicherplätze 0100 . . . 0105 in einen anderen Speicherbereich kopiert wurde, stehen an diesen Adressen noch die gleichen Daten. Bild 3 verdeutlicht den Kopiervorgang. In diesem Beispiel ist der LDA-Befehl bereits von früher bekannt. Der einzige Unterschied ist, daß das "#" durch einen Strich "–" ersetzt wurde. Neu ist der STA-Befehl. Speicherbefehle schreiben den Inhalt von Accu, X- oder Y-Register in bestimmte Speicherzellen. Die symbolische Schreibweise ist A→M. Wie schon anfangs erwähnt, bestehen Instruktionen mit Absolute Addressing aus drei Bytes. Das erste Byte ist der OP-Code. Das zweite Byte ist das niederwertige Adreßbyte ADL (L=Low) und das dritte Byte der Instruktion ist das hochwertige Adreßbyte ADH (H=High).
Um uns weiter mit Absolute Addressing vertraut zu machen, schreiben wir ein weiteres Programm, das zwei 16-Bit Zahlen zusammenzählt. Damit

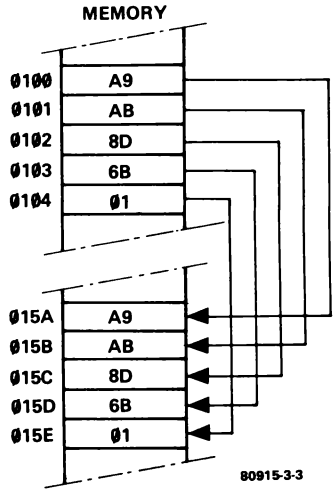


Bild 3. So funktioniert das Kopierprogramm bei Absolute Addressing!

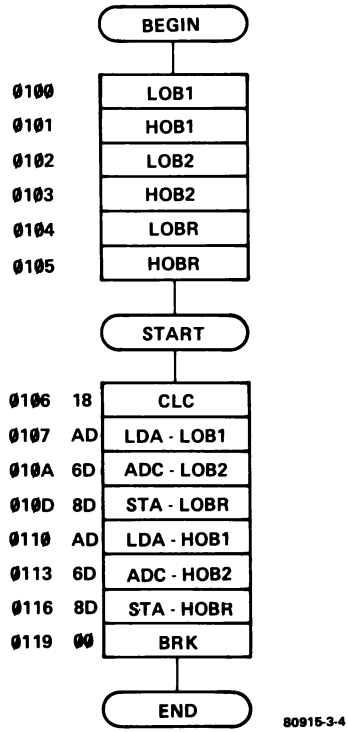


Bild 4. Flußdiagramm (Flow Chart) eines Programms, das zwei 16-Bit Zahlen addiert.

das Programm übersichtlich wird, ordnen wir den Zahlen symbolische Namen zu:

1. Zahl: HOB1, LOB1. Dabei bedeutet HOB1 = High Order Byte der ersten Zahl, LOB1 = Low Order Byte der ersten Zahl.
2. Zahl: HOB2, LOB2. HOB2 = High Order Byte der zweiten Zahl, LOB2 = Low Order Byte der zweiten Zahl.

Das Resultat ist wieder eine 16-Bit Zahl, bei der eventuell ein Übertrag der 16-ten auf die 17-te Stelle erfolgt. Ein solcher Übertrag wird durch ein gesetztes C-Flag angedeutet. Auch dem Resultat der Addition ordnen wir einen symbolischen Namen zu: HOBR, LOBR.

Bevor wir das Additionsprogramm im Detail besprechen, sehen wir uns den Ablaufplan in Bild 4 an: die Flow Chart. Ab Adresse Ø1ØØ sind sechs Speicherplätze für die beiden zu addierenden Zahlen und das Ergebnis dieser Addition reserviert. Die Speicherplätze haben die Adressen Ø1ØØ . . . Ø1Ø5. Vor dem Programmstart legen wir in den Speicherzellen LOB1, HOB1 und LOB2, HOB2 die beiden zu addierenden Zahlen ab. Das Programm schreibt dann in die Speicherzellen LOBR und HOBR das Resultat der Addition. Das eigentliche Additionsprogramm beginnt bei Adresse Ø1Ø6 und endet mit der Adresse Ø119. Bevor wir das Programm ausführen können, müssen wir es in den Junior-Computer eingeben. Mit Hilfe des Programms wollen wir die beiden hexadezimalen Zahlen Ø4EF (1. Zahl) und 23AB (2. Zahl) addieren. Das geschieht folgendermaßen:

(STEP: OFF; DISPLAY: ON)

RST	AD				xxxx	XX	
1	A	7	A		1A7A	XX	
DA		Ø	Ø		1A7A	ØØ	STEP-Vorbereitung (folgender Abschnitt)
+		1	C		1A7B	1C	
+ +					1A7D	XX	
+		Ø	Ø		1A7E	ØØ	BRK-Vorbereitung
+		1	C		1A7F	1C	
AD					1A7F	1C	
Ø	1	Ø	Ø		Ø1ØØ	XX	
DA		E	F		Ø1ØØ	EF	LOB1
+		Ø	4		Ø1Ø1	Ø4	HOB1
+		A	B		Ø1Ø2	AB	LOB2
+		2	3		Ø1Ø3	23	HOB2
+					Ø1Ø4	XX	Speicherplatz für LOBR
+					Ø1Ø5	XX	Speicherplatz für HOBR
+		1	8	CLC	Ø1Ø6	18	clear Carry-Flag
+		A	D	LDA-	Ø1Ø7	AD	LOB1 in A
+		Ø	Ø		Ø1Ø8	ØØ	
+		Ø	1		Ø1Ø9	Ø1	
+		6	D	ADC-	Ø1ØA	6D	A+LOB2→A (Carry-Flag zählt mit)
+		Ø	2		Ø1ØB	Ø2	
+		Ø	1		Ø1ØC	Ø1	

+		8	D	STA-	Ø1ØD	8D	Resultat (=LOBR) zur Adresse Ø1Ø4
+		Ø	4		Ø1ØE	Ø4	
+		Ø	1		Ø1ØF	Ø1	
+		A	D	LDA-	Ø11Ø	AD	HOB1 in A
+		Ø	1		Ø111	Ø1	
+		Ø	1		Ø112	Ø1	
+		6	D	ADC-	Ø113	6D	A+HOB2→A (Carry-Flag zählt mit)
+		Ø	3		Ø114	Ø3	
+		Ø	1		Ø115	Ø1	
+		8	D	STA-	Ø116	8D	Resultat (=HOBR) zur Adresse Ø1Ø5
+		Ø	5		Ø117	Ø5	
+		Ø	1		Ø118	Ø1	
+		Ø	Ø	BRK	Ø119	ØØ	Ende des Programms
AD							
Ø	1	Ø	6		Ø1Ø6	18	Startadresse
GO					Ø11B	xx	Programm-Start
AD							
Ø	1	Ø	4		Ø1Ø4	9A	Resultat: LOBR
+					Ø1Ø5	28	Resultat: HOBR

Nachdem wir uns überzeugt haben, daß das Programm die beiden hexadezimalen Zahlen Ø4EF und 23AB addiert hat, vollziehen wir im einzelnen nach, wie dieses Programm im Computer abläuft. Dazu schreiben wir die beiden Zahlen binär an:

1. Zahl: Ø4EF = ØØØØ Ø1ØØ 1110 1111
2. Zahl: 23AB = ØØ1Ø ØØ11 1Ø1Ø 1Ø11
←HOB→←LOB→

Zuerst wird bei der Adresse Ø1Ø6 die Carry Flag zurückgesetzt und das niederwertige Byte "LOB1" in den Accu geladen. Dazu wird das niederwertige Byte der 2. Zahl "LOB2" addiert, wobei die C-Flag nach erfolgter Addition gesetzt ist:

```
          1110 1111   LOB1
          1Ø1Ø 1Ø11   LOB2
      +    111 1111   carry
Carry  1  1ØØ11Ø1Ø   LOBR
          ←9→←A→
```

Dann speichert der Computer in der Speicherzelle Ø1Ø4 das Ergebnis "LOBR". Im Anschluß daran lädt der Accu das hochwertige Byte der 1. Zahl "HOB1"; dazu wird das hochwerige Byte der 2. Zahl "HOB2" plus Carry von der ersten Addition addiert. Im Computer sieht das folgendermaßen aus:

```
          ØØØØØ1ØØ   HOB1
          ØØ1ØØØ11   HOB2
                   1  Carry von LOBR
      +       111     Carry
Carry Ø ØØ1Ø1ØØØ     HOBR
        2     8
```

Nach der zweiten Addition ist die C-Flag zurückgesetzt. Das Ergebnis dieser Addition wird anschließend in die Speicherzelle Ø1Ø5 abgelegt. Somit können wir bei den Adressen Ø1Ø4 und Ø1Ø5 das Ergebnis der Addition nachschlagen. Übrigens muß nach dem Einschalten des Junior-Computers bei den Adressen 1A7A, 1A7B sowie 1A7E, 1A7F die Startadresse der Monitor Saveroutine abgelegt werden. Diese Startadresse ist bekanntlich 1CØØ. Nur dann läßt sich das Additionsprogramm in "Step by step mode" durchlaufen (die Besprechung folgt in nächsten Abschnitt) und das BRK-Kommando durchführen.
Eine neue Adressierungsart lernten wir kennen: Absolute Addressing. Das Additionsprogramm machte fast nur von Instruktionen Gebrauch, die diese Adressierungsart verwenden. Instruktionen mit Absolute Addressing sind immer drei Bytes lang: OP-Code – niederwertiges Adreßbyte – hochwertiges Adreßbyte. Es gibt noch wesentlich mehr Instruktionen, die Absolute Addressing verwenden. In der folgenden Übersicht sind diese Instruktionen zusammengestellt:
Lade- und Schreibbefehle:

LDA- Code AD (M → A) load accu with memory
LDX- Code AE (M → X) load index X with memory
LDY- Code AC (M → Y) load index Y with memory
STA- Code 8D (A → M) store accumulator in memory
STX- Code 8E (X → M) store index X in memory
STY- Code 8C (Y → M) store index Y in memory

Arithmetische Befehle:

ADC- Code 6D (A+M+C → A) add memory to accumulator with carry
SBC- Code ED (A−M−$\overline{C}$ → A) subtract memory from accumulator with carry

sowie Increment- und Decrement-Befehle:

INC- Code EE (M+1 → M) increment memory by one
DEC- Code CE (M−1 → M) decrement memory by one

Diese Befehle erhöhen (increment) oder erniedrigen (decrement) den Inhalt einer bestimmten Speicherzelle um Eins. Logische Befehle:

ORA- Code ØD (A ∨ M → A) OR memory with accumulator
AND- Code 2D (A ∧ M → A) AND memory with accumulator
EOR- Code 4D (A ⊻ M → A) Exclusive OR memory with accumulator

Bevor wir weitere Adressiermöglichkeiten der 6502-CPU beschreiben, erklären wir eine neue, interessante Programmierart des Junior-Computers, die "step by step mode".

Step by Step Mode: Schrittweise ein Programm durchlaufen

Das Drücken der GO-Taste hat zur Folge, daß ein Programm bei einer bestimmten Adresse startet. Mit der BRK-Instruktion stoppen wir das Programm, nachdem es ausgeführt ist. Der Schalter STEP steht dabei in der Position OFF. Bringen wir diesen Schalter in die Position ON, dann läßt sich mit der GO-Taste ein Programm schrittweise durchlaufen. Damit der Junior-Computer dieses schrittweise Durchlaufen eines Programms ausführen kann, muß die Speicherzelle 1A7A mit ØØ und die Speicherzelle 1A7B mit 1C geladen werden. Warum, das besprechen wir am Schluß

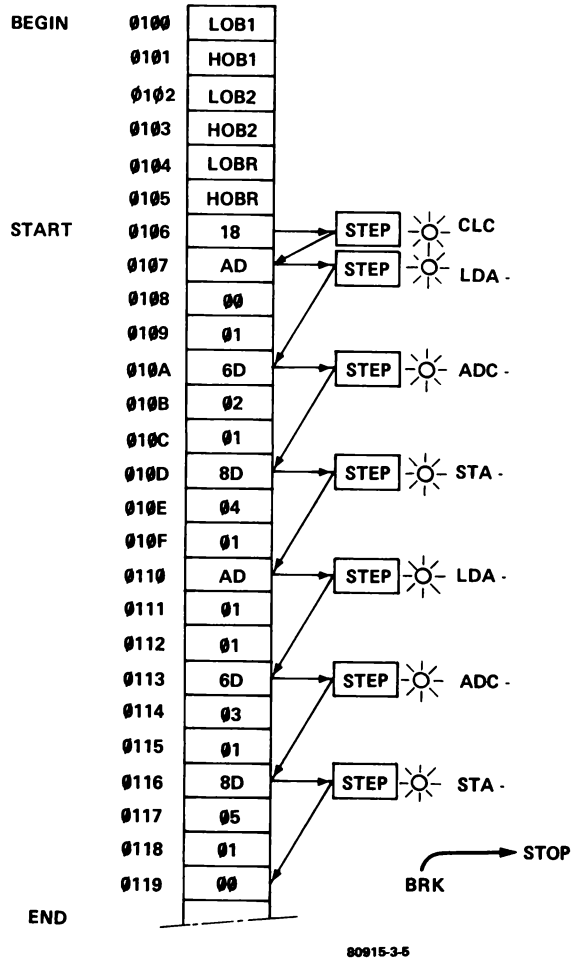
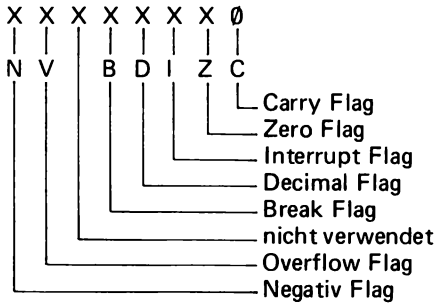


Bild 5. Durchlaufen des Additionprogramms von Bild 4 in step by step mode. Dabei springt nach dem Drücken der GO-Taste der Prozessor von OP-Code zu OP-Code. Leuchtet die LED in der GO-Taste, befindet sich der Computer in step by step mode.

dieses Kapitels. Wie man mit der GO-Taste das Additionsprogramm schrittweise (=step by step mode) durchlaufen kann, ist in Bild 5 dargestellt. Dazu bringen wir die Adresse Ø1Ø6, die Startadresse des Additionsprogramms ins Display. Die LED in der GO-Taste leuchtet, da der Schalter STEP in der Position ON steht. Die Instruktion bei der Startadresse des Additionsprogramms ist CLC mit dem OP-Code 18. Drücken wir nun auf die GO-Taste, führt der Junior-Computer die CLC-Instruktion aus und im Display erscheint der OP-Code der folgenden Instruktion mit der zu ihm gehörenden Adresse Ø1Ø7 AD = LDA—. Jetzt wollen wir wissen, ob die

C-Flag tatsächlich Null ist. Wie wir bereits wissen, werden alle internen CPU-Register in bestimmte Speicherzellen kopiert. Die C-Flag ist ein Teil des P-Registers, das in die Speicherzelle mit der Adresse ØØF1 gerettet wird. Deshalb drücken wir die Tasten AD Ø Ø F 1 und im Datendisplay sehen wir den Inhalt des Prozessor Status Registers. Den Aufbau dieses Registers haben wir bereits bei der Beschreibung des Programmiermodells der 6502-CPU besprochen:



Jedem Bit in diesem Register ist eine Flag zugeordnet. Die "X" deuten an, daß diese Flags belanglos sind. Nur dem C-Flag gilt unser Interesse. Bit Ø ist diesem Flag zugeordnet. Da der Inhalt der Speicherzelle ØØF1 nach dem CLC-Befehl eine gerade Zahl ist, muß die C-Flag Null sein. Wäre Carry nicht Null, dann wäre in der Speicherzelle ØØF1 eine ungerade Zahl. Somit ist festgestellt werden, daß der CLC-Befehl tatsächlich ausgeführt wurde.
Jetzt soll der Computer den folgenden Befehl im Additionsprogramm abarbeiten. Dazu müssen wir wieder ins Additionsprogramm zurückkehren. Durch Drücken auf die PC-Taste ist das leicht möglich. PC steht für Programm Counter oder Programmzähler. Nach Drücken dieser Taste erscheint im Display des Junior-Computers der folgende Befehl mit seiner Adresse, der nun abgearbeitet werden soll. Es wäre jedoch auch möglich gewesen, auf die AD-Taste zu drücken und den Junior-Computer durch Eingabe einer Adresse an die Stelle des Additionsprogrammes zu führen, an der der folgende Befehl ausgeführt werden soll.
Nach dem Drücken auf die PC-Taste erscheint also auf dem Display Ø1Ø7 AD. AD ist der OP-Code von LDA-. Betätigen wir nochmals die GO-Taste, dann erscheint auf dem Display Ø1ØA 6D. 6D ist der OP-Code der ADC-Befehles, der dem LDA-Befehl folgt. Die Instruktion LDA-LOB1 ist bereits ausgeführt. Wir erinnern uns, daß LOB1 = EF ist. Im Accu der CPU muß also EF stehen, wenn der LDA-Befehl ausgeführt ist. Wie wir bereits wissen, ist der Inhalt des Accus an der Adresse ØØF3 abgelegt. An dieser Adresse sehen wir nach und sehen auf dem Display: ØØF3 EF. Der Accu wurde also tatsächlich mit EF geladen. Durch Drücken der PC-Taste können wir wieder ins Additionsprogramm zurückkehren und den folgenden Befehl ausführen.
Die "step by step mode" ist ein wirkungsvolles Werkzeug, Programme zu testen und Fehler aufzuspüren. Besonders für den Anfänger, der häufig Programmierfehler beim Erstellen eigener Programme macht, ist diese

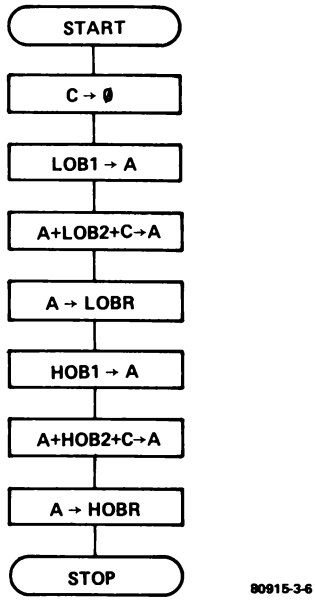


Bild 6. Das grobe (globale) Flußdiagramm des Additionsprogramms von Bild 4.

Programmierungsart des Junior-Computers unentbehrlich. Beim Erstellen eigener Programme sollte man wie folgt vorgehen:

1. Grobe Flowchart (Flußdiagramm) zeichnen (z.B. Bild 6).
2. Detaillierte Flowchart mit OP-Codes anfertigen, wie es in Bild 4 dargestellt ist.
3. Die detaillierte Flowchart mit dem Tastenfeld in den Computer eingeben.

Zero Page Addressing

Eine Zero Page Adresse ist eine Adresse innerhalb der Seite Null im Speicher des Junior-Computers. Zero Page Addressing ist eine besondere Form von Absolute Addressing. Instruktionen mit absoluter Adressierung bestanden aus drei Bytes: OP-Code – Niederwertiges Adreßbyte ADL – hochwertiges Adreßbyte ADH. Instruktionen mit Zero Page Addressing unterscheiden sich dadurch, daß das hochwertige Adreßbyte ADH immer Null ist. Der Adreßbereich für Instruktionen dieser Adressierungsart ist also: ØØØØ . . . ØØFF. Diese 256 Adressen bilden die Seite Null (Zero Page) des Junior-Computers. Die folgenden 256 Adressen gehören zur Seite 1, die der Seite Ø folgt. Die Adressen in Seite 1 laufen von ØlØØ . . . Ø1FF. Das geht so weiter bis zur Seite FF, deren Adressen von FFØØ . . . FFFF laufen. Somit setzt sich der Adreßbereich des Junior-Computers aus 256 Seiten zusammen, von denen jede 256 Bytes enthält. Das kommt auch mit der Rechnung überein: $256^2 = 65536$ adressierbare Speicherzellen. Die

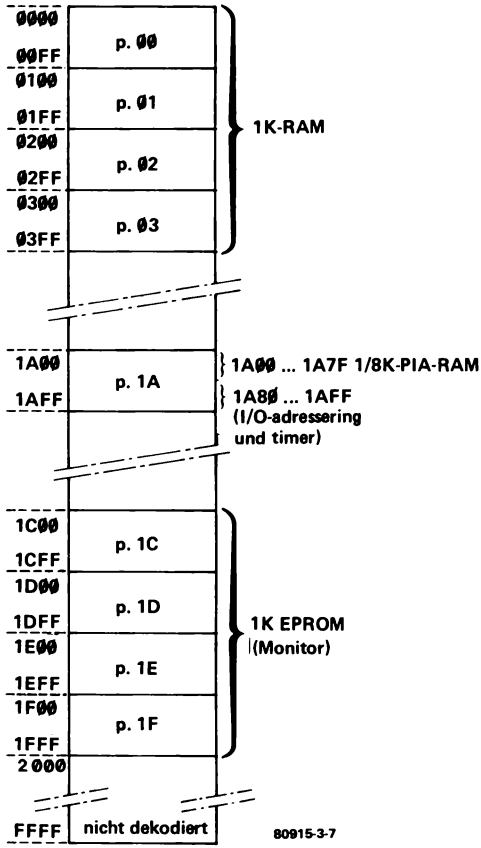


Bild 7. Der Speicher der CPU 6502 ist in 256 Seiten aufgeteilt. Jede Seite ist 256 Bytes lang. Das hochwertige Adreßbyte gibt die Seitennummer an (ØØ . . . FF).

Seitenstruktur ist in Bild 7 dargestellt. Seite Ø bis Seite 3 ist der RAM-Speicher auf der Basisplatine des Junior-Computers. In diesen vier Seiten legen wir die Daten ab, wenn wir mit dem Keyboard Programme eingeben. Die Seite 1A ist den PIA zugeordnet, der auch noch ein 128 Byte umfassendes internes RAM, einen Timer und andere Register hat. In den Seiten 1C . . . 1F ist das Monitorprogramm des Junior-Computers abgelegt. In der Standardausführung ist der gesamte Adreßbereich nicht vollständig dekodiert. (siehe Adreßdekodierung im 1. Kapitel). Wegen der nicht vollständigen Adreßdekodierung ist die höchste Adresse des Junior-Computers 1FFF.
Aber zurück zu den Adressierungsarten. Da die CPU automatisch weiß, daß bei einem Befehl mit Page Zero Addressing das hochwertige Adreßbyte ADH immer ØØ ist, müssen wir dieses Adreßbyte nicht in den Computer eingegeben. Befehle mit Page Zero Addressing sind also nur zwei

Bytes lang: OP-Code – ADL. Dem OP-Code folgt das niederwertige Adreßbyte. Da das hochwertige Adreßbyte ADH bei diesen Instruktionen entfällt bzw. die CPU automatisch ØØ setzt, läßt sich eine Menge Speicherplätze einsparen: 33% gegenüber Absolute Addressing. Befehle mit Page Zero Addressing sind durch ein "Z" gekennzeichnet. Eine Befehlsübersicht ist in der folgenden Tabelle zusammengestellt:

Lade- und Schreibbefehle:
LDAZ Code A5 (M→A) load accumulator with memory
LDXZ Code A6 (M→X) load index X with memory
LDYZ Code A4 (M→Y) load index with memory
STAZ Code 85 (A→M) store accumulator in memory
STXZ Code 86 (X→M) store index X in memory
STYZ Code 84 (Y→M) store index Y in memory

Arithmetische Befehle:
ADCZ Code 65 (A+M+C→A) add memory to accumulator with carry
SBCZ Code E5 (A−M−$\overline{C}$→A) subtract memory from accu w. carry
INCZ Code E6 (M+1→M) increment memory by one
DECZ Code C6 (M−1→M) decrement memory by one

Logische Befehle:
ORAZ Code Ø5 (A ∨ M→A) OR memory with accumulator
ANDZ Code 25 (A ∧ M→A) AND memory with accumulator
EORZ Code 45 (A ⊻ M→A) Exclusive OR memory with accumulator

An anderen Stellen in diesem Kapitel ist noch ausreichend Gelegenheit, sich mit diesen Instruktionen vertraut zu machen. Da Zero Page Addressing sehr ähnlich den Absolute Addressing ist, sehen wir hier von einem Übungsprogramm ab und wenden uns sofort einem neuen Adressierungsverfahren zu.

Relative Addressing

Relative Addressing ist das Adressierungsverfahren der bedingten Sprünge. Bedingte Sprünge werden im englischen Sprachraum als Branch-Instruktionen bezeichnet (branch = verzweigen). Die 6502-CPU kennt zwei Arten von Branch-Instruktionen: bedingte und nicht bedingte Verzweigungen. Bei bedingten Sprüngen verzweigt das Programm nur, wenn bestimmte Voraussetzungen erfüllt sind: zum Beispiel das Ergebnis einer arithmetischen Operation Null oder ungleich Null ist, das C-Flag gesetzt oder nicht gesetzt ist usw. Es ist somit leicht erkennbar, daß bedingte Sprünge im Programm mit den Flags im Prozessor Status Register zusammenhängen. Wie diese bedingten Verzweigungen in einer Flowchart aussehen, haben wir bereits im 2. Kapitel gezeigt. Im Gegensatz zu den bedingten Sprungbefehlen wird bei den nichtbedingten stets gesprungen. Beide Arten von Sprunginstruktionen wollen wir nun unter die Lupe nehmen. Zuerst die bedingten Sprunginstruktionen mit Relative Addressing. Die 6502-CPU hat folgende bedingte Sprungbefehle:

1. BCC opcode 9Ø Branch on Carry Clear. Springe wenn C=Ø
   BCS opcode BØ Branch on Carry Set. Springe wenn C=1
2. BNE opcode DØ Branch Not Equal. Springe wenn Z=Ø
   BEQ opcode FØ Branch on EQual. Springe wenn Z=1

3. BPL opcode 1Ø Branch on PLus. Springe wenn N=Ø
   BMI opcode 3Ø Branch on MInus. Springe wenn N=1
4. BVC opcode 5Ø Branch on oVerflow Clear. Springe wenn V=Ø
   BVS opcode 7Ø Branch on oVerflow Set. Springe wenn V=1

Mit diesen Instruktionen wollen wir nun nacheinander Programme schreiben. Beginnen wir mit dem Programm in Bild 8. Bei der Adresse ØØØØ wird das Y-Register mit ØA geladen, und anschließend mit einer DEY-Instruktion um eins vermindert. Der neue Inhalt des Y-Registers ist dann Ø9. Anschließend trifft die CPU auf den BNE-Befehl. Mit ihm wird gefragt, ob das Y-Register schon Null ist. Ist es nicht Null, so springt der Mikrocomputer nach Start1 und dekrementiert das Y-Register abermals um Eins. Das Programm macht also nicht anderes, als das Y-Register solange mit Eins zu vermindern, bis es schließlich Null ist. Ist diese Bedingung eingetreten, dann ist die Z-Flag gesetzt und die CPU verläßt die Programmschleife. Der folgende BRK-Befehl stoppt das Programm.
In Bild 8 steht unter dem BNE-Befehl (DØ) die hexadezimale Zahl FD. Diese Zahl gibt an, um wieviele Speicherzellen das Programm zurückspringen muß, um auf den DEY-Befehl zu stoßen. Sprünge nach vorwärts sind positiv und nach rückwärts negativ.
FD ist 11111ØØ1 und das ist das Zweierkomplement der Zahl –3. Das stimmt genau mit der Schrittweite überein, die der Computer im Programm zurückspringen soll. Warum? Direkt nach dem Abarbeiten des BNE-Befehls steht der Programmzähler PC auf der Adresse Ø2Ø5. Von dieser Adresse muß nun ein Offset (Rücksprung) von –3 erfolgen, um nach Start 1 zu gelangen. Unter Offset ist dabei die Schrittweite zu verstehen, mit der die CPU verzweigen soll. Der Offsetbereich variiert zwischen 127 Schritten nach vorwärts (Ø . . . +127 = ØØ . . . 7F) und maximal 128 Schritten nach rückwärts (–1 . . . –128 = FF . . . 8Ø). Bei der Berechnung eines Offsets müssen wir von der Adresse ausgehen, auf die der Programmzähler zeigt, wenn der Verzweigungsbefehl abgearbeitet ist. An dieser

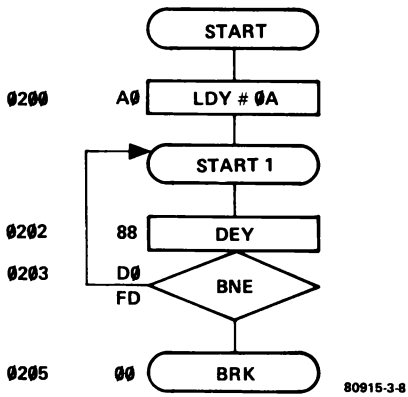


Bild 8. Flußdiagramm eines Programms mit einer bedingten Sprunginstruktion. Die bedingte Sprunginstruktion ist BNE mit dem OP-Code DØ, das Offset ist FD (drei Schritte zurück).

Adresse steht immer der Befehl, der dem Verzweigungsbefehl folgt. Um das zu verdeutlichen, schreiben wir das Programm von Bild 8 an, wie es im Speicher des Junior-Computers abgelegt ist:

```
0200  A0   LDY #
0201  0A
0202  88   DEY
0203  D0   BNE
0204  FD        Anzahl der Rücksprünge
0205  00   BRK
```

Nebenbei lernten wir noch den Befehl DEY kennen. Diese 1 Byte Instruktion vermindert den Inhalt des Y-Registers um Eins. Später gehen wir noch ausführlich auf diese 1 Byte Befehle ein.

Offset-Berechnung mit dem Monitor

Das Berechnen von Offsets ist ohne Hilfsmittel zeitraubend und nicht ganz einfach. Deshalb ist im Monitorprogramm des Junior-Computers eine Routine untergebracht, die selbstständig Offsets berechnet. Mit dem Tastenfeld muß nur die Adresse eingegeben werden, an der der Sprung beginnt, und anschließend die Adresse, zu der gesprungen werden soll. Das Offset berechnet dann der Computer. Zeichnet man eine Flowchart wie in Bild 8 dargestellt, dann sind die Start- und die Endadresse der Programmverzweigung bekannt. Da nur +127 Schritte nach vorwärts und −128 Schritte nach rückwärts verzweigt werden können, müssen nur die niederwertigen Adreßbytes der Start- und Endadresse eingegeben werden. Die Routine im Monitor zur Berechnung der Offsets heißt BRANCH und beginnt an der Adresse 1FD5. Ist diese Adresse eingegeben, dann läßt sich das Programm mit der GO-Taste starten. Nach dem Start zeigt das Display 0000000 und das Programm wartet nun auf die Eingabe der Start- und Endadresse der Verzweigung. Der BNE-Befehl in Bild 8 steht an der Adresse 0203, an der Startadresse der Verzweigung. Das niederwertige Adreßbyte von 0203 ist 03. Von dieser Adresse soll die Verzweigung nach 0202 führen, der Endadresse des bedingten Sprunges. Das niederwertige Adreßbyte von 0202 ist 02. Diese beiden niederwertigen Adreßbytes geben wir nun in den Computer ein und erhalten im Datendisplay das Offset der Verzweigung:

```
AD                XXXX XX
 1   F   D   5    1FD5 D8
GO                0000 00
 0   3   0   2    0302 FD  notieren
RST
```

Mit der Routine BRANCH lassen sich auch mehrere Offsets hintereinander berechnen. Vor jeder Berechnung läßt sich das Display durch Drücken einer beliebigen Kommandotaste auf 000000 zurücksetzen. Dieses Zurücksetzen ist auch möglich, wenn man sich bei der Dateneingabe vertippt hat. Durch das Drücken der RST-Taste kann das Programm

wieder verlassen werden und der Junior-Computer steht bereit, für die Eingabe des Programms von Bild 8:

```
AD
 Ø 2     Ø    Ø      Ø2ØØ  XX
DA       A    Ø      Ø2ØØ  AØ   LDY #
 +       Ø    A      Ø2Ø1  ØA
 +       8    8      Ø2Ø2  88   DEY
 +       D    Ø      Ø2Ø3  DØ   BNE
 +       F    D      Ø2Ø4  FD   Offset
 +       Ø    Ø      Ø2Ø5  ØØ   BRK
```

Auf den ersten Blick scheint dieses Programm zwecklos zu sein. Das ist aber nicht richtig! Dieses Programm läßt sich dazu verwenden, um definierte Zeitverzögerungen zu erzeugen. Wie wir bereits wissen, benötigt der Computer für das Abarbeiten eines Befehls eine bestimmte Zeit (siehe Instruction Code Tabelle am Ende dieses Buches). Somit lassen sich durch Programmschleifen Zeitverzögerungen beliebiger Längen herstellen.

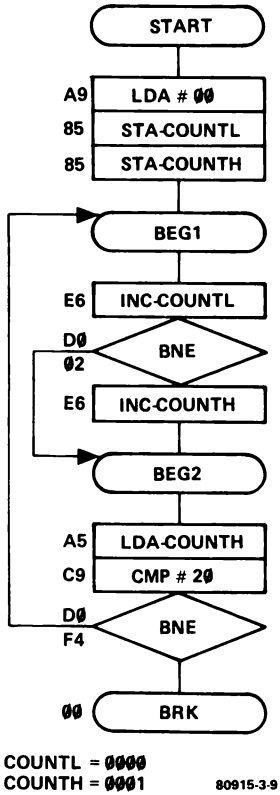


Bild 9. Flußdiagramm eines Software Zählers. Der Zähler zählt von ØØØØ . . . 2ØØØ. Beim Erreichen des höchsten Zählerstandes stoppt der Zählvorgang.

Ein Softwarezähler

Um uns weiter mit den bedingten Sprunginstruktionen vertraut zu machen, schreiben wir ein Programm, mit dem sich ein Softwarezähler realisieren läßt. Softwarezähler deshalb, da weder Lötkolben noch ICs für diesen Zähler benötigt werden. Bei diesem Programm lernen wir einen neuen Befehl kennen, den CMP-Befehl oder Vergleichsbefehl.
Der Softwarezähler kann von ØØØØ . . . 2ØØØ zählen (hexadezimal). Ist der maximale Zählerstand erreicht, stoppt der Zählvorgang. Die Flowchart des Zählers zeigt Bild 9. Um die hexadezimale Zahl darzustellen, benötigt man zwei Bytes. Zwei Speicherzellen müssen also für den Zähler reserviert sein. Auf der Adresse ØØØ1 steht das hochwertige Zählerbyte COUNTH und auf ØØØØ das niederwertige COUNTL. Am Anfang setzt das Programm diese beiden Zählerzellen auf Null. Dann wird COUNTL um Eins erhöht, und mit dem folgenden BNE-Befehl geprüft, ob ein Übertrag auf COUNTH erfolgen soll. Ein Übertrag soll stattfinden, wenn die Z-Flag gesetzt ist. Ist der Inhalt von COUNTL ungleich Null, dann verzweigt das Programm nach BEG2. Der Accu wird nun mit dem Inhalt von COUNTH geladen und mit dem CMP-Befehl mit 2Ø verglichen. Somit läßt sich prüfen, ob der Zählvorgang gestoppt oder fortgesetzt werden soll. Der CMP-Befehl beeinflußt die Z-Flag im Prozessor Status Register. Dieses Flag ist Eins, wenn der Accuinhalt 2Ø ist, und der folgende BRK-Befehl stoppt das Programm. Mit anderen Worten: COUNTL wird nach 256 Inkrementierungen ØØ. Das hat jedesmal eine Inkrementierung von COUNTH zur Folge. COUNTH wird dabei solange inkrementiert, bis der Inhalt dieser Speicherzelle 2Ø ist. Die Eingabe des Softwarezähler-Programms in den Junior-Computer sieht so aus:

AD					xxxx		xx	
1	F	D	5		1FD5		D8	Startadresse von BRANCH
GO					ØØØØ		ØØ	
1	8	1	C		181C		Ø2	Offset 1. BNE
2	Ø	1	6		2Ø16		F4	Offset 2. BNE
RST	AD							
Ø	2	1	Ø		ØØ21Ø		xx	Startadresse des Zählers
DA		A	9		ØØ21Ø		A9	LDA #
+		Ø	Ø		ØØ211		ØØ	
+		8	5		ØØ212		85	STA-
+		Ø	Ø		ØØ213		ØØ	
+		8	5		ØØ214		85	STA-
+		Ø	1		ØØ215		Ø1	
+		E	6		ØØ216		E6	INC-
+		Ø	Ø	12	ØØ217		ØØ	
+		D	Ø	11	ØØ218		DØ	BNE
+		Ø	2	10	ØØ219		Ø2	Offset
+		E	6	9	ØØ21A		E6	INC-
+		Ø	1	8	ØØ21B	1	Ø1	

+		A	5	7	Ø21C	2	A5	LDA-
+		Ø	1	6	Ø21D		Ø1	
+		C	9	5	Ø21E		C9	CMP #
+		2	Ø	4	Ø21F		2Ø	Operand CMP #
+		D	Ø	3	Ø22Ø		DØ	BNE
+		F	4	2	Ø221		F4	Offset
+		Ø	Ø	1	Ø222		ØØ	BRK
AD					Ø222		xx	
Ø	2	1	Ø		Ø21Ø		A9	Startadresse
GO					Ø212		xx	Programmstart

Bei diesem Tastenprogramm sind auch die Offsets der Verzweigungen eingezeichnet. Nach den 1. BNE-Befehl verzweigt das Programm um +2 Schritte vorwärts und nach dem 2. BNE-Befehl um −12 Schritte rückwärts. Die Offsets zu beiden Verzweigungen sind Ø2 und F4.

Der CMP-Befehl

Mit dem CMP-Befehl (CMP = CoMPare) lassen sich zwei Daten im Speicher des Junior-Computers miteinander vergleichen. CMP hat mehrere Adressierungsarten. Im Zählerprogramm machten wir zweimal von Immediate Addressing Gebrauch. CMP # hat den Op-Code C9. Der CMP-Befehl führt eine herkömmliche Subtraktion aus, ohne Berücksichtigung der C-Flag. Die Operationsweise dieses Befehles ist: A−M. CMP beeinflußt nur die Flags im Prozessor Status Register, jedoch nicht den Inhalt des Accus. Das Ergebnis der Subtraktion beeinflußt folgende drei Flags:

a) N-Flag = 1, wenn A < M, Verzweigungsbefehl, wenn ja: BMI
b) Z-Flag = 1, wenn A = M, Verzweigungsbefehl, wenn ja: BEQ
c) C-Flag = 1, wenn A ≥ M, Verzweigungsbefehl, wenn ja: BCS

Diese mathematischen Zusammenhänge lassen sich auch in Worte fassen:

a) Mit einem CMP-Befehl läßt sich feststellen, ob der Accuinhalt kleiner ist, als der Inhalt einer bestimmten Speicherzelle. Ein anschließender BMI-Befehl bestimmt, ob die Forderung "<" zutrifft oder nicht.
b) Mit einem CMP-Befehl läßt sich feststellen, ob der Accuinhalt dem Inhalt einer bestimmten Speicherzelle gleich ist. Ein anschließender BEQ-Befehl bestimmt, ob die Forderung "=" zutrifft oder nicht.
c) Mit einem CMP-Befehl läßt sich feststellen, ob der Accuinhalt größer oder gleich dem Inhalt einer bestimmten Speicherzelle ist. Ein anschließender BCS-Befehl bestimmt, ob die Forderung "≥" zutrifft oder nicht.

Somit lernten wir einige bedingte Sprunginstruktionen und den CMP-Befehl kennen. Verzweigungen im Zusammenhang mit dem CMP-Befehl ermöglichen es, Programme einfach und übersichtlich zu schreiben. Später kommen wir noch häufig darauf zurück, doch zunächst sollen die nichtbedingten Sprungbefehle erklärt werden.

Nichtbedingte Sprungbefehle

Nichtbedingte Sprungbefehle sind im Gegensatz zu den bedingten Sprungbefehlen weder von den Flags im Prozessor Status Register noch von

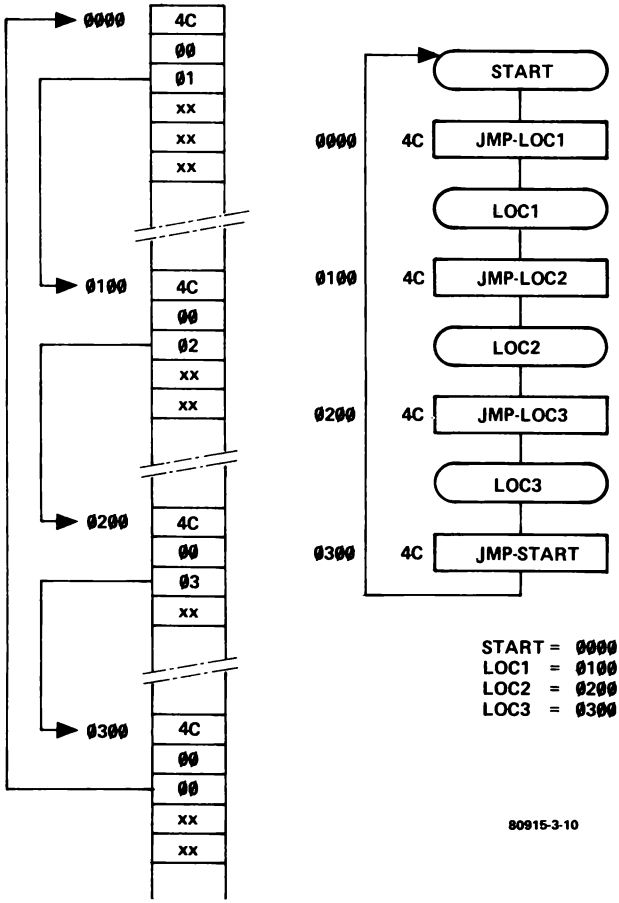


Bild 10. Eine Programmschleife, die aus vier nichtbedingten Sprüngen besteht (unendliche Programmschleife).

anderen Umständen abhängig. Trifft also der Mikroprozessor auf einen bedingten Sprungbefehl, so springt er zur nächsten Speicherzelle, in der wiederum ein Befehl abgelegt ist. Die 6502-CPU kennt zwei nichtbedingte Sprungbefehle: JMP- und JMP-(indirekt). Der direkte oder absolute Sprungbefehl hat den OP-Code 4C und der indirekte Sprungbefehl den OP-Code 6C. Zunächst erklären wir den direkten oder absoluten Sprungbefehl. Dieser Befehl ist drei Bytes lang und führt den Prozessor von einer absoluten Adresse zu einer anderen. Die Funktionsweise dieses Sprungbefehles wird klar, wenn wir Bild 10 betrachten. Dort ist ein Programm dargestellt, das ausschließlich aus Sprunginstruktionen besteht. Das Programm springt von einer Adresse zu einer anderen und schließlich wieder zum Startpunkt zurück. Die JMP-Instruktion ist wie folgt aufgebaut:

OP-Code – ADL – ADH. Den indirekten Programmsprung erklären wir noch bei einer weiteren Adressierungsart der 6502-CPU, dem Indirect Addressing.
JMP-Instruktionen verwendet man häufig, wenn bei einem Mikrocomputer die Pages im Speicher nicht lückenlos aneinander liegen (vergl. Bild 7). Somit läßt sich leicht von einer Speicherseite in eine andere springen. Manchmal sind auch in einem Programm Verzweigungen von mehr als +127 oder –128 Schritten erforderlich. Dann verzweigt das Programm zu einer JMP-Instruktion, die sich in der Nähe des Verzweigungsbefehles befindet. Somit sind Verzweigungen über den gesamten adressierbaren Speicherbereich möglich. Bei der Besprechung des Systemmonitors im 2. Buch des Junior-Computers treffen wir öfters auf Verzweigungen, die weit über den Branch-Bereich der 6502-CPU hinausführen.

JSR und RTS: Springe in ein Unterprogramm und kehre danach wieder ins Hauptprogramm zurück

Angenommen, ein Schüler hat ein schwieriges mathematisches Problem zu lösen. Damit er dieses Problem lösen kann, gab ihm der Lehrer ein "Kochrezept", das Punkt für Punkt beschreibt, wie bei der mathematischen Aufgabe vorzugehen ist. Immer, wenn der Schüler ein ähnliches mathematisches Problem lösen muß, wird er auf dieses Kochrezept zurückgreifen. Die Verfahrensweise, die zur Lösung führt, besteht aus drei Einzelschritten:
1) Der Schüler will ein mathematisches Problem lösen (Hauptproblem)
2) Um es zu lösen, sieht er in einem Kochrezept nach, wie dabei vorzugehen ist (Teilproblem)
3) Nachdem die Verfahrenweise, die zur Lösung führt, klar ist, kann sich der Schüler wieder um das Hauptproblem, die Lösung seiner Aufgabe kümmern (Rückkehr vom Teilproblem zum Hauptproblem)

Genauso ist es beim Mikrocomputer. Beim Durchlaufen eines Hauptprogramms muß häufig eine Sequenz von Instruktionen mehrmals durchlaufen werden. Diese Sequenz von Instruktionen, die ein Teil des Hauptprogramms sind, legt man irgendwo im Speicher des Mikrocomputers ab und ruft sie auf, wenn diese für eine Operation gebraucht werden. Der Programm-Ablaufplan für den Computer sieht dann wie folgt aus:
1) Im Hauptprogramm soll eine Aufgabe gelöst werden.
2) Springe in ein Unterprogramm (JSR = Jump to SubRoutine) in dem ein Teil der Aufgabe gelöst wird (= Kochrezept beim Schüler).
3) Kehre danach wieder ins Hauptprogramm zurück (RTS = ReTurn from Subroutine) um mit einer Teillösung der Aufgabe weiterzuarbeiten.

Ein Unterprogramm oder eine Subroutine ist also immer ein Teil des Hauptprogramms. Mit dem JSR-Befehl läßt sich ein und dieselbe Subroutine beliebig oft aufrufen. Ein RTS-Befehl am Ende der Subroutine führt den Prozessor wieder zurück ins Hauptprogramm. Somit ist verständlich, das sich mit einer Subroutine viele Speicherplätze einsparen lassen. Wieviele Speicherplätze, das hängt davon ab, wie häufig die Subroutine während des Hauptprogramms aufgerufen wird.
Bild 11 zeigt den Sprung zu einer Subroutine und die anschließende Rückkehr ins Hauptprogramm. Die Subroutine belegt dabei die Speicherplätze 1A21 . . . 1A3B. Die Adresse 1A21 beinhaltet den ersten OP-Code der Subroutine und bei der Adresse 1A3B steht der Befehl RTS, der den

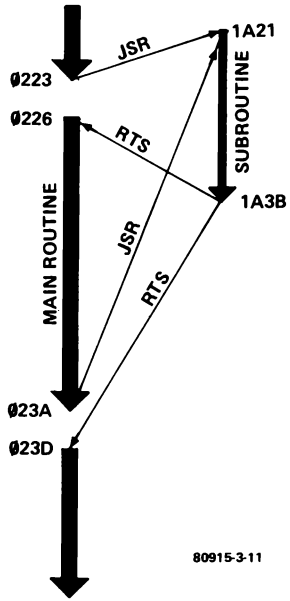


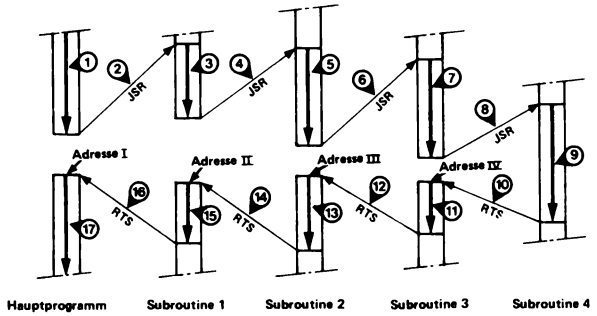
Bild 11. Das Hauptprogramm verzweigt zweimal zu einer Subroutine (JSR). Der Befehl RTS am Ende der Subroutine führt den Prozessor wieder ins Hauptprogramm zurück.

Prozessor wieder ins Hauptprogramm zurückführt. Diese Subroutine wird vom Hauptprogramm zweimal aufgerufen. Die Adresse Ø223 hat den OP-Code von JSR. Die beiden folgenden Adressen Ø224 (ADL) sowie Ø225 (ADH) zeigen auf die Startadresse der Subroutine (1A21 = ADH, ADL). Damit der Computer nach dem Durchlaufen der Subroutine weiß, zu welcher Adresse er ins Hauptprogramm zurückkehren soll, muß diese Rückkehradresse irgendwo im Speicher abgelegt werden. Der JSR-Befehl erledigt das automatisch, indem er sich des Stacks (Stack = Stapel) bedient, das heißt, die Rückkehradresse wird auf dem Stack abgelegt. Den Stack besprechen wir anschließend noch genau.
Nach der Rückkehr von der Subroutine läuft das Hauptprogramm ab der Adresse Ø226 in der gewohnten Weise ab. Bei der Adresse Ø23A trifft der Prozessor wieder auf einen JSR-Befehl, der ihn abermals zu derselben Subroutine führt, die anfangs schon einmal durchlaufen wurde. Am Schluß der Subroutine steht wieder ein RTS-Befehl, der den Prozessor wieder an die richtige Stelle ins Hauptprogramm zurückführt. Die JSR-Instruktion ist drei Bytes lang und hat Absolute Addressing. RTS ist die Umkehrung der JSR-Instruktion und nur ein Byte lang.

Der Stack und sein Pointer

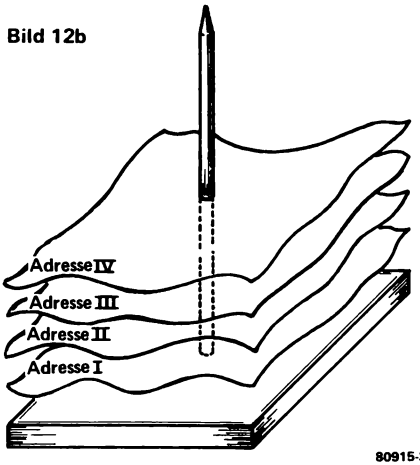
Subroutinen haben den Vorteil, daß sich ein Hauptprogramm sehr übersichtlich schreiben läßt: Eine große oder kleine Anzahl von Instruktionen wird in einer Subroutine zusammengefaßt und vom Hauptprogramm bei

Bild 12a



80915-3-12a

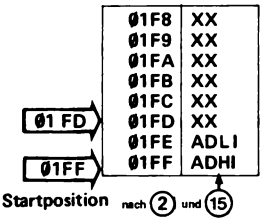
Bild 12b

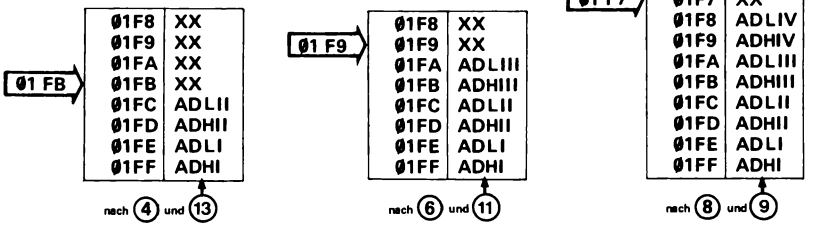


80915-3-12b

Notizensammler

Bild 12c





80915-3-12c

Bild 12. Vier Subroutinen sind hier ineinander verschachtelt (a). Bei Sprüngen in Subroutinen findet der Stack Verwendung. Dieses Register in Seite 1 läßt sich mit einem Notizensammler vergleichen. (b). Wie bei den einzelnen Sprüngen in die Subroutinen die Rückkehradressen auf dem Stack abgelegt werden zeigt Bild 12c.

Bedarf mit einem JSR-Befehl aufgerufen. Es ist jedoch auch möglich, innerhalb einer Subroutine andere Subroutinen aufzurufen. Das heißt, Subroutinen lassen sich ineinander verschachteln. Bild 12a zeigt den Ablaufplan von ineinander verschachtelten Subroutinen.
Zu einem jeden Sprung in einer Subroutine gehört eine Sprung- und eine Rückkehradresse. Die Sprungadresse ist die Startadresse der Subroutine. Die Rückkehradresse führt den Prozessor wieder an die richtige Stelle ins Haupt- oder Unterprogramm zurück. Die Sprungadressen kann der Programmierer selbst bestimmen oder, wie in dem Buch "Junior-Computer 2" gezeigt, selbst berechnen. Die Rückkehradressen verwaltet der Computer mit Hilfe des Stacks. Das ist bei der 6502-CPU ein Stück Speicher in Page 1. Bei einem Sprung in eine Subroutine legt die CPU auf dem Stack die Rückkehradressen ab. Eine Rückkehradresse besteht aus zwei Bytes, dem nieder- und dem hochwertigen Adreßbyte der Rückkehradresse. Die Steuerung des Stacks übernimmt der Stackpointer. Das ist ein Zeiger, der immer auf eine Stackzelle zeigt, in der die folgende Rückkehradresse abgelegt werden soll. Werden wie in Bild 12a mehrere Subroutinen ineinander verschachtelt, dann wächst der Stack von unten nach oben. Das heißt, je mehr der Stack wächst, desto niedriger werden die Page 1-Adressen, an denen die Rückkehradressen abgelegt sind.
In welcher Reihenfolge arbeitet der Prozessor die Subroutinen ab? Betrachtet man nochmals Bild 12a. Dort wird das Hauptprogramm verlassen und Subroutine 1 aufgerufen. Beim Sprung in die Subroutine 1 wird die Rückkehradresse (Adresse I) auf dem Stapel gelegt. Der Stackpointer wird jetzt vom Prozessor so eingerichtet, daß er auf eine Stackzelle zeigt, in der wiederum ein Byte der Rückkehradresse abgelegt werden kann.

In der Subroutine 1 erfolgt ein Sprung in die Subroutine 2. Auch bei diesem Sprung in eine Subroutine wird die Rückkehradresse auf dem Stack abgelegt. Das läuft so weiter, bis schließlich die vierte Subroutine aufgerufen wird. Auch bei diesem Sprung legt der Prozessor die Rückkehradresse (Adresse IV) auf dem Stack ab. Vier Subroutinen sind nun nacheinander aufgerufen worden, ohne daß die CPU auf einen RTS-Befehl gestoßen ist! Am Ende der Subroutine 4 steht der erste RTS-Befehl, der den Prozessor zurück in die Subroutine 3 führt. Die Rückkehradresse (Adresse IV) holt sich die CPU vom Stack. Derselbe Vorgang ereignet sich am Ende von Subroutine 3. Auch dort steht ein RTS-Befehl, der den Prozessor zurück in die Subroutine 2 führt. Die Rückkehradresse (Adresse IV) holt sich die CPU vom Stack. Schließlich führt die letzte Rückkehradresse (Adresse I) den Prozessor wieder ins Hauptprogramm zurück. Was haben wir aus diesem Beispiel gelernt? Ganz einfach: Werden mehrere Subroutinen ineinander verschachtelt, so muß für jede Subroutine eine Rückkehradresse auf dem Stack abgelegt werden.
Der Stack läßt sich sehr schön mit einem Notizensammler vergleichen (Bild 12b). Notizzettel sind dort übereinander gestapelt. Auf jedem Notizzettel steht eine Rückkehradresse. Die Rückkehradresse der Subroutine, die zuletzt aufgerufen wurde, liegt ganz oben auf dem Notizensammler. Analog dazu läßt sich auch der Stack beschreiben (Bild 12c). Die Steuerung des Stacks erledigt der Stack Pointer. Dieser Pointer zeigt immer auf eine Adresse in der Page (Seite), an der ein Adreßbyte der folgenden

Rückkehradresse abgelegt werden soll. Nehmen wir an, zu Beginn ① zeigt der Stack Pointer auf die Adresse Ø1FF. Das ist die letzte Adresse in Page 1. Dann kommt der Aufruf einer Subroutine 2. Bevor der Prozessor in die Subroutine springt, rettet er die Rückkehradresse ADHI, ADLI auf den Stack. Der Stack Pointer zeigt jetzt auf die Adresse Ø1FD. Trifft der Prozessor in der Subroutine 1 ③ auf einen weiteren JSR-Befehl, so rettet er wieder die Rückkehradresse der Subroutine 1 auf den Stack ④. Die Rückkehradresse für Subroutine 1 ist ADHII, ADLII. Auch der Stack Pointer wird neu eingerichtet und zeigt auf die Adresse Ø1FB, bevor der Prozessor in die Subroutine 2 springt. Das läuft so weiter, bis schließlich der letzte JSR-Befehl an der Reihe ist ⑧. Bevor der Prozessor in die Subroutine 4 springt, rettet er wieder die Rückkehradresse auf den Stack. Der Stack Pointer zeigt jetzt auf die Adresse Ø1F7. Da jede Rückkehradresse aus zwei Adreßbytes besteht, einem hoch- und einem niederwertigen Adreßbyte, ist der Stack im vorliegenden Beispiel auf acht Speicherzellen angewachsen.
Am Ende der Subroutine 4 trifft die CPU auf einen RTS-Befehl, der sie in die Subroutine 3 zurückführt ⑩. Zu welcher Adresse? Die beiden Adreßbytes der Rückkehradresse holt sich der Prozessor vom Stack (ADHIV, ADLIV). Auch der Stack Pointer wird neu eingerichtet und zeigt auf die Adresse Ø1F7. Die RTS-Befehle am Schluß der Subroutinen führen den Prozessor wieder zurück ins Hauptprogramm. Auch der Stack Pointer zeigt wieder auf die Startposition: Ø1FF.
Zusammenfassend läßt sich also sagen, der Stack ist ein Notizensammler (Bild 12b) für die Rückkehradressen von Subroutinen. Die Rückkehradresse der Subroutine, zu der zuletzt gesprungen wird, liegt ganz oben auf dem Stack, während die Rückkehradresse der ersten Subroutine unten ist. Der Stack wächst also von unten nach oben und wird in der umgekehrten Reihenfolge wieder abgebaut. Im englischen Sprachraum bezeichnet man den Stack als LIFO (LIFO = Last In First Out). Das heißt, was zuletzt auf den Stack kam, kommt auch zuerst wieder herunter.
Der Mechanismus des Stacks mit seinem Pointer ist nicht ganz einfach zu verstehen. Deshalb wollen wir uns an Hand eines praktischen Beispiels damit vertraut machen. In Bild 13 wird von einer Hauptroutine (Main Routine) in eine Subroutine gesprungen. Bei der Adresse Ø216 trifft der Prozessor auf einen JSR-Befehl (OP-Code = 2Ø). Springen zu welcher Adresse? Auf den Adressen Ø217 und Ø218 steht die Startadresse der Subroutine: Ø3ØØ. Bei der Adresse Ø217 steht das niederwertige Adreßbyte ØØ und bei der Adresse Ø218 das hochwertige Byte Ø3. Wie bereits bekannt, rettet der Prozessor bei einem JSR-Befehl die Rückkehradresse auf den Stapel. Die Startposition des Stack Pointers ist wieder Ø1FF. Bei dieser Adresse wird das hochwertige Adreßbyte der Rückkehradresse abgelegt und bei der folgenden das niederwertige (Ø1FF – Ø2, 01FE – 18). Die Rückkehradresse auf dem Stack ist also Ø218. Sobald die Rückkehradresse auf dem Stack gerettet ist, verzweigt der Prozessor in die Subroutine, die an der Adresse Ø3ØØ beginnt. An dieser Adresse steht die erste Instruktion der Subroutine. An der Adresse Ø3Ø6 trifft der Prozessor wieder auf einen RTS-Befehl (OP-Code 6Ø). Wohin zurückgesprungen wird, steht auf dem Stack: Ø218. An dieser Stelle steht jedoch kein OP-Code, sondern das hochwertige Adreßbyte des JSR-Befehles. Keine Bange!

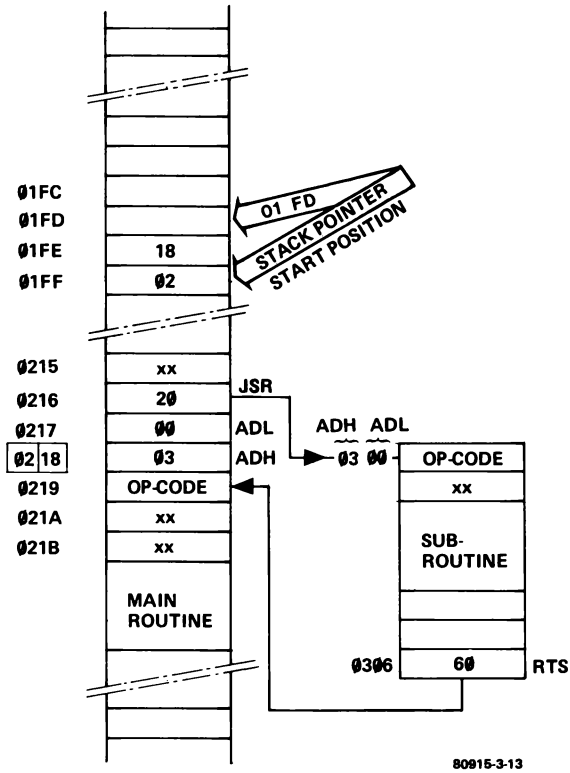


Bild 13. Programmbeispiel für den Sprung in eine Subroutine.

Die CPU erhöht automatisch die Rückkehradresse auf dem Stack um eins und erzeugt somit die richtige Rückkehradresse Ø219. Bei dieser Adresse steht ein OP-Code, der dem JSR-Befehl folgt. Das Hauptprogramm läuft nun wieder in der gewohnten Weise ab.
Nach dieser grauen Theorie wird es nun höchste Zeit, Programme zu schreiben, die Subroutinen enthalten. Dabei greifen wir auf Subroutinen im Monitor des Junior-Computers zurück. Somit ist es möglich, den System-Monitor kennenzulernen und mit wenig Aufwand Demonstrationsprogramme zu schreiben.
Wir wollen nun ein Programm schreiben, das es ermöglicht, den Wert einer gedrückten Taste im Display anzuzeigen. Den einzelnen Tasten sind hexadezimale Zahlen zugeordnet. Folgende Zahlenwerte sind den Keyboard-Tasten zugeordnet:

Ø : ØØ	5 : Ø5	A : ØA	F : ØF	PC : 14
1 : Ø1	6 : Ø6	B : ØB	AD : 1Ø	
2 : Ø2	7 : Ø7	C : ØC	DA : 11	
3 : Ø3	8 : Ø8	D : ØD	+ : 12	
4 : Ø4	9 : Ø9	E : ØE	GO : 13	

Die Werte der Datentasten Ø . . . F liegen auf der Hand: der Tastenwert ist gleich der gedrückten Taste. Auch den Funktionstasten AD, +, GO und PC sind Zahlenwerte zugeordnet. Da allen Tasten eindeutig Zahlenwerte zugeordnet sind, läßt sich auf dem Display der Wert einer gedrückten Taste anzeigen.
Bevor sich dieses Programm schreiben läßt, sind ein paar Subroutinen im System-Monitor zu besprechen. Diese Monitor-Routinen lassen sich beliebig in eigene Programme einbauen und der Programmierer muß diese Hilfsprogramme nicht selbst entwickeln. Bei der Adresse 1D8E startet die Subroutine SCANDS. Diese Subroutine sorgt dafür, daß der Inhalt der Speicherzellen ØØF9, ØØFA und ØØFB gemäß Bild 14 auf dem sechsstelligen Display des Junior-Computers angezeigt wird.
Die 7-Segment Displays können Zahlen von Ø . . . F wiedergeben. Das sind verschiedene Möglichkeiten, die sich mit vier Bits darstellen lassen. Vier Bits sind jedoch ein halbes Byte. Deshalb lassen sich in einem Speicherplatz zwei Zahlen, die auf dem Display angezeigt werden sollen, ablegen. Für sechs Displays sind somit drei Speicherplätze erforderlich. Wie bringt nun die Subroutine SCANDS die Zahlen in den drei Displaybuffer zum Display? Zuerst werden die linken vier Bits in der Speicherzelle ØØFB in den 7-Segment Code umgesetzt und Di1 zugeführt. Di1 wird ca. 500 μs eingeschaltet. Dann sind die rechten vier Bits von ØØFB an der Reihe. Auch diese vier Bits werden wieder in den 7-Segment-Code umgesetzt und an Di2 weitergegeben. Das läuft so weiter, bis der Inhalt der drei Displaybuffer im Multiplexverfahren an alle Displays durchgegeben wurde. An die Subroutine SCANDS schließt sich die Subroutine AK an. AK läßt sich ebenfalls mit einem JSR-Befehl aufrufen und beginnt an der Adresse 1DB1. Während SCANDS für das Display zuständig ist, bedient AK das Keyboard. Die Subroutine AK stellt nur fest, ob irgend eine Keyboard-Taste gedrückt wurde oder nicht. Das heißt, AK kann den Wert einer gedrückten Taste nicht feststellen. Ist nach der Rückkehr von der Subroutine AK der Accu und CPU gleich Null, dann ist keine Taste gedrückt. Ist aber der Accu ungleich null, dann ist irgendeine Taste betätigt worden. Mit einer BEQ- oder BNE-Instruktion kann jetzt festgelegt werden, ob die gedrückte Taste dekodiert oder negiert wird.
Sobald eine gedrückte Taste erkannt wurde (Accu ungleich eins nach Rückkehr von AK) muß ihr hexadezimaler Wert berechnet werden. Das

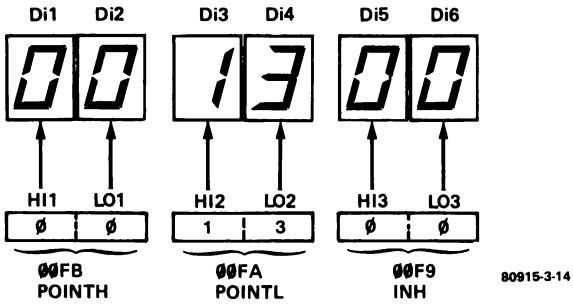


Bild 14. Die zu den Displays Di1 . . . Di6 gehörenden Datenbuffer in Page Zero.

erledigt die Subroutine GETKEY im Monitorprogramm des Junior-Computers. GETKEY startet an der Adresse 1FD9 und läßt sich mit einem JSR-Befehl aufrufen. Nach der Rückkehr von der Subroutine GETKEY hat der Prozessor im Accu den hexadezimalen Wert der gedrückten Taste.
Das grobe Flußdiagramm des Programms, das den Wert einer gedrückten Keyboard-Taste auf dem Display anzeigt, ist in Bild 15 gezeichnet. Dabei soll der Tastenwert auf den beiden mittleren Displays angezeigt werden, während die beiden äußeren Displays ØØ sein sollen. Deshalb lädt das Programm zu Beginn in die beiden Displaybuffer ØØF9 und ØØFB die hexadezimalen Zahlen ØØ. Dann kommt der Prozessor zu SCAN1. Die Subroutine SCANDS und die anschließende Subroutine AK werden aufgerufen. SCANDS bringt den Inhalt der Displaybuffer ØØF9, ØØFB ins Display, während AK entscheidet, ob irgendeine Taste gedrückt ist. Ist eine Taste gedrückt, dann verzweigt das Programm nach der Rückkehr von AK nach SCAN1. Das Display wird wieder aufgefrischt und mit AK entschieden, ob eine Keyboard-Taste immer noch gedrückt ist. Erst wenn

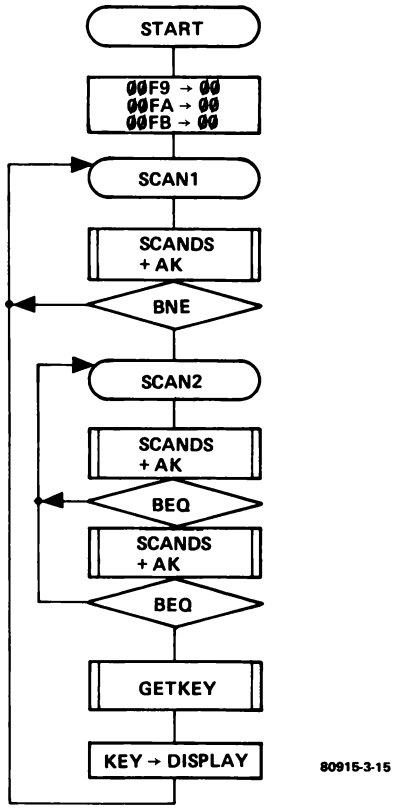


Bild 15. Das Flußdiagramm für ein Programm, das den Wert einer gedrückten Taste im Display anzeigt. Dabei werden Subroutinen des Monitorprogramms aufgerufen.

die betreffende Keyboard-Taste losgelassen ist, führt der Prozessor den bedingten BNE-Befehl nicht aus und gelangt zu SCAN2. Die Subroutinen SCANDS+AK werden wieder aufgerufen. Der Prozessor verweilt solange in der Programmschleife SCAN1, SCANDS+AK, BEQ, SCAN1, bis eine Taste gedrückt wird. Erkennt die Subroutine AK eine gedrückte Taste, dann verläßt die CPU diese Programmschleife und durchläuft abermals die Subroutinen SCANDS+AK. Der folgende BEQ-Befehl entscheidet, ob die Verzweigung zu SCAN2 führen soll oder nicht.
Auf den ersten Blick erscheint es sinnlos, die Subroutinen SCANDS+AK zweimal hintereinander zu durchlaufen. Der Sinn liegt darin, daß der Prozessor für das Durchlaufen von SCAN+AK einige Millisekunden Zeit benötigt. Während dieser Zeit ist das Tastenprellen einer gedrückten Taste abgeklungen und der Prozessor kann somit feststellen, ob es sich um eine gedrückte Taste oder einen Störimpuls handelt. Ohne diese Software-Entprellung der Tasten könnte im Extremfall der Computer eine Taste erkennen, die gar nicht gedrückt ist oder eine falsche Taste dekodieren.

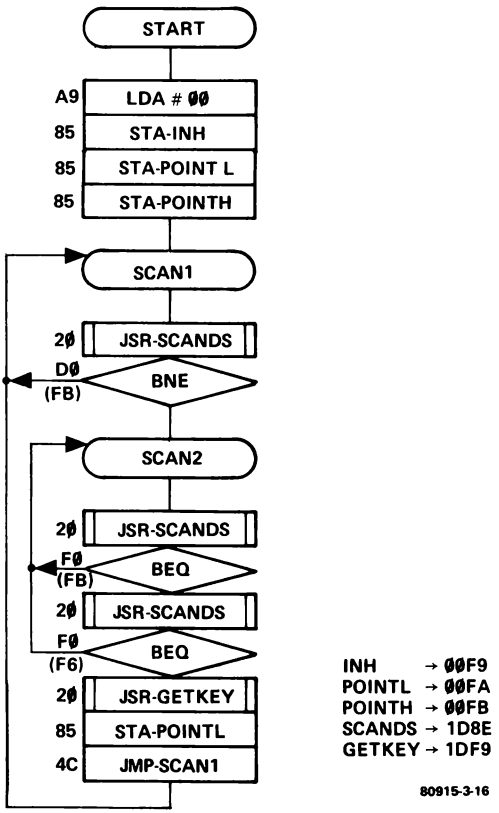


Bild 16. Detailliertes Flußdiagramm von Bild 15.

Nachdem eine gedrückte Taste erkannt wurde, springt der Prozessor zur Subroutine GETKEY und berechnet dort den Wert der gedrückten Taste. Nach der Rückkehr von dieser Subroutine befindet sich der Tastenwert im Accu und der Prozessor bringt diesen Wert zu dem Displaybuffer für die beiden mittleren Displays.
Nachdem der Tastenwert im Displaybuffer steht, springt das Programm zurück nach SCAN1 und der beschriebene Vorgang wiederholt sich. Der Computer befindet sich also in einer Unendlich-Schleife.
Das detaillierte Flußdiagramm zeigt Bild 16. Die Startadresse des Programms ist ØØØØ und die Adressen für die Displaybuffer sowie Subroutinen sind neben der Flowchart angegeben. Auch die Subroutinen des Monitors sind deutlich gekennzeichnet. Bevor wir das Programm in den Junior-Computer eingeben können, müssen noch die Offsets für die Verzweigungen berechnet werden. Das folgende Tastenprogramm veranschaulicht, wie dabei vorzugehen ist:

AD				xxxx	XX	
1	F	D	5	1FD5	D8	
GO				ØØØØ	ØØ	
Ø	B	Ø	8	ØB08	FB	→ Offset notieren!
1	Ø	Ø	D	1ØØD	FB	→ Offset notieren!
1	5	Ø	D	15ØD	F6	→ Offset notieren!
RST						
AD						
Ø	2	Ø	Ø	Ø2ØØ	XX	
DA		A	9	Ø2ØØ	A9	LDA #
+		Ø	Ø	Ø2Ø1	ØØ	
+		8	5	Ø2Ø2	85	STAZ
+		F	9	Ø2Ø3	F9	INH
+		8	5	Ø2Ø4	85	STAZ
+		F	A	Ø2Ø5	FA	POINTL
+		8	5	Ø2Ø6	85	STAZ
+		F	B	Ø2Ø7	FB	POINTH
+		2	Ø	Ø2Ø8	2Ø	JSR—
+		8	E	Ø2Ø9	8E	SCANDS + AK
+		1	D	Ø2ØA	1D	
+		D	Ø	Ø2ØB	DØ	BNE
+		F	B	Ø2ØC	FB	
+		2	Ø	Ø2ØD	2Ø	JSR—
+		8	E	Ø2ØE	8E	SCANDS + AK
+		1	D	Ø2ØF	1D	
+		F	Ø	Ø21Ø	FØ	BEQ
+		F	B	Ø211	FB	
+		2	Ø	Ø212	2Ø	JSR—
+		8	E	Ø213	8E	SCANDS + AK
+		1	D	Ø214	1D	

+		F	Ø	Ø215	FØ	BEQ
+		F	6	Ø216	F6	
+		2	Ø	Ø217	2Ø	JSR–
+		F	9	Ø218	F9	GETKEY
+		1	D	Ø219	1D	
+		8	5	Ø21A	85	STAZ
+		F	A	Ø21B	FA	POINTL
+		4	C	Ø21C	4C	JMP–
+		Ø	8	Ø21D	Ø8	SCAN 1
+		Ø	2	Ø21E	Ø2	
AD				Ø21E	Ø2	
Ø	2	Ø	Ø	Ø2ØØ	A9	Startadresse des Programms
GO				ØØØØ	ØØ	Programm läuft; das Display Nullen, solange keine Taste wurde
GO				ØØ13	ØØ	Tastenwert GO (!!!)
6				ØØØ6	ØØ	Tastenwert 6
AD				ØØ1Ø	ØØ	Tastenwert AD
DA				ØØ11	ØØ	Tastenwert DA
B				ØØØB	ØØ	Tastenwert B

So läßt sich jeder Tastenwert auf dem Display sichtbar machen. Hat man genug:

RST

Nach der Eingabe der Startadresse wird das Programm mit der GO-Taste gestartet. Das Display zeigt dann Nullen. Erst wenn abermals die GO-Taste gedrückt wird, erscheint ihr Wert im Display. Dieser Wert ist 13. Mit der RST-Taste kann man das Tastenprogramm wieder verlassen.

Implied Addressing

25 Instruktionen beziehen sich ausschließlich auf die internen CPU-Register und Flags. So lassen sich beispielsweise interne Prozessorregister miteinander vertauschen. Bisher lernten wir Instruktionen kennen, die aus einem OP-Code und einem Operanden von ein bis zwei Bytes bestanden. Die CPU 6502 hat auch Instruktionen, die nur ein Byte lang sind, also nur aus einem OP-Code bestehen. Einige dieser Instruktionen haben wir bereits kennen gelernt: CLC, SEC, DEY und RTS. Hier folgen alle Instruktionen, die Implied Addressing verwenden:

BRK Code ØØ BReak
CLC Code 18 Clear Carry
CLD Code D8 Clear Decimal mode
CLI Code 58 Clear Interrupt flag
CLV Code B8 Clear oVerflow flag
DEX Code CA DEcrement X-register by one
DEY Code 88 DEcrement Y-register by one
INX Code E8 INcrement X-register by one

INY Code C8 INcrement Y-register by one
NOP Code EA No OPeration* (eine brauchbare Instruktion)
PHA Code 48 PusH Accumulator on stack
PHP Code Ø8 PusH Processor-status on stack
PLA Code 68 PulL Accumulator from stack
PLP Code 28 PUlL Processor-status from stack
RTI Code 4Ø ReTurn from Interrupt
RTS Code 6Ø ReTurn from Subroutine
SEC Code 38 SEt Carry flag
SED Code F8 SEt Decimal flag
SEI Code 78 SEt Interrupt flag
TAX Code AA Transfer Accumulator to X-register
TAY Code A8 Transfer Accumulator to Y-register
TSX Code BA Transfer Stackpointer to X-register
TXA Code 8A Transfer X-register to Accumulator
TXS Code 9A Transfer X-register to Stack pointer
TYA Code 98 Transfer Y-register to Accumulator

Einige dieser Instruktionen wollen wir nun gesondert besprechen.
SED und CLD (set decimal mode und clear decimal mode)
Mit den Instruktionen ADC und SBC war es möglich, zum Inhalt des Accus den Inhalt einer bestimmten Speicherzelle zu addieren oder von diesem zu subtrahieren. Dabei verwendeten wir nur hexadezimale Zahlen. Es ist jedoch auch möglich, den Junior-Computer als dezimal rechnende Maschine zu benutzen. Die Befehle dafür sind SED = SEt Decimal mode und CLear Decimal mode. Die Dezimal Mode soll nur unmittelbar vor arithmetischen Operationen gewählt und nach Ende der Operation sofort wieder zurückgesetzt werden. Anderenfalls denkt der Junior-Computer ausschließlich dezimal, dann kann vieles im Programm falsch laufen! Die Addition von zwei hexadezimalen Zahlen hatte ein Carry zur Folge, wenn das Ergebnis größer als 11111111 = FF war. Ganz anders wird die Carry Flag gesetzt, wenn die Maschine in Decimal Mode ist: Immer, wenn das Ergebnis größer als 99 ist, entsteht ein Übertrag auf das neunte Bit, was einer gesetzten Carry Flag entspricht.
Deshalb folgender Tip: Soll binär gerechnet werden, dann muß zu Beginn des Programmes ein CLD-Befehl stehen. Beim Drücken der RST-Taste wird der Junior-Computer automatisch auf binäre Arbeitsweise gebracht (CLD). Soll der Computer dezimal arbeiten, so ist zu Beginn des Programms ein SED-Befehl zu geben, der nach Abschluß der dezimalen Operationen wieder aufgehoben wird (CLD).

NOP (No OPeration)

Häufig hat man ein Programm eingetippt, das viele Speicherplätze belegt und nicht arbeitet. Da bleibt nur eines übrig: Die Fehler beseitigen. Dabei ist festzustellen, daß dieser oder jener Befehl überflüssig ist, oder gar am Anfang des Programms eine Instruktion fehlt. Das ist besonders bitter, da dann das gesamte Programm nochmals neu eingetippt werden muß. Es sei denn, man fügt einige NOP-Befehle an wichtigen Stellen des Programms ein. Der Mikroprozessor führt dann die NOP-Befehle aus, was jedoch keine Auswirkung auf das Programm hat. Diese NOPs lassen sich auch durch Instruktionen überschreiben und somit ist es möglich, ohne abermaliges

Eintippen, Instruktionen nachträglich in ein Programm einzufügen. Der NOP-Befehl ist also doch nicht sinnlos, wie er auf den ersten Anblick erscheinen mag.

Push-Pull-Transfer

All Instruktionen mit Implied Addressing, deren Mnemonics mit P oder T beginnen, haben einen Datentransport von oder zu den Registern A, X, Y, S oder P zur Folge. Viele Möglichkeiten erschließen sich dem Programmierer durch diese Push-Pull-Transfer-Befehle.

An Hand eines Beispiels wollen wir das erklären. Angenommen, wir springen in eine Subroutine. Das X-Register wird sowohl in der Subroutine als auch im Hauptprogramm benötigt. Deshalb retten wir das X-Register vor dem Sprung in die Subroutine auf den Stack. Während der Subroutine steht das X-Register zur freien Verfügung. Am Ende der Subroutine holen wir dieses Register wieder vom Stack herab und stellen somit seinen ursprünglichen Zustand wieder her. Mit folgendem Programmablauf läßt sich das realisieren:

```
TXA         kopiere X-Register in den Accu
PHA         lege den Accu auf dem Stack ab
JSR-XXXX    X-Register steht zur freien Verfügung
 .
 .
 .
RTS         kehre von der Subroutine zurück
PLA         hole altes X-Register vom Stack
TAX         kopiere den Accu in das X-Register
```

Auch anders hätte man das X-Register retten können:

```
STX-SAVX    rette das X-Register in die Speicherzelle SAVX
JSR-XXXX    das X-Register steht zur freien Verfügung
 .
 .
 .
RTS         kehre von der Subroutine zurück
LDX-SAVX    stelle den alten Zustand des X-Registers her
```

Bei absoluter Adressierung benötigen die Instruktionen LDX und STX mehr Bytes als die Push-Pull-Prozedur. Der Stack ist also ein idealer zeitlicher Zwischenspeicher für Daten aller Art.

Manchmal müssen in Subroutinen alle internen Prozessorregister zur Verfügung stehen. Wie vorher das X-Register auf den Stack gerettet wurde, lassen sich auch die übrigen CPU-Register auf dem Stack ablegen und stehen dann in einer Subroutine zur freien Verfügung. Bild 17 zeigt den Programmablauf, mit dem sich die Register A, X, Y und P auf den Stack retten und anschließend wieder in den ursprünglichen Zustand herstellen lassen. SAVE rettet auf den Stack (Push-Prozedur) und RESTO stellt wieder her (Pull-Prozedur). Beim Rückholen der Register vom Stack ist darauf zu achten, daß das, was zuletzt auf dem Stack abgelegt wurde, zuerst heruntergeholt wird. Das hängt mit dem Mechanismus des Stack Pointers zusammen, der bereits bei den Subroutinen erklärt wurde.

Der Stack Pointer zeigt immer auf eine Adresse, an der Daten abgelegt

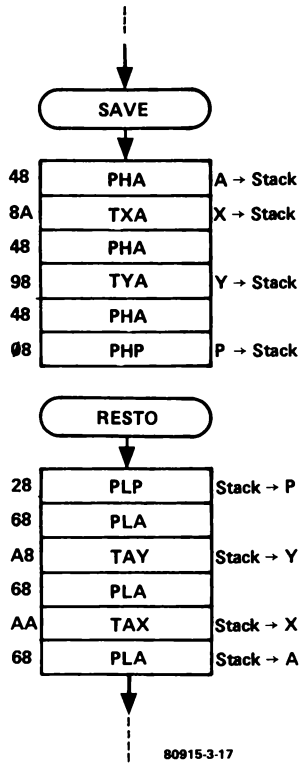


Bild 17. Mit dieser Programmsequenz lassen sich alle internen Maschinenregister der CPU 6502 auf den Stack retten und anschließend wieder herabholen (Push-Pull-Instruktionen). Nach der Rettung auf den Stack stehen alle Register zur freien Verfügung.

werden können. Bei der Initialisierung eines Computers (Reset) ist der Stack Pointer immer richtig einzurichten. Nach dem Aufrufen des Betriebsmonitors mit der RST-Taste zeigt der Stack Pointer immer auf die Adresse Ø1FF. Diese Adresse ist das Ende von Seite 1 im RAM-Speicher des Junior-Computers. Es ist jedoch auch möglich, den Stack Pointer auf eine andere Adresse in Seite 1 zu setzen. Das funktioniert so:

LDX 81 lade das X-Register mit 81
TXS setze den Stack Pointer auf die Adresse Ø181

Der Stack Pointer zeigt nun auf die Adresse Ø181. Werden Rückkehradressen oder andere Daten auf dem Stack abgelegt, so wächst der Stack ab der Adresse Ø181, Ø18Ø, usw. Da bei der Initialisierung der Stack Pointer auf die Adresse Ø1FF zeigt, lassen sich auf dem Stack 256 Daten oder 128 Rückkehradressen ablegen. Das ist mehr als ausreichend, bedenkt man, daß der Systemmonitor des Junior-Computers nur 13 Stackzellen benötigt.

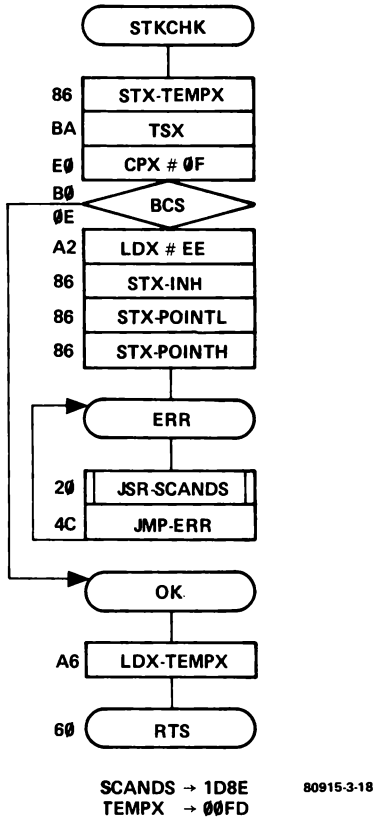


Bild 18. Dieses Programm gibt eine Fehlermeldung auf dem Display, wenn der Stackbereich überschritten wurde.

Mit der Subroutine STCHK (Bild 18) läßt sich sicherstellen, daß der Computer eine Errormeldung sendet, wenn der Speicherbereich für den Stack überschritten ist. Sobald der Stack über die Adresse Ø1ØF anwächst, entsteht auf dem Display eine Fehlermeldung in der Form EEEEEE. Mit der RST-Taste läßt sich der Computer wieder aus der ERR-Schleife holen. Die Adresse Ø1ØF wurde gewählt, um bei einer Errormeldung dem Programmierer mitzuteilen, daß nur noch wenige Plätze auf dem Stack zur Verfügung stehen.
Nur die wichtigsten Instruktionen mit Implied Addressing wurden erkärt. Weitere Instruktionen, die ebenfalls dieses Adressierungsverfahren verwenden, besprechen wir noch an einer anderen Stelle in diesem Buch.

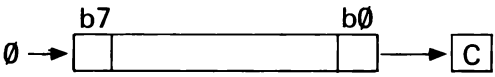
Schiebe- und Rotiere-Befehle

In diesem Kapitel wenden wir uns einer neuen Gruppe von Befehlen zu, den Schiebe- und Rotiere-Befehlen der CPU 6502. Diese Befehle benötigt

man in erster Linie dafür, um mit Software Schieberegister beliebiger Länge herstellen zu können. Schieberegister benötigt man, wenn parallel anliegende Daten seriell über eine Leitung geschickt werden sollen. Ein noch bekannteres Beispiel dürfte der Taschenrechner sein: Werden Zahlen mittels Keyboard in eine solche Rechenmaschine eingegeben, so werden die gedrückten Tasten in Form von Zahlen von links nach rechts oder umgekehrt ins Display geschoben. Diese Aufgabe wollen wir im folgenden Programm mit dem Junior-Computer lösen: das Schieben von Daten in ein Display läßt sich mit diesen Instruktionen denkbar leicht durchführen.

LSR-A-Logical Shift Right

Die Operation dieses Befehles läßt sich wie folgt darstellen:



Der LSR-Befehl schiebt den Inhalt des Accumulators oder einer Speicherzelle um eine Bitposition nach rechts. Dabei wird in das höchste Bit, also Bit 7 immer eine "Ø" hineingeschoben und das niederwertigste Bit, also Bit Ø in die Carry Flag geschoben. Nach achtmaliger Wiederholung des LSR-Befehles ist der Inhalt des Accumulators oder der betreffenden Speicherzelle immer Null. LSR beeinflußt im Prozessor Statusregister folgende Flags:

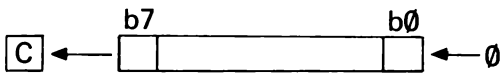
N-Flag: ist immer zurückgesetzt (N = Ø)

Z-Flag: ist gesetzt, wenn das Resultat der Schiebeoperation Null ist. Im anderen Fall ist diese Flag zurückgesetzt.

C-Flag: wird nach jedem LSR-Befehl Bit Ø im Accumulator oder der betreffenden Speicherzelle gleichgesetzt. Bit Ø wird also immer in die Carry Flag geschoben.

ASL-A-Arithmetic Shift Left

Die Operation dieses Befehls läßt sich wie folgt darstellen:



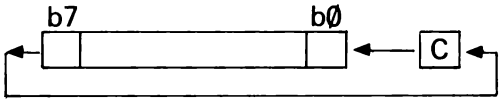
Der ASL-Befehl schiebt den Inhalt des Accumulators oder einer Speicherzelle um eine Bitposition nach links. Dabei wird das höchstwertige Bit, also Bit 7, in die Carry Flag geschoben und Bit Ø immer Null gesetzt. ASL beeinflußt im Prozessor Statusregister die folgenden Flags:

N-Flag: wird gesetzt, wenn nach dem ASL-Befehl Bit 7 = "1" ist, im anderen Fall ist diese Flag zurückgesetzt.

Z-Flag: wird gesetzt, wenn das Ergebnis der Schiebeoperation Null ist. Im anderen Fall ist diese Flag zurückgesetzt.
C-Flag: wird nach jedem ASL-Befehl Bit 7 im Accumulator oder der betreffenden Speicherzelle gleichgesetzt. Bit 7 wird also immer in die Carry Flag geschoben.

ROL - ROtate Left

Die Operation dieses Befehles läßt sich wie folgt darstellen:

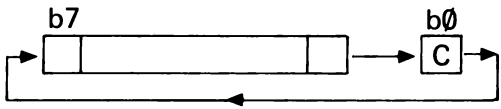


Der ROL-Befehl schiebt den Inhalt des Accumulators oder einer Speicherzelle um eine Bitposition nach links. Dabei wird das höchstwertige Bit, also Bit 7, in die Carry Flag geschoben und die Carry Flag in Bit Ø. ROL beeinflußt im Prozessor Statusregister folgende Flags:

N-Flag: wird gesetzt, wenn Bit 6 vor der Schiebeoperation Null war, im anderen Fall wird diese Flag zurückgesetzt.
Z-Flag: wird gesetzt, wenn das Ergebnis der Rotationsoperation Null ist. Im anderen Fall wird diese Flag zurückgesetzt.
C-Flag: wird nach jedem ROL-Befehl Bit 7 gleichgesetzt.

ROR - ROtate Right

Die Operation dieses Befehles läßt sich wie folgt darstellen:



Der ROR-Befehl schiebt den Inhalt des Accumulators oder einer Speicherzelle um eine Bitposition nach rechts. Dabei wird das niederwertigste Bit, also Bit Ø, in die Carry Flag geschoben und die Carry Flag in Bit 7. ROR beeinflußt im Prozessor Statusregister folgende Flags:

N-Flag: wird gesetzt, wenn die Carry Flag vor der Schiebeoperation log. 1 war, im anderen Fall wird diese Flag zurückgesetzt.
Z-Flag: wird gesetzt, wenn das Ergebnis der Rotationsoperation Null ist.
C-Flag: wird nach jedem ROR-Befehl Bit Ø gleichgesetzt.

Merke: Bei Shift-Befehlen, wie ASL und LSR, gehen Bits im Accumulator oder der betreffenden Speicherzelle verloren.
Bei Rotiere-Befehlen, wie ROR und ROL, geht keine Information verloren, solange sich die Schiebeoperation auf ausschließlich ein und dasselbe Byte beschränkt.

Accumulator Addressing

An dieser Stelle führen wir ein neues Adressierungsverfahren sein: Accumulator Addressing. Wie der Name schon sagt, bezieht sich dieses Adressierungsverfahren ausschließlich auf ein internes CPU-Register, den Accumulator. Der Befehlssatz der CPU 6502 hat nur vier Befehle, die Accumulator Addressing verwenden. Diese Befehle sind Schiebe- und Rotiere-Befehle, wie wir sie im vorigen Kapitel kennen gelernt haben. Die Befehle sind im einzelnen:

ASL-A —— ØA Arithmetic Shift Left Accumulator
LSR-A —— 4A Logical Shift Right Accumulator
ROL-A —— 2A ROtate Left Accumulator
ROR-A —— 6A ROtate Right Accumulator

Kombination von Schiebe- und Rotiere-Befehlen

Wie bereits erwähnt, werden Schiebe- und Rotiere-Befehle meist dazu benutzt, um parallel anliegende Daten seriell zu übertragen, oder um das Display eines Rechners mit Zahlen zu füllen. Wir sagten, eine gedrückte Taste wird ins Display geschoben. Genau das wollen wir jetzt machen! Wie bereits bekannt, befinden sich die Zahlen, die im 6-stelligen Display des Junior-Computers angezeigt werden, in den drei Displaybuffern POINTH (ØØFB) POINTL (ØØFA) und INH (ØØF9). Da die Darstellungsweise eines Zahlensymbols einer Hexzahl nur vier Bit benötigt, lassen sich in einem Byte zwei Hex-Zahlensymbole unterbringen. Wie füllt man nun von rechts nach links ein Display mit Zahlen? Für jedes Zahlensymbol, das ins Display geschoben werden soll, müssen vier Bits freigemacht werden. Bild 19 demonstriert, wie der Inhalt der drei Displaybuffer um vier Bitpositionen nach links geschoben wird, um Platz für ein neues Zahlensymbol in INH zu schaffen. Der Vorgang läßt sich wie folgt beschreiben:

1) Schiebe den Displaybuffer INH um eine Bitposition nach links. Der entsprechende Befehl ist ASLZ —— Ø6. *Folge:* in Bit Ø von INH wird eine Null geschoben, Bit 7 wandert in die Carry Flag.
2) Rotiere den Displaybuffer POINTL um eine Bitposition nach links. Der entsprechende Befehl ist ROLZ —— 26. *Folge*: in Bit Ø von POINTL wird die Carry Flag geschoben. Diese Carry Flag ist jedoch Bit 7 von INH. Oder man kann auch sagen, Bit 7 von INH wird über die Carry Flag in Bit Ø von POINTL geschoben. Gleichzeitig schiebt jedoch der Rotiere-Befehl Bit 7 von POINTL in die Carry Flag.
3) Rotiere den Displaybuffer POINTH um eine Bitposition nach links. Der entsprechende Befehl ist wieder ROLZ. *Folge*: in Bit Ø von POINTH wird die Carry Flag geschoben. Diese Carry Flag ist jedoch Bit 7 von POINTL. Oder man kann auch sagen, Bit 7 von POINTL wird über die Carry Flag in Bit Ø von POINTH geschoben. Gleichzeitig schiebt jedoch der Rotiere-Befehl Bit 7 von POINTH in die Carry Flag. Da hinter POINTH keine Speicherzelle vorgesehen ist, in die Bit 7 von POINTH über die Carry Flag geschoben wird, geht das hochwertigste Bit, also Bit 7, verloren. Das ist jedoch nicht schlimm, da nur sechs Displays zur Verfügung stehen.
4) Wiederhole die Schiebe-Rotiereoperation insgesamt viermal. Die *Folge* davon ist:

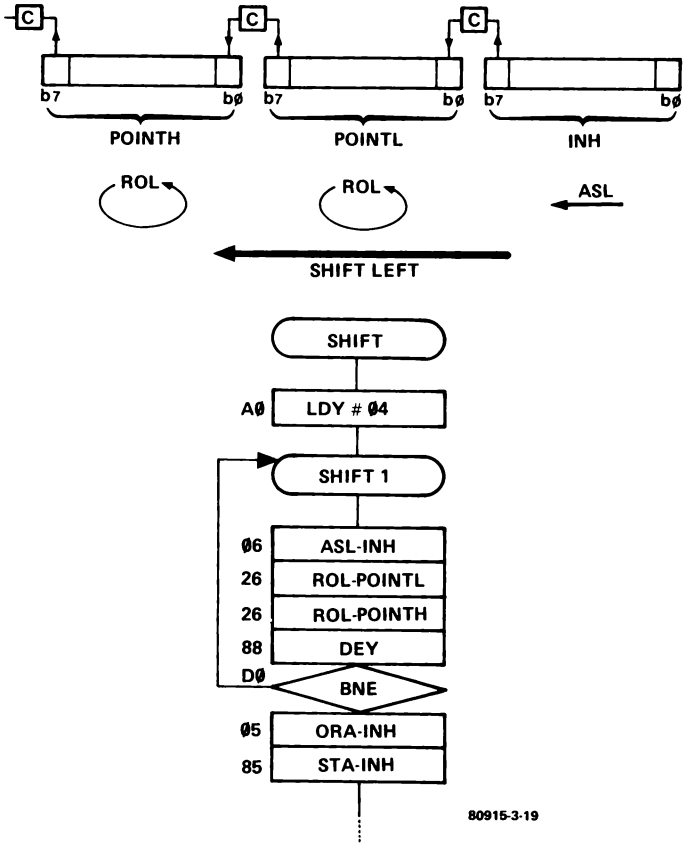


Bild 19. Das Schieben und Rotieren von Daten in die drei Displaybuffer. Die Schieberichtung ist von rechts nach links.

- Die hochwertigen vier Bits von INH werden über die Carry Flag in die niederwertigen vier Bits von POINTL geschoben.
- Die hochwertigen vier Bits von POINTL werden über die Carry Flag in die niederwertigen vier Bits von POINTH geschoben.
- Die hochwertigen vier Bits von POINTH werden hintereinander in die Carry Flag rotiert und gehen verloren.
- In die niederwertigen vier Bits von INH sind wegen dem ASL-Befehl Nullen geschoben worden. Auf diese vier Nullen läßt sich jetzt die neue Hexzahl "ordern".

Die Programmsequenz, die das Schieben, Rotieren und Ordern erledigt, ist ebenfalls in Bild 19 dargestellt. Mit dem soeben gewonnenen Wissen sind wir in der Lage, den Junior-Computer als kleine Rechenmaschine zu programmieren. Alles, was man dazu benötigt, wurde bereits an Hand von Programmbeispielen gezeigt:

– Addieren oder Subtrahieren mit den arithmetischen Befehlen der CPU 6502.
– Das Detektieren einer gedrückten Taste mit den Subroutinen AK und GETKEY.
– Die Scannung des Displays mit der Subroutine SCANDS.
– Das Schieben von Daten oder Zahlen in den Displaybuffer des Junior-Computers. Das folgende Programmbeispiel kann uns also nicht mehr schrecken!

Der Junior-Computer als kleine Rechenmaschine

Was Taschenrechner zu leisten vermögen, ist wirklich erstaunlich. Sie beherrschen neben den vier Grundrechenarten auch wissenschaftliche Funktionen, haben Fließkomma Arithmetik, Exponenten und vieles andere mehr. Prinzipiell ist es möglich, den Junior-Computer so zu programmieren, daß er beliebige Zahlen knacken kann. Doch um solche Programme entwickeln zu können, benötigt man etwas mehr Programmierpraxis als wir jetzt haben. Deshalb wollen wir zunächst den Junior-Computer als einfache Addiermaschine programmieren. Der Abwechslung halber wollen wir eine dezimale Addiermaschine schaffen. Eines soll allerdings mit einem herkömmlichen Taschenrechner überstimmen: Die Folge, wie die Tasten gedrückt werden müssen bei der Eingabe der Zahlen und Rechenoperationen. Bei einem herkömmlichen Rechner sind bei einer Addition die Tasten wie folgt zu drücken:

XX + YY = ZZ
XX → 1. Zahl
+ → Plus-Zeichen
YY → 2. Zahl
= → Gleichheits-Zeichen
ZZ → Ergebnis

Da das Display des Junior-Computers nur sechs Displays hat, lassen sich nur 6-stellige Zahlen darstellen. Um das Programm so einfach wie nur möglich zu halten, soll folgende Einschränkung gelten:

– Bei Überlauf des Displays infolge zu großer Zahlen soll keine Error-Meldung erscheinen. Dieses Problem lösen wir in einer Aufgabe am Ende dieses Kapitels.
– Die beiden zu addierenden Zahlen sollen bei der Eingabe von rechts nach links ins Display geschoben werden. Vor der Eingabe einer Zahl soll das Display automatisch gelöscht werden.
– Eine eventuell falsch eingegebene Zahl soll mit einer Clear-Taste gelöscht werden können.

Bei einer dezimalen Addition benötigt man folgende Symbole:

1) Die Zahlentasten Ø; 1; 2; 3; 4; 5; 6; 7; 8; 9
2) Die Commandotasten +; = ; Clear

Die Zahlentasten Ø . . . 9 sind bereits auf dem Keyboard des Junior-Computers vorhanden. Auch die +Commandotaste ($ 12) ist vorhanden. Aber das Gleichheitszeichen und die Clear-Taste fehlen noch. Deshalb treffen wir folgende Vereinbarung:

DA-Taste → = → $ 11
AD-Taste → Clear → $ 1Ø

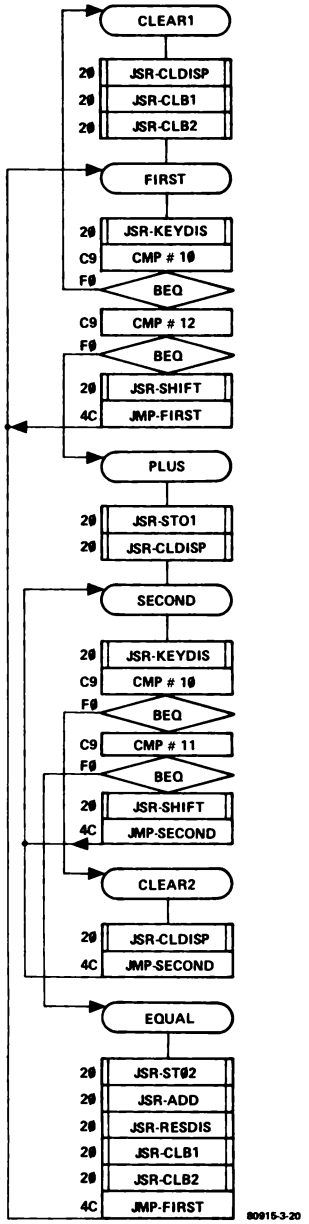


Bild 20. Hauptprogramm für die Addition von zwei 6-stelligen Dezimalzahlen. Der Junior-Computer ist als kleine Rechenmaschine programmiert.

Bild 20 zeigt die Flow Chart der Additionsroutine. Betrachtet man das Hauptprogramm, startend bei CLEAR1, so ist festzustellen, daß es durch Aneinanderreihen von mehreren Subroutinen aufgebaut wurde. Jede dieser Subroutinen erfüllt eine spezifische Aufgabe. Dadurch ist es möglich, Übersicht ins Hauptprogramm zu bringen. Wird eine Subroutine öfters benötigt, so läßt sich diese beliebig oft mit dem JSR-Befehl aufrufen. Betrachtet man die Flow Chart der Additionsroutine global, so läßt sich leicht feststellen, daß sich hinter der Programmstruktur ein Baukastensystem befindet: Die Bausteine des Hauptprogrammes sind die Subroutinen.
Bevor wir uns dem Programm im Detail zuwenden, sei noch etwas über die verwendeten Buffer-Zellen gesagt! Die Buffer-Zellen befinden sich bei den folgenden Adressen:

POINTH ØØFB	POINTL ØØFA	INH ØØF9	Display-Buffer
B12 ØØØ2	B11 ØØØ1	B1Ø ØØØØ	Buffer für 1. Zahl
B22 ØØØ5	B21 ØØØ4	B2Ø ØØØ3	Buffer für 2. Zahl
R2 ØØØ8	R1 ØØØ7	RØ ØØØ6	Buffer für das Additionsergebnis

Die Flow-Chart des gesamten Additionsprogramms zeigt Bild 2Ø. Um das Programm schnell und leicht verfolgen zu können, beschreiben wir zunächst den Ablauf in Worten:
CLEAR 1: Setze den Displaybuffer sowie die Zahlenbuffer der zu addieren Zahlen auf Null (Buffer-Reset).
FIRST: Springe zur Subroutine KEYDIS. Die Subroutine steuert das Display des Junior-Computers mit der altbekannten Subroutine SCANDS. Auch das Abfragen des Keyboards sowie die Entprellung der Tasten erledigt die Subroutine KEYDIS. Der Rücksprung von KEYDIS ins Hauptprogramm ist erst möglich, wenn ein gedrückte Taste erkannt wurde. Welche Taste gedrückt wurde, detektiert in gewohnter Weise die Subroutine GETKEY. Nach der Rückkehr von KEYDIS ins Hauptprogramm befindet sich das Äquivalent der gedrückten Taste im Accumulator der CPU.
Jetzt wird im Hauptprogramm gefragt, welche Taste gedrückt ist. Mit den folgenden CMP-Befehlen läßt sich das leicht feststellen:
CMP #$ 1Ø ist es die CLEAR-Taste?
CMP #$ 12 ist es die + Taste?
Wenn es keine dieser beiden Kommando-Tasten ist, dann kann es nur noch eine Zahlen-Taste sein. In der jetzt folgenden Subroutine SHIFT wird die Zahl ins Display des Junior-Computers geschoben und mit der Subroutine KEYDIS im Display angezeigt.
Was geschieht nun, wenn eine der Kommando-Tasten CLEAR oder + gedrückt ist? Ist die CLEAR-Taste gedrückt, dann werden wie beim Eintritt in die Additionsroutine alle Buffer gelöscht. Das Display wird also wieder mit Nullen gefüllt und die soeben eingegebene Zahl gelöscht. Ist jedoch die +Taste gedrückt, dann verzweigt sich das Programm zum Label PLUS.

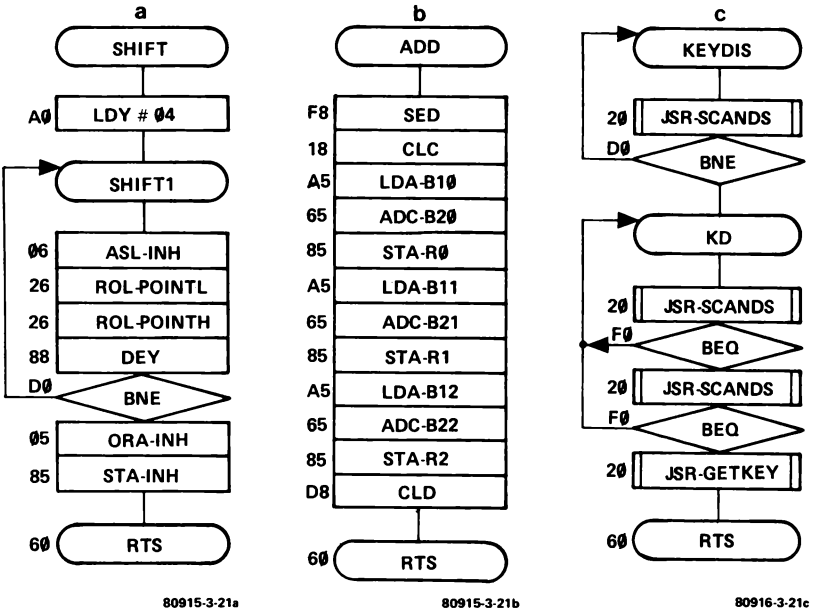
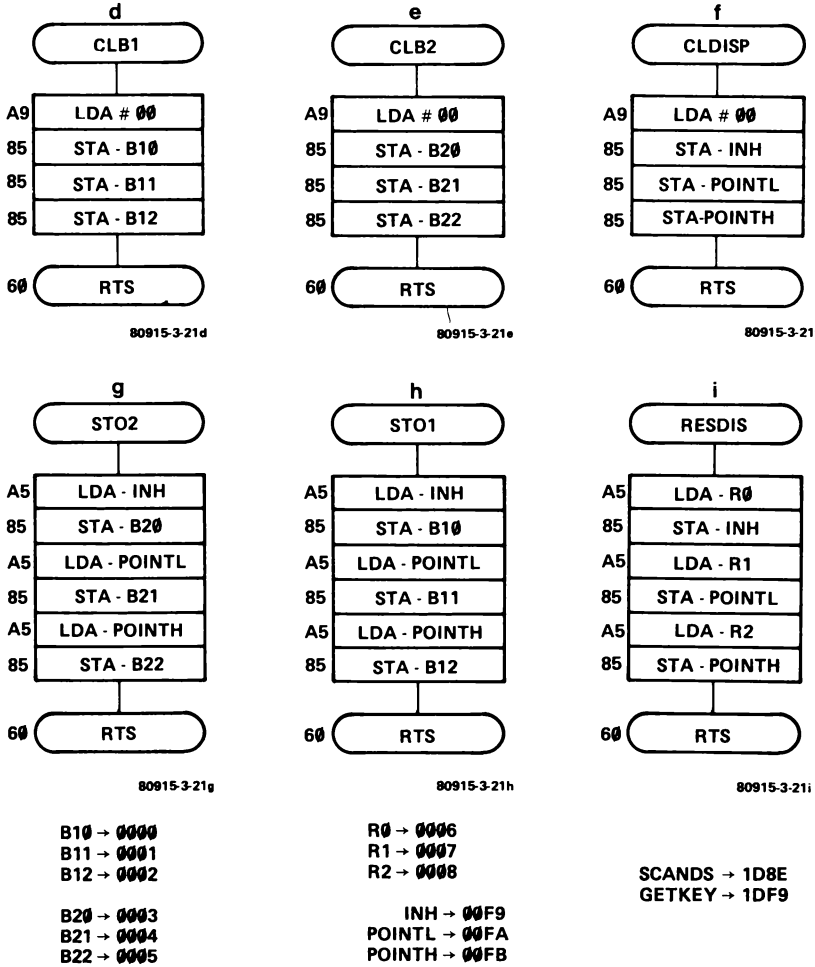


Bild 21. Die Subroutinen für das Additionsprogramm von Bild 20.

PLUS: Springe zur Subroutine STO1. STO1 bringt die soeben eingegebene Zahl vom Displaybuffer in den Buffer der ersten Zahl (Display Buffer —— Buffer der 1. Zahl). Der Display Buffer ist jetzt für die Eingabe einer neuen oder zweiten Zahl frei. Er muß vorher nur noch zurückgesetzt werden. Die folgende Subroutine CLDISP erledigt automatisch diese Aufgabe.
SECOND: Das Hauptprogramm springt abermals die Subroutine KEYDIS an. Dort wird wieder das Display und das Keyboard des Junior-Computers gesteuert. KEYDIS kann erst wieder verlassen werden, wenn eine Taste gedrückt ist. Nach der Rückkehr von KEYDIS ins Hauptprogramm befindet sich das Äquivalent der gedrückten Taste im Accumulator des Mikroprozessors.
Jetzt wird zum zweiten Mal in einer Kommandoschleife gefragt, welche Taste gedrückt ist. Mit den folgenden CMP-Befehlen läßt sich das auch jetzt feststellen:
CMP # $ 10 ist es die CLEAR-Taste?
CMP # $ 12 ist es die +Taste?
Wenn es keine dieser beiden Kommando-Tasten ist, dann kann es nur noch eine Zahlentaste sein. In der folgenden Subroutine SHIFT wird die 2. eingegebene Zahl ins Display des Junior-Computers geschoben und in der folgenden Subroutine KEYDIS angezeigt.
Ist aber die CLEAR-Taste gedrückt, dann wird die zuletzt eingegebene Zahl gelöscht. Die Subroutine CLDISP erledigt diese Aufgabe. Das Display wird also wieder mit Nullen gefüllt. Ist aber die =Taste betätigt worden, dann bedeutet das für das Hauptprogramm, daß die Zahleneingabe beendet



ist und die Addition der beiden eingegebenen Zahlen durchgeführt werden soll. Das Programm verzweigt sich dann zum Label EQUAL, um diese Aufgabe zu meistern.

EQUAL: Jetzt führt das Hauptprogramm, unterstützt von einigen Subroutinen die eigentliche Addition aus und bereitet den Junior-Computer auf eine neue Zahleneingabe vor. Im Detail läuft das folgendermaßen ab:

Die Subroutine STO2 bringt die zuletzt eingegebene Zahl vom Display Buffer in den Zahlen Buffer. Die logische Funktion ist: Display Buffer —— Buffer der 2. Zahl. Die beiden eingegebenen Zahlen befinden sich jetzt in ihren Zahlen Buffern und das Display ist somit frei für die Ausgabe des Resultats. In der folgenden Subroutine ADD werden die beiden Zahlen dezimal addiert und das Ergebnis im Resultat Buffer abgelegt. Der Inhalt des Resultat Buffers kommt jetzt mit der Subroutine RESDIS in

den Display Buffer des Junior-Computers. Die Subroutinen CLB1 und CLB2 setzen die Zahlen Buffer wieder auf Null und der Junior-Computer ist für die Eingabe von zwei neuen Zahlen bereit. Die Anzeige des Resultats erledigt wieder die Subroutine KEYDIS.
Merke: Aus dem Additionsprogramm ist leicht ersichtlich, daß Subroutinen ein Hauptprogramm überschaubar machen.
Beim Schreiben eines Programms sollte daher das erwähnte Baukastenprinzip angewendet werden. Somit läßt sich sogar ein kompliziertes Programm von Zweiten und Dritten leicht verfolgen. Bild 21 zeigt die verwendeten Subroutinen.

Indexed Addressing

In diesem Kapitel besprechen wir die indizierten Adressierungsarten der CPU 6502. Die indizierte Adressierung läßt sich in vier Gruppen aufteilen:
- Absolute Indexed, X Addressing
  Zero Page Indexed, X Addressing
- Absolute Indexed, Y Addressing
- (Indirect), Y —— Indirekt Indexed Addressing
- (Indirect,X) —— Indexed Indirect Addressing

Diese vier Gruppen von Indexed Addressing werden wir im Verlauf dieses Kapitels genau unter die Lupe nehmen. Besonders den indirekten Adressierungsarten, die das Programmieren enorm komfortabel machen, schenken wir besonderes Interesse.

Absolute Indexed, X Addressing

An Hand eines Lade-, Schreibe- und eines arithmetischen Befehls wollen wir Absolute Indexed, X Addressing erklären.
Bevor wir das folgende Programm unter die Lupe nehmen, muß noch die Bezeichnung "Feld" (englisch FIELD) erklärt werden. Unter einem Feld versteht man eine Anhäufung von Daten (= Zahlen), die hintereinander in irgendeinem Speicherbereich des Mikrocomputers abgelegt sind. Das folgende Beispiel soll das erläutern:

Ø16A AA
Ø16B BB
Ø16C 1F
Ø16D 9A
Ø16E 4D

Das Feld erstreckt sich von Ø16A . . . Ø16E und enthält fünf Daten. In großen Programmen, wie zum Beispiel ein Text-Editor, ein BASIC Compiler etc., sind häufig Daten-Felder vorhanden, die einen Speicherraum von mehreren KBytes einnehmen.
Unter einem Operanden ist der Name einer Speicherzelle zu verstehen. Beispiele sind: INH, POINTL, POINTH, B11, B12, B13, TEMPX, COUNTL, COUNTH . . . Ein Operand kann jedoch auch eine hexadezimale Zahl oder eine hexadezimale Adresse sein. Die Länge der Zahlen ist dabei unbeschränkt. Ist der Junior-Computer in Decimal Mode, dan kann jede beliebige Dezimalzahl ein Operand sein.
Im folgenden Programm (Bild 22) haben wir es mit drei Datenfeldern zu

tun, von denen jedes eine Länge von zehn Bytes hat. Die Felder heißen:
FIELD1 $0120 . . . $0129
FIELD2 $0011 . . . $001A
FIELD3 $0300 . . . $0309 mit den Daten:

FIELD1:	0120	FIELD2:	0011	FIELD3:	0300
0120	01	0011	10	0300	11
0121	02	0012	20	0301	22
0122	03	0013	30	0302	33
0123	04	0014	40	0303	44
0124	05	0015	50	0304	55
0125	06	0016	60	0305	66
0126	07	0017	70	0306	77
0127	08	0018	80	0307	88
0128	09	0019	90	0308	99
0129	0A	001A	A0	0309	AA

Das Programm hat die Aufgabe jeweils zwei Daten, die in FIELD1 und FIELD2 stehen, zu addieren und das Resultat in FIELD3 abzulegen. Zum Beispiel heißt das:
– Lade die fünfte Zahl von FIELD1 in den Accumulator
– Addiere dazu die fünfte Zahl von FIELD2
– Speichere das Ergebnis als fünfte Zahl in FIELD3 ab.
Dieses Problem läßt sich ohne weiteres mit dem altvertrauten Absolute Addressing lösen. Doch ist diese Lösung sehr umständlich. Bei langen Datenfeldern würde dieses kleine Additionsprogramm unendlich lang werden. Deshalb verwenden wir die Absolute Indizierte, X Adressierung

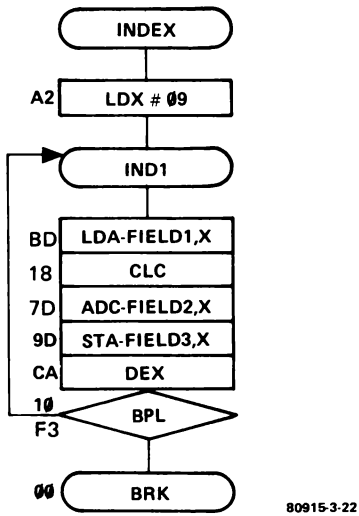


Bild 22. Programm für die Addition von zwei Feldern. Das Resultat ist in Feld 3 abgelegt.

der CPU 6502. Diese Adressierungsart ist drei Bytes lang und läßt sich folgendermaßen beschreiben:

Allgemein:	Beispiel:
OP-Code	LDA-FIELD1,X lade Accu indexed
ADL	niederwertiges Adreßbyte von FIELD1
ADH	hochwertiges Adreßbyte von FIELD1

Die effektive Adresse, von der aus der Accumulator geladen wird, ist *nicht* die Startadresse von FIELD1 (ADH, ADL = Ø12Ø) sondern ADH, ADL *plus* Inhalt des X-Registers. Ist beispielsweise der Inhalt des X-Register $Ø4 und die Startadresse von FIELD1 Ø12Ø, so gelangen in den Accumulator die Daten, die bei der Adresse Ø124 abgelegt sind. Denn die Adresse Ø124 berechnet sich: ADH, ADL + X = Ø12Ø + Ø4 = Ø124. Im Accumulator steht also nach der indizierten Ladeoperation $Ø5.
Sind alle Daten in den Feldern addiert und in Feld3 abgelegt, dann ist das X-Register null. Der Prozessor verläßt dann die Programmschleife und unterbricht das Programm (BRK). Bild 23 zeigt eine Momentaufnahme dieses Additionsprogramms.

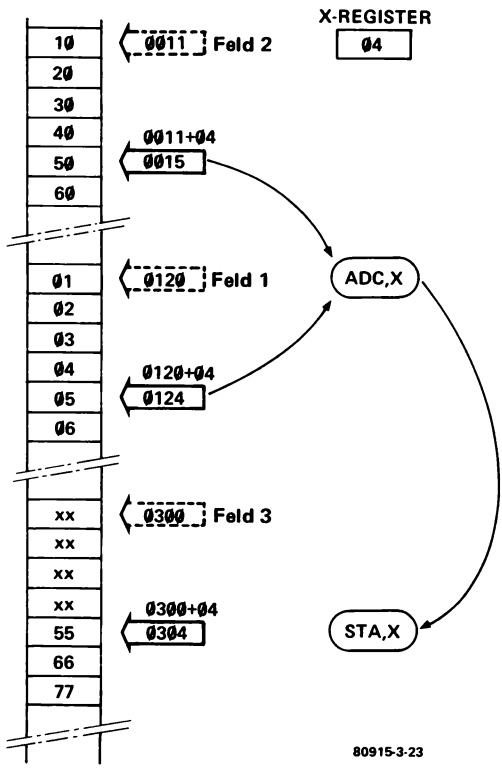


Bild 23. Eine Momentaufnahme des Programms von Bild 22 für X = Ø4.

Häufig müssen Datenblöcke von einem Speicherbereich in einen anderen transportiert werden. Man könnte auch sagen, ein Feld muß in einen anderen Speicherbereich bewegt werden. Bild 24 zeigt, was damit gemeint ist. Der Datenblock, der transportiert werden soll beginnt bei FROMAD. Die Startadresse ist Ø172. Der Datenblock hat nur fünf Daten. Diese Daten werden in einen neuen Speicherbereich des Junior-Computers gebracht, der bei der Adresse TOAD oder ØØ3AØ beginnt.
Verfolgt man den Ablauf der Flow Chart, dann steht zunächst im X-Register Ø4. Dann kommen in den Accumulator die Daten, die an der Adresse FROMAD + X = Ø172 + Ø4 = Ø176 abgelegt sind. Der folgende STA,X-Befehl speichert den Inhalt des Accumulators an der Adresse TOAD + X = ØØ3AØ + Ø4 = ØØ3A4 ab. Das Indexregister X wird jetzt um Eins dekrementiert und somit der Zugriff auf eine neue Speicherzelle ermöglicht. Der folgende Verzweigungsbefehl BPL führt solange zu einer Verzweigung, bis der Inhalt des X-Registers negativ ist. Die Werte, die das X-Register während des Programmes annimmt, sind: Ø4, Ø3, Ø2, Ø1, ØØ. Am Ende des Programms MOVE wird in den Betriebsmonitor des Junior-Computers gesprungen, der an der Adresse 1C1D beginnt. Nach der Ausführung des Programms meldet sich der Junior-Computer mit einem leuchtendem Display zurück.
Wie das Programm MOVE mit dem Keyboard in den Junior-Computer eingegeben wird, zeigt das folgende Tastenprogramm:

AD				xxxx	xx	
Ø	2	Ø	Ø	ØØØØ	xx	
DA		A	2	ØØØØ	A2	LDX#
+		Ø	4	ØØØ1	Ø4	
+		B	D	ØØØ2	BD	LDA-,X
+		7	2	ØØØ3	72	ADL von FROMAD
+		Ø	1	ØØØ4	Ø1	ADH von FROMAD
+		9	D	ØØØ5	9D	STA-,X
+		A	Ø	ØØØ6	AØ	ADL von TOAD
+		Ø	3	ØØØ7	Ø3	ADH von TOAD
+		C	A	ØØØ8	CA	DEX
+		1	Ø	ØØØ9	1Ø	BPL
+		F	7	ØØØA	F7	Offset
+		4	C	ØØØB	4C	JMP
+		3	3	ØØØC	33	ADL von Monitor-Adresse
+		1	C	ØØØD	1C	ADH von Monitor-Adresse
AD				ØØØD	1C	
Ø	2	Ø	Ø	ØØØØ	A2	
GO				ØØØØ	A2	Programmstart

Bevor man das Programm startet, müssen noch beginnend bei der Adresse Ø172 die Daten AA, BB, CC, DD, und EE in den Speicher geladen werden. Nach dem Starten des Programms kann man an den Adressen ØØ3AØ . . . ØØ3A4 nachsehen, daß dort tatsächlich eine Kopie des Datenblocks steht.

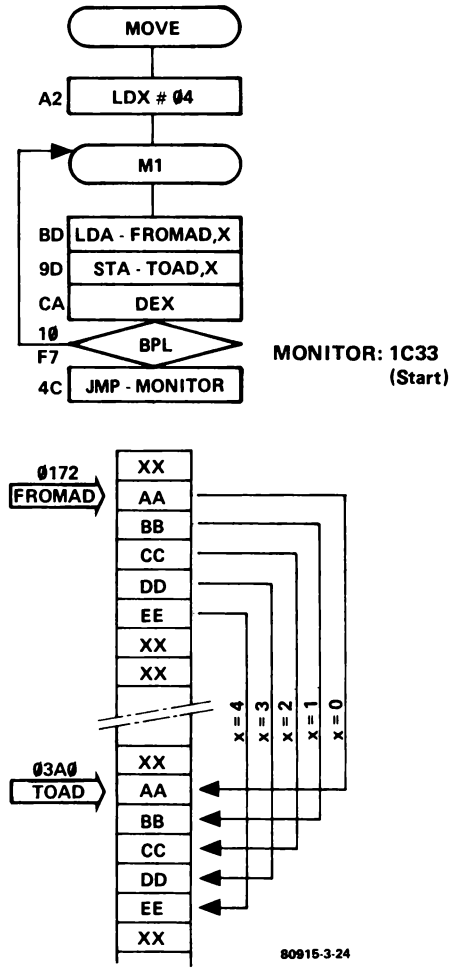


Bild 24. Dieses Programm kopiert Daten von einem Speicherbereich in einen anderen.

Lädt man am Anfang des Programms das X-Register mit $FF, dann lassen sich maximal 256 Byte von einem Speicherbereich in einen anderen kopieren. Dabei ist zu bemerken, daß sich die beiden Speicherbereiche nicht überlappen dürfen. Sonst werden im Überlappungsgebiet Daten, die kopiert werden sollen, irrtümlicherweise überschrieben beziehungsweise zerstört.

Zero Page Indexed,X Addressing

Zero Page Indexed,X Addressing ist eine besondere Form von Absolute Indexed,X Addressing. Die Schreibweise für Zero Page Indexed,X Addressing bei einem Ladebefehl ist:
LDA-Operand,X OP-Code B5

Der Operand ist der Name der Adresse einer Speicherzelle, die sich innerhalb der Page Zero befindet. Der Adreßbereich ist also ØØØØ . . . ØØFF, wie wir das von Zero Page Addressing her wissen.

Absolute Indexed,Y und Zero Page Indexed,Y

Diese Adressierungsarten bedürfen keiner Erklärung, da sie dieselbe Funktion ausführen wie Indexed,X Addressing. Der einzige Unterschied zwischen Indexed,X und Indexed,Y besteht darin, daß im ersten Fall das X-Register als Indexregister Verwendung findet und im zweiten Fall das Y-Register. Die Frage ist nun: warum zwei verschiedene Indexregister? Der Vorteil von zwei Indexregistern liegt darin, daß man in einem Programm mit den beiden Indexregistern zwei Programmschleifen ineinander verschachteln kann. Im folgenden Kapitel werden wir diesen Vorteil kennen lernen. Später zeigen wir auch bei einer Codeumwandlung, wie einfach und übersichtlich sich Programme schreiben lassen, weil bei der 6502-CPU zwei Indexregister zur Verfügung stehen.

Indirekte Adressierungsverfahren

Die indirekten Adressierungsverfahren stehen am Schluß von Kapitel 3. Damit schließen wir die Beschreibung der 13 Addressing Modes der 6502-CPU ab. Vorweg sei bemerkt, daß Indirect Addressing kein leicht zu verstehendes Adressierungsverfahren ist! Deshalb wollen wir uns wieder an Hand von Beispielen mit dieser Adressierungsart vertraut machen. Ist aber dieses Adressierungsverfahren in Fleisch und Blut übergegangen, dann wird der Programmierer das enorm starke Instruction Set der 6502-CPU schätzen lernen: Durch die indirekte Adressierung werden Programme kurz und sehr übersichtlich. Ein weiterer, nicht übersehbarer Vorteil ist, daß Adressen, in die geschrieben oder aus denen gelesen werden soll, bei der Erstellung eines Programms noch nicht bekannt sein müssen. Der Programmierer hat die Möglichkeit, später diese Adressen in Page Zero abzuspeichern oder gar dem Computer die Aufgabe übertragen, die gewünschten Adressen zu berechnen. Das indirekte Adressierungsverfahren ist sicherlich der Hauptgrund, daß sich die 6502-CPU zu einem der erfolgreichsten Mikroprozessoren gemausert hat. Der Einsatz in Prozeßrechneranlagen, Navigationscomputern und die Beliebtheit bei Computeramateuren sind der beste Beweis dafür. Haben andere Prozessoren drei, sechs oder ein paar Pointer-Register mehr auf dem CPU-Chip, so hat die 6502-CPU mit Ausnahme des Stack Pointers kein einziges Pointerregister auf dem Chip. Dafür läßt sich aber die gesamte Page Zero als Pointerregister programmieren. Da ein Pointer 16 Bit breit ist, lassen sich in Page Zero 128 Pointer unterbringen! Doch was es mit diesen Pointern auf sich hat, werden wir nun besprechen.

Indirekt Indexed Addressing

Bevor wir die Indirekt Indizierte Adressierung im Einzelnen besprechen, rufen wir uns nochmal die absolute indizierte Adressierung ins Gedächtnis. Dabei verwenden wir als Indexregister das Y-Register. Dann wird die Absolute Indexed,Y Addressing mit Indirect Indexed Addressing verglichen.

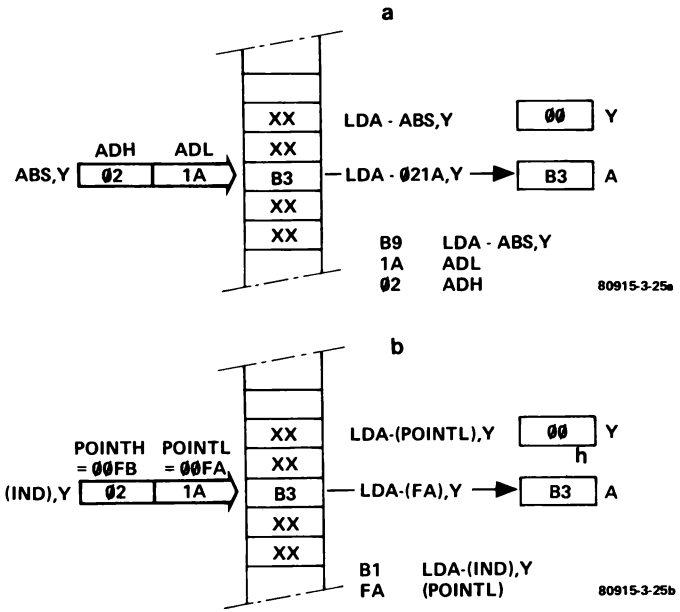


Bild 25. Das Laden des Accus mit Indirect Indexed Addressing. Der Inhalt der Speicherzellen POINTH, POINTL ist ein Adreßpointer, der auf die Adresse zeigt, von der die Daten in den Accu geladen werden.

Bild 25 zeigt zunächst Absolute Indexed,Y Addressing, wobei Y als Indexregister verwendet wird. Die Operation, die ausgeführt wird, ist das indizierte Laden des Accumulators mit dem Inhalt einer Speicherzelle. Das Y-Register ist in unserem Beispiel $ØØ. Im Speicher stehen die einzelnen Bytes folgendermaßen hintereinander:

AØ OP-Code LDY
ØØ Operand
B9 OP-Code LDA-ABS,Y
1A niederwertiges Adreßbyte
Ø2 hochwertiges Adreßbyte (Operand)

Der OP-Code B9 veranlaßt die CPU die Daten, die an der Adresse Ø21A abgelegt sind, in den Accumulator zu laden. Denn die effektive Adresse, von der aus geladen wird, berechnet sich zu:

Adresse = ADH,ADL+Y = Ø21A+ØØ = Ø21A

Soweit also nichts Neues, sieht man einmal davon ab, daß gegenüber dem vorangehenden Abschnitt in diesem Buch das Y-Register als Indexregister Verwendung fand. Ist der Operationscode B9 ausgeführt, dann steht B3 im Accumulator der CPU (Bild 25a).

Bei der absoluten Adressierung, die eigentliche direkte Adressierung heißen müßte, muß immer die Adresse, von der aus geladen oder in die geschrieben werden soll direkt bekannt sein. Das heißt, die absolute Adresse muß direkt auf den OP-Code folgen. Dasselbe gilt auch für die indizierte Adres-

sierungsart. Bei dieser Adressierungsart wird der Inhalt des Indexregisters zu der Adresse addiert, die direkt dem OP-Code folgt.
Bei der indirekten Adressierung ist die Adresse, von der aus geladen oder in die geschrieben werden soll fürs Erste noch nicht bekannt. Bild 25b macht das deutlich:
Auch hier werden wieder die Daten, die sich in der Adresse Ø21A befinden, in den Accumulator geladen. Allerdings nicht wie vorher Absolute Indexed,Y, sondern Indirect Indexed,Y. Die Schreibweise dieser Adressierungsart ist: (IND),Y. Zwischen den beiden Klammern steht eine indirekte Adresse, bei der die CPU nachsieht, von welcher effektiven Adresse Daten geholt oder zu welcher Daten geschrieben werden sollen. Diese indirekte Adresse, ist bei der CPU 6502 *immer* eine Page Zero Adresse. Da die effektive Adresse, das ist die Adresse, mit deren Daten irgendetwas geschehen soll, bei der CPU 6502 16 Bit lang ist, müssen für diese Adressen zwei Bytes in der Page Zero reserviert werden. Mit diesem Wissen ist es jetzt leicht, Indirect Indexed Addressing zu beschreiben. Was geschieht in Bild 25b bei der indirekten indizierten Adressierung? Wie bereits erwähnt, wird der Inhalt der Speicherzelle Ø21A in den Accumulator der CPU geladen. Um das zu begreifen, fassen wir die einzelnen Operationen, die der Mikroprozessor bei der indirekten indizierten Adressierung ausführt, in Worte:

1. In Page Zero befinden sich zwei aufeinanderfolgende Adressen, genannt POINTL und POINTH. Ihre Hex-Adressen sind:
   POINTL = ØØFA
   POINTH = ØØFB
2. POINTL und POINTH sind RAM-Zellen, deren Inhalte sich beliebig modifizieren lassen. Wir schreiben mittels Keyboard folgende Daten in diese Speicherzellen:
   Daten in POINTL: 1A
   Daten in POINTH: Ø2.
   Beide Bytes hintereinander gelesen lassen sich wieder als eine absolute Adresse interpretieren. Diese absolute Adresse setzt sich wiederum aus einem niederwertigen Adreßbyte 1A und einem hochwertigen Adreßbyte Ø2 zusammen. Das niederwertige Adreßbyte befindet sich in der Speicherzelle POINTL und das hochwertige Adreßbyte in der Speicherzelle POINTH.
3. Der Zugriff zu den Speicherzellen POINTL, und POINTH erfolgt über die indirekte Adressierung der 6502-CPU. Der OP-Code B1 steht für LDA-(IND),Y. Gemeint ist damit, die CPU greift zuerst auf die indirekten Adressen POINTL und POINTH zu, und holt aus ihnen die direkte Adresse, von der aus geladen werden soll. Dabei ist die direkte Adresse dasselbe wie eine absolute Adresse.
4. Zu der direkten oder absoluten Adresse, die in den Speicherzellen POINTL und POINTH abgelegt ist, wird noch der Inhalt des Y-Registers addiert. Im vorliegenden Beispiel ist der Inhalt des Y-Registers Null. Die direkte Adresse, von der die Daten in den Accumulatoren geladen werden berechnet sich also: Ø21A + Y = Ø21A + ØØ = Ø21A. In den Accumulator gelangen also tatsächlich die Daten, die in der Speicherzelle mit der Adresse Ø21A abgelegt sind.
5. Da sich die indirekten Adressen bei der CPU 6502 immer in Page Zero

befinden, sind Instruktionen, die Indirect Indexed Addressing verwenden, nur zwei Byte lang. Wenn wir eine indirekte Ladeoperation in den Speicher mittels Keyboard schreiben, dann sieht das für die oben beschriebene Ladeoperation folgendermaßen aus:

B1 OP-Code von LDA-(IND),Y

FA Operand, ist das erste indirekte Adreßbyte in Page Zero, in dem das niederwertige Adreßbyte der direkten Ladeadresse abgelegt ist. Aus ØØFB holt sich die CPU automatisch das hochwertige direkte Adreßbyte. Schreiben wir nun die indirekte indizierte Adressierung im gewohnten Format an, dann sieht das so aus:

```
AØ # ØØ  LDY ØØ          lade das Y-Register mit ØØ
B1 FA    LDA-(POINTL),Y   lade den Accumulator indirekt
```

Jetzt läßt sich leicht eine Definition der indirekten indizierten Adressierung angeben.

Merke: Instruktionen, die Indirect Indexed Addressing verwenden, sind zwei Bytes lang. Das erste Byte ist der OP-Code. Das ihm folgende zweite Byte zeigt auf eine indirekte Adresse in Page Zero. Zum Inhalt dieser indirekten Speicherzelle in Page Zero wird der Inhalt des Y-Registers addiert. Das Resultat ist das niederwertige Adreßbyte der direkten oder absoluten Adresse. Entsteht bei dieser Addition ein Übertrag auf die Carry Flag, dann wird diese automatisch zum Inhalt der folgende indirekten Page Zero Adresse addiert. Das Ergebnis davon ist das hochwertige Adreßbyte der direkten oder absoluten Adresse.

Blocktransfer mit Indirekt Indexed Addressing

Einen Blocktransfer führten wir bereits in Bild 24 mit Absolut Indexed,X Addressing durch. Mit diesem Programm ließen sich maximal 256 Bytes von einem Speicherbereich in einen anderen kopieren. Häufig müssen jedoch größere Datenmengen als 256 Byte von einem Speicherbereich in einen anderen transportiert werden. Mit der winzigen Subroutine BLMOVE in Bild 26 ist das ohne weiteres möglich. Mit dieser Subroutine lassen sich bis zu 255 Datenblöcke, von denen jeder 256 Bytes enthält von einem Speicherbereich in einen anderen transportieren. Daß die Subroutine BLMOVE, mit der sich enorm viele Daten transportieren lassen, so kurz geschrieben werden kann, liegt am indirekten indizierten Adressierungskonzept der 6502-CPU.

Im Beispiel Bild 26 wollen wir der Einfachheit halber nur zwei Datenblöcke von je 256 Bytes von einem Speicherbereich in einen anderen kopieren. Dazu treffen wir folgende Vereinbarungen:

1. Die Beginnadresse des zu transportierenden Datenblocks sei Ø2ØØ. Diese direkte oder absolute Adresse sei in den beiden aufeinanderfolgenden Zero Page-Speicherzellen BEG und BEG+1 abgelegt. BEG und BEG+1 weisen wir folgende Adressen zu:
   BEG = ØØØØ der Inhalt ist FRADL = ØØ
   BEG+1 = ØØØ1 der Inhalt ist FRADH = Ø2
2. Der Adreßbereich, in den die beiden Datenblöcke kopiert werden sollen, beginnt bei der Adresse A8ØØ. Diese direkte oder absolute Adresse sei in den beiden aufeinanderfolgenden Zero Page Speicherzellen MOV und MOV+1 abgelegt. MOV und MOV+1 weisen wir folgende Adressen zu:

MOV = ØØØ2 der Inhalt ist TOADL = ØØ
MOV+1 = ØØØ3 der Inhalt ist TOADH = A8

3. Die Anzahl der Blöcke, die transportiert werden soll, steht in der Speicherzelle BLOCKS. Wollen wir 12 Datenblöcke transportieren, so schreiben wir in diese Speicherzelle ØC. Im vorliegenden Beispiel wollen wir aber nur zwei Datenblöcke transportieren. Deshalb schreiben wir in die Speicherzelle BLOCKS den Wert Ø2. Der Speicherzelle BLOCKS weisen wir folgende Adresse zu:
   BLOCKS = ØØØ4 der Inhalt ist N = Ø2.

Das Laden der Bytes in die Speicherzellen BEG, BEG+1, MOV, MOV+1 und BLOCKS läßt sich sowohl von Hand mittels Keyboard, als auch von einem Programm erledigen. Das Programm DEFMOV setzt automatisch die direkten Adressen in die dafür reservierten indirekten Adressen (ØØØØ . . . ØØØ4) ein. Anschließend springt es in die Subroutine BLMOVE und führt den Datentransport in der gewünschten Weise durch. Was geschieht nun im Einzelnen in der Subroutine BLMOVE? Das wollen wir jetzt beantworten:

1. Das Indexregister X wird als Blockzähler programmiert. Dazu wird es mit dem Inhalt der Speicherzelle Blocks geladen und erhält somit die Information, wieviele Datenblöcke der Programmierer transportieren möchte.
2. Das Y-Register findet als echtes Indexregister Verwendung. Sein Inhalt zeigt an, die wievielte Speicherzelle innerhalb eines Datenblocks von 256 Byte angesprochen ist. Ist der Inhalt des Y-Registers beispielsweise 21, dann ist die 33. Speicherzelle innerhalb dieses Datenblocks angesprochen.
3. Die Daten, die von einem Speicherbereich in den anderen kopiert werden sollen, holt die CPU über Indirect Indexed Addressing in den Accumulator. Der Befehl dafür ist: LDA-(BEG),Y. Anschließend schreibt die CPU die soeben erhaltenen Accumulator-Daten mit Indirect Indexed Addressing in den neuen Speicherbereich ein. Der Befehl dafür ist: STA-(MOV),Y.
4. Durch das anschließende Dekrementieren des Y-Registers hat die CPU auf eine neue Speicherzelle innerhalb des 256-Byte-Datenblocks Zugriff. Das Y-Register hat in diesem Programm die Werte: ØØ, FF, FE . . . Ø2, Ø1, ØØ, FF, FE usw.
5. Immer wenn ein Datenblock von 256 Bytes von einem Speicherbereich in einen anderen transportiert wurde, wird der Inhalt der beiden indirekten Adressen BEG+1 und MOV+1 inkrementiert. Somit schafft sich die CPU Zugriff auf einen neuen Datenblock, der wiederum 256 Bytes enthält.
6. Da jetzt bereits ein Datenblock von einem Speicherbereich in einen anderen transportiert wurde, kann das als Blockzähler programmierte X-Register um Eins dekrementiert werden. Erst wenn alle Datenblöcke transportiert sind, ist der Inhalt des X-Registers ØØ. Dann verläßt die CPU die Transportschleife LOOP und springt zurück ins Hauptprogramm.

Die oben besprochenen sechs Punkte sind in Bild 26 auch grafisch dargestellt. Erst jetzt dürfte Indirect Indexed Addressing bis ins Detail klar werden:

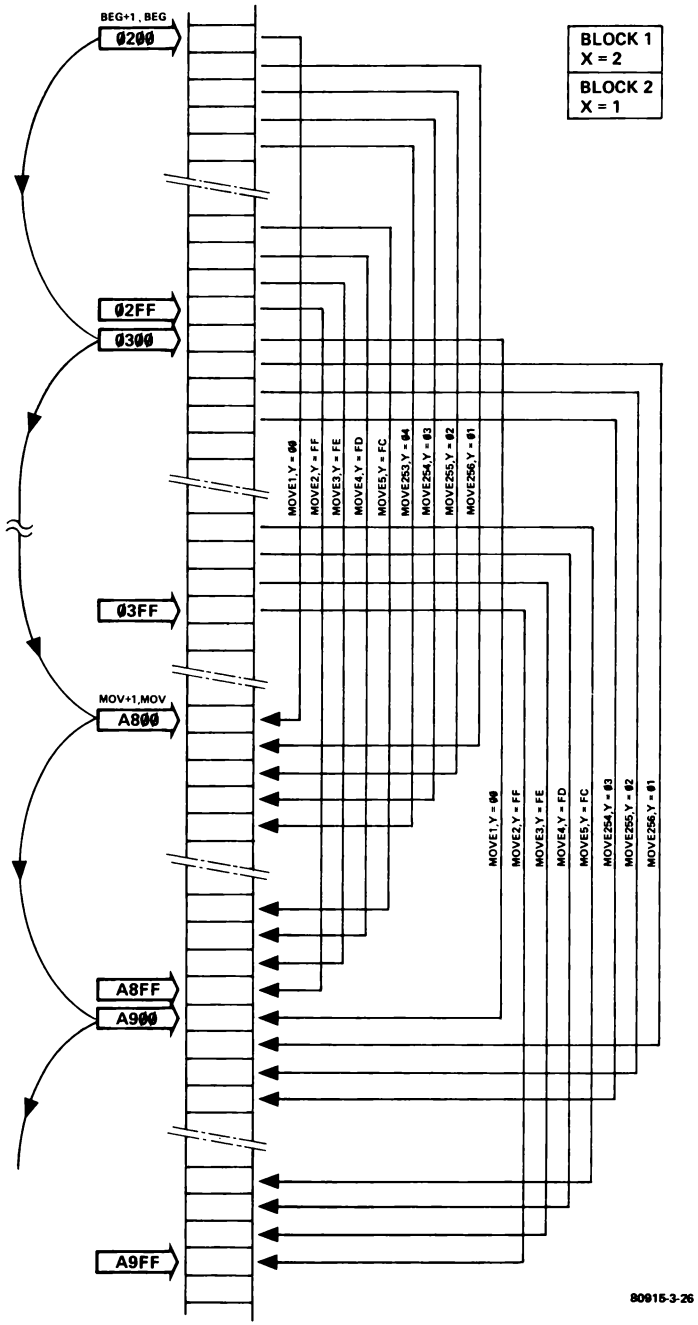


Bild 26. Das Verschieben von zwei Datenblöcken mit je 256 Bytes. Das Programm, das diese Verschiebung durchführt, zeigt Bild 27.

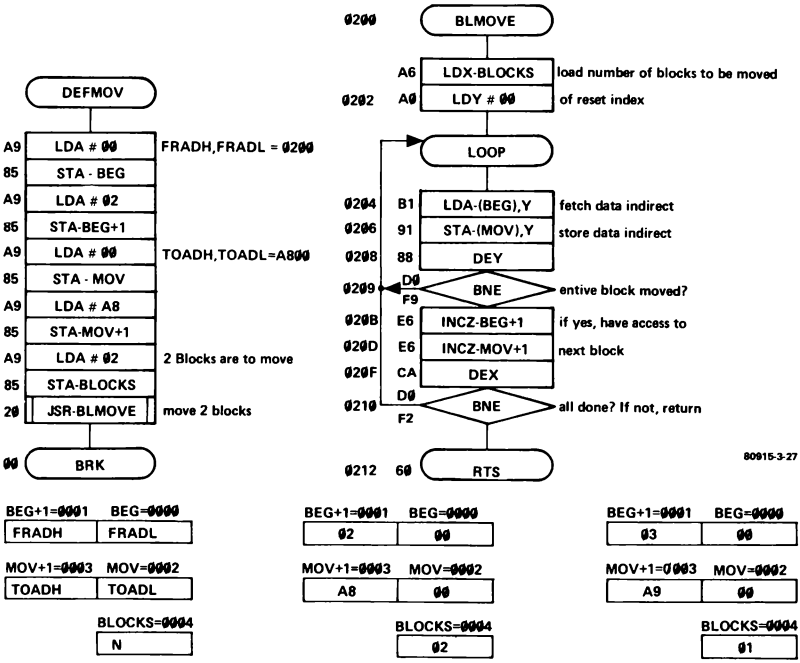


Bild 27. Mit diesem Programm lassen sich maximal 256 Speicherblöcke mit je 256 Bytes verschieben. Das ist der gesamte adressierbare Speicherbereich der CPU 6502. Die Zahl in der Speicherzelle BLOCKS gibt an, wieviele Datenblöcke transportiert werden sollen.

- In den indirekten Adressen BEG+1,BEG ist die direkte Adresse gespeichert, die der Mikroprozessor ansprechen will. *In Zukunft bezeichnen wir Speicherzellen wie BEG und BEG1 als Adreßpointer (BEG+1, BEG = Adreßpointer). Ein Adreßpointer ist aus zwei aufeinanderfolgenden indirekten Speicherzellen innerhalb der Page Zero "zusammengebaut". Der Inhalt des Adreßpointers zeigt direkt auf die Speicherzelle, mit der die CPU viel über den Adreß- und Datenbus Kontakt aufnehmen möchte.*
- Über das Indexregister Y hat der Mikroprozessor auf 256 Speicherzellen Zugriff. Diese 256 Speicherzellen beginnen bei der Adresse, auf die der Adreßpointer BEG+1,BEG = Ø2ØØ zeigt. Die effektive Adresse berechnet sich zu: BEG+1,BEG+Y. Ist Y = ØØ, dann kommen bei der indirekten Ladeoperation die Daten an der Adresse Ø2ØØ in den Accumulator, ist aber Y = FF, dann kommen die Daten an der Adresse Ø2FF in den Accumulator.
- Bei der Schreiboperation wird der Adreßpointer MOV+1,MOV verwendet. Auch zu diesem Pointer wird der Inhalt des Indexregisters Y addiert. Die Lade- und die Schreiboperationen in der Grafik in Bild 26 mittels indirekter Adressierung unter Verwendung des Y-Registers dürften nun verstanden worden sein.

Im folgenden Kapitel wenden wir uns einer neuen Adressierungart der CPU 6502 zu, die wiederum das soeben beschriebene Pointerkonzept verwendet. Es ist empfehlenswert, dieses Kapitel nochmals durchzuarbeiten, wenn man noch nicht verstanden hat, was ein Adreßpointer ist und wie man diesen manipulieren kann.

Indexed Indirect Addressing

Nachdem wir im vorigen Kapitel Indirect Indexed Addressing sehr genau beschrieben haben, können wir Indexed Indirect Addressing in kurzen Zügen beschreiben. Beide Adressierungsverfahren haben gemeinsam, daß sie Speicherzellen in Page Zero als Adreßpointer verwenden. Auch die Länge der Instruktion ist in beiden Fällen nur zwei Bytes lang.
Indexed Indirect Addressing wird dazu verwendet, um über *einen* Adreßpointer *ein* Datenbyte aus dem Speicher zu holen oder *ein* Datenbyte im Speicher zu modifizieren. Die Adreßpointer sind wiederum in aufeinanderfolgenden Speicherzellen in Page Zero abgelegt. Als Indexregister wird das X-Register verwendet. An Hand von Bild 28 wollen wir den Ablauf einer indizierten indirekten Ladeoperation beschreiben.
Der obere Teil in diesem Bild zeigt Indexed Indirect Addressing allgemein, während im unteren Teil mit konkreten Daten gearbeitet wird. An der Adresse Ø32A trifft die CPU auf den OP-Code A1 = LDA-(IND,X), der eine indirekte indizierte Ladeoperation des Accus ausführt. Die Adressen, mit deren Daten der Accu geladen werden soll, sind als Pointer in Page Zero abgelegt. Der erste Adreßpointer ist in den Speicherzellen mit den Adressen ØØ9Ø, ØØ91, der zweite Adreßpointer in den Speicherzellen ØØ92, ØØ93 usw. abgelegt. Über welchen Adreßpointer die Daten in den Accu geladen werden sollen, entscheidet das X-Register.
Aus dem unteren Teil von Bild 28 wird das ersichtlich:
In Page Zero sind drei Adreßpointer abgelegt. Ihre Adressen sind ØC24, 1CFD und 1CFE. Ist dabei das X-Register ØØ, dann kommen die Daten von der Speicherzelle ØC24 in den Accu. Wurde vor der Ladeoperation das X-Register beispielsweise mit Ø4 geladen, dann kommen die Daten von der Adresse 1CFE in den Accu. Durch zweimaliges Inkrementieren des X-Registers hat man auf den folgenden Pointer, der in Page Zero abgelegt ist, Zugriff.
Wie aus diesem Beispiel leicht ersichtlich ist, kann man mit Indexed Indirect Addressing Page Zero als "Pointer Look Up Table" programmieren. Das heißt, in Page Zero können Adreßpointer abgelegt werden, die auf bestimmte Daten im Speicher zeigen. Diese Pointer lassen sich mit Indexed Indirect Addressing sowie dem X-Register als Index aufrufen. Indexed Indirect Addressing wird relativ selten gebraucht. Jedoch beim Implementieren von höheren Programmiersprachen, wie zum Beispiel Basic, trifft man öfters auf diese Adressierungsart.

NMI, IRQ, RESET, oder das Interruptkonzept der 6502-CPU

Die Begriffe Interrupt und Vektor werden im letzten Teil von Kapitel 3 beschrieben. Damit wurden alle möglichen Adressierungsverfahren der 6502-CPU besprochen. Vorweg sei bemerkt, daß wir es hier wieder mit einem indirekten Adressierungsverfahren zu tun haben, ähnlich wie das

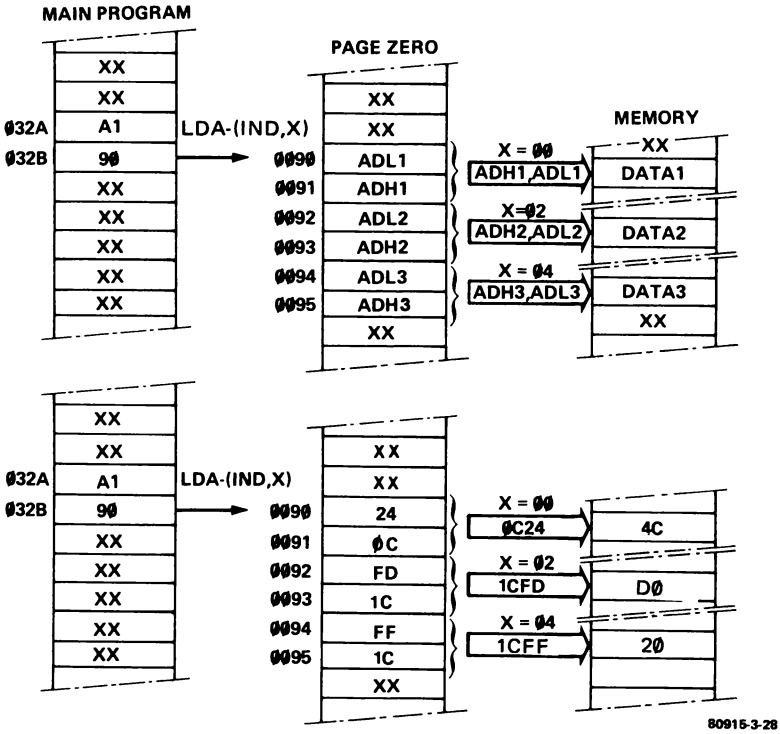


Bild 28. Ein Beispiel für Indexed Indirect Adressing. Bei diesem Adressierungsverfahren läßt sich Page Zero als Pointer Look up Table programmieren.

indirekte indizierte Adressierungsverfahren. Die Vorkenntnisse sind also vorhanden, so daß man leicht verstehen kann, was ein Interrupt ist.
Über Interrupts ist schon vieles geschrieben worden. Leider findet man in der Literatur nur ungenaue Beschreibungen, die noch dazu meist auf ein System bezogen sind. Um genau zu beschreiben, wozu Interrupts Verwendung finden, ist es am besten, diese mit bereits bekannten Adressierungsarten zu vergleichen. Sehen wir uns dazu Bild 29a an:
Wir sehen wieder den altvertrauten Sprung in eine Subroutine. Der Befehl, um vom Hauptprogramm in diese Subroutine zu kommen, ist JSR. Mit diesem Sprungbefehl ist es möglich, ein bestimmtes Unterprogramm im Speicher des Computers aufzurufen. Ist das Unterprogramm oder die Subroutine abgearbeitet, dann trifft der Mikroprozessor auf den Befehl RTS mit dem OP-Code 6Ø. Der Befehl RTS führt den Prozessor wieder zurück ins Hauptprogramm. Damit der Prozessor weiß, an welche Adresse er ins Hauptprogramm zurückkehren muß, hat dieser vor dem Sprung in die Subroutine den alten Programmzähler auf dem Stack abgelegt.
Ganz ähnlich läuft ein Interrupt ab! Wir springen jetzt nicht mehr mit dem Befehl JSR in eine Subroutine, sondern über einen IC-Pin des Mikroprozessors in eine Interruptroutine. Mit dem Befehl JSR sind wir *software-*

Bild 29b.

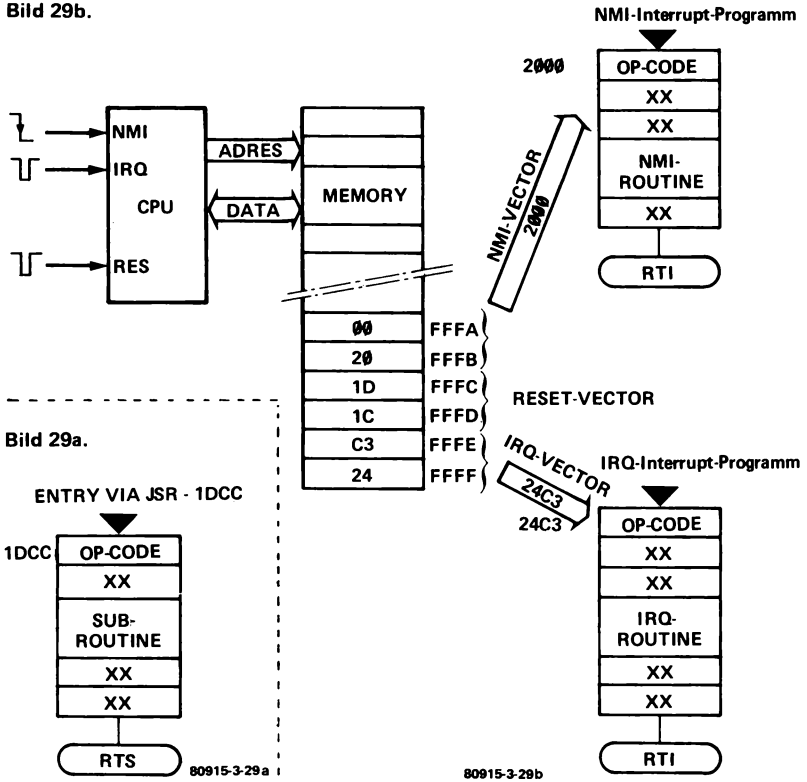


Bild 29. Hier sind der Sprung in eine Subroutine und der Sprung in eine Interruptroutine gegenübergestellt. Der IRQ- oder NMI-Vektor zeigt auf die Startadresse der jeweiligen Interruptroutine. Der Befehl RTI führt den Prozessor wieder ins Hauptprogramm zurück.

mäßig in ein Unterprogramm gesprungen. Mit einem Interrupt können wir jedoch *hardwaremäßig*, das heißt über ein von außen an den Prozessor angelegtes Digitalsignal in eine Subroutine, genannt Interruptroutine, springen (Bild 29b). Die 6502-CPU besitzt für den Sprung in eine Interruptroutine zwei Pins:
NMI-PIN = Non Maskable Interrupt
IRQ-Pin = Interrupt ReQest
Non Maskable Interrupt bedeutet: nicht maskierbarer Interrupt. Das heißt, liegt am NMI-Pin der CPU ein digitales Signal an, dann *muß* der Prozessor unbedingt in eine Interruptroutine springen. Die CPU muß also ein gerade laufendes Programm verlassen und den Interrupt bedienen.
Interrupt Request bedeutet: Interrupt Anforderung. Liegt am IRQ-Pin der CPU ein digitales Signal an, dann *kann* der Prozessor in eine Interruptroutine springen. Ob er nun in die Interruptroutine springt oder nicht,

entscheidet die Interrupt Flag im Prozessor Statusregister. Die Interrupt Flag läßt sich durch folgende Befehle setzen oder zurücksetzen:
CLI, OP-Code 58 I = Ø Interrupt Enable
SEI, OP-Code 78 I = 1 Interrupt Disable
Ist die Interrupt Flag im Prozessor Statusregister zurückgesetzt, also I = Ø, nimmt die CPU über den IRQ-Pin einen Interrupt an. Das heißt, liegt am IRQ-Pin ein entsprechendes digitales Signal, führt der Mikroprozessor den Interrupt aus. Ist aber die Interrupt Flag gesetzt, also I = 1, dann ignoriert der Prozessor das am IRQ-Pin anliegende Signal und führt keinen Interrupt aus. Man sagt auch, der Interrupt, der über den IRQ-Pin der CPU ausgelöst wird, läßt sich mit der Interrupt Flag im Prozessor Statusregister maskieren. Außer, daß sich ein Interrupt, der über den IRQ-Pin ausgelöst wird, maskieren läßt, und ein Interrupt, der über den NMI-Pin ausgelöst wird, nicht maskieren läßt, haben beide Interrupt-Pins einen weiteren grundsätzlichen Unterschied:
Der NMI-Pin ist *flankensensitiv* während der IRQ-Pin *niveausensisitiv* ist. Das sollte unbedingt Beachtung finden, wenn man über einen solchen Pin in eine Interruptroutine springen möchte. Was macht aber nun die CPU, nachdem an einem der beiden Interrupt-Pins ein Interrupt auslösendes Signal ankommt? Betrachten wir zuerst den NMI-Pin. Eine negative Flanke an diesem Eingang löst einen Non Maskable Interrupt aus. Wie wir bereits wissen, muß die CPU diesen Interrupt sofort annehmen, darf sich also nicht um die Interrupt Flag im Prozessor Statusregister kümmern. Nach dem Eintreffen der negativen Flanke (TTL-Pegel) muß der Prozessor in die Interruptroutine geleitet werden. Wie läuft dieser Vorgang ab? Um nun in die Interruptroutine zu kommen, sieht der Mikroprozessor in den Speicherzellen FFFA und FFFB nach, an welcher Stelle im Speicher die Interruptroutine beginnt. An der Adresse FFFA befindet sich das niederwertige Adreßbyte der Interruptroutinen-Startadresse und an der Adresse FFFB das hochwertige Adreßbyte. In diesen Speicherzellen befinden sich in unserem Beispiel folgende Daten:
FFFA ØØ niederwertiges Adreßbyte
FFFB 2Ø hochwertiges Adreßbyte

Aus diesen beiden Bytes baut sich jetzt der Mikroprozessor den NMI-Vektor zusammen. Dabei ist zu bemerken, daß unter Vektor und Adreßpointer dasselbe zu verstehen ist. Wie beim Indirect Indexed Addressing der Adreßpointer in Page Zero abgelegt ist, so sind die Interrupt Vektoren am Ende von Page FF abgelegt!
Der NMI-Vektor zeigt in unserem Beispiel auf die Adresse 2ØØØ. An dieser Adresse steht der erste OP-Code der NMI-Routine. Diese Routine arbeitet jetzt der Mikroprozessor in der gewohnten Weise wieder ab. Nebenbei sei bemerkt, daß von der NMI-Routine auch Programmverzweigungen in Subroutinen stattfinden können, so wie wir früher in einer Subroutine mehrere andere Subroutinen angesprungen haben. Wie weiß nun der Prozessor, daß die Interruptroutine zu Ende ist? Ähnlich wie der Befehl RTS die CPU von der Subroutine wieder ins Hauptprogramm zurückgeführt hat, führt der Befehl RTI den Prozessor wieder an die richtige Programmstelle zurück. RTI hat den OP-Code 4Ø.
Wie bereits gesagt, ist der IRQ-Eingang der CPU niveausensitiv. Legt man an diesen Eingang für eine bestimmte Zeit ein log. Ø-Signal an, dann wird

ein Interrupt Request ausgelöst. (Wielange der IRQ-Pin auf Masse gezogen werden muß, damit die CPU einen IRQ erkennt, ist an dieser Stelle noch nicht von Bedeutung. Das erklären wir noch an einer anderen Stelle.) Sobald die CPU einen Interrupt Request erkannt hat, sieht sie im Prozessor Statusregister nach, ob die I-Flag gesetzt oder zurückgesetzt ist. Ist die Interrupt Flag I=Ø, dann führt die CPU den Interrupt aus, das heißt, sie springt in die IRQ-Routine. Bei der Adresse, wo die IRQ-Routine beginnt, wird der Mikroprozessor wieder über einen Vektor geleitet, den IRQ-Vektor ist bei den Adressen FFFE und FFFF abgelegt. Bei diesen beiden Adressen befinden sich in unserem Beispiel folgende Daten:

FFFE C3 niederwertiges Adreßbyte
FFFF 24 hochwertiges Adreßbyte.

Aus diesen beiden Bytes baut sich jetzt der Mikroprozessor den IRQ-Vektor zusammen. Der IRQ-Vektor zeigt auf die Adresse 24C3. Bei dieser Adresse steht der erste OP-Code der IRQ-Routine. Dort verweilt der Prozessor so lange, bis er wieder auf den Befehl RTI trifft. RTI führt den Prozessor wieder ins Hauptprogramm zurück.

Eine Frage müssen wir bei der Betrachtung der Interrupts noch beantworten: Wie findet die CPU nach der Abarbeitung der Interruptroutine wieder an die richtige Stelle im Hauptprogramm zurück? Die Antwort ist denkbar einfach: Erinnern wir uns an den Befehl JSR. Trifft der Mikroprozessor auf diesen Befehl, dann rettet er die Rückkehradresse auf den Stack. Erst dann wird der Sprung in die Subroutine ausgeführt. Am Ende der Subroutine steht der Befehl RTS. Dieser Befehl veranlaßt die CPU, die zuvor auf dem Stack gerettete Rückkehradresse wieder zu holen, um die richtige Adresse im Hauptprogramm wiederzufinden.

Ähnlich läuft eine Interruptsequenz ab. Erkennt der Mikroprozessor einen Interrupt, dann rettet er wie beim JSR-Befehl das hoch- und niederwertige Adreßbyte der Rückkehradresse auf den Stack. Außer der Rückkehradresse wird beim Interrupt noch ein weiteres, sehr wichtiges CPU-Register automatisch auf den Stack gerettet: Das Prozessor Statusregister. Das hat zur Folge, daß nach der Rückkehr aus der Interruptroutine auch das Prozessor Statusregister wieder in seinen ursprünglichen Zustand vorhanden ist, wenn der Prozessor von der Interruptroutine zurückkehrt. Der Befehl, der die CPU am Ende der Interrupt Routine wieder ins Hauptprogramm zurückführt, ist RTI = ReTurn from Interrupt.

Zusammenfassend läßt sich zwischen dem JSR-Befehl und einer Interruptanforderung folgende Analogie angeben:

JSR-Befehl	**Interruptanforderung**
JSR: OP-Code 2Ø	negative Impulsflanke an NMI-Pin oder log. Ø – Niveau am IRQ-Pin
RTS: OP-Code 6Ø	RTI: OP-Code 4Ø

Den exakten Ablauf einer Interruptsequenz erläutern wir im folgenden Abschnitt. Hier sollte nur *allgemein* beschrieben werden, was ein Interrupt ist, und wie der Mikroprozessor auf eine Interruptanforderung reagiert. Im Bild 29b ist noch ein CPU-Anschluß auf einen Pin herausgeführt: RES. Dieser Anschluß ist der Reseteingang des Mikroprozessors. Führt man dem Junior-Computer die Versorgungsspannung zu, dann weiß der Mikroprozessor von sich aus nicht, an welcher Stelle im Monitorprogramm die Arbeit aufzunehmen ist. Wie sich ein Flipflop mit dem Reseteingang

zurücksetzen läßt, kann man die CPU mit dem Reseteingang ins Monitorprogramm leiten. Dieser Vorgang beginnt ähnlich wie eine Interruptsequenz. Wird der RES-Eingang für eine bestimmte Zeit log. Ø, erkennt die CPU eine Resetanforderung. Sie sieht dann in den Speicherzellen FFFC und FFFD nach, auf welche Adresse der Resetvektor zeigt. Beim Junior-Computer ist das immer die Adresse 1C1D. Bei dieser Adresse startet der Systemmonitor, das heißt der Programmteil, der die Scannung des Displays und die Abfrage des Keyboards übernimmt.
Eine weitere interessante Frage müssen wir noch beantworten: Wie kann die CPU aus den Speicherzellen FFFA . . . FFFF die Interrupt- und Resetvektoren holen, wenn der Junior-Computer für diese Adressen keinen Speicher hat? Betrachtet man das Schaltbild des Junior-Computers, dann läßt sich schnell feststellen, daß nur die Adreßleitungen AØ . . . A9 für die Adressierung von Speicherzellen im EPROM, RAM und PIA verwendet werden. Die CS-Signale werden von den Adreßleitungen A1Ø . . . A12 abgeleitet, während die Adreßleitungen A13 . . . A15 nicht dekodiert sind. Das Bitmuster, das auf den Adreßleitungen A13 . . . A15 steht, hat also für die Adressierung der Speicherelemente auf der Platine des Junior-Computers keine Auswirkungen. Deshalb ist kein Unterschied, ob die CPU den Resetvektor von den Adressen FFFC, FFFD oder von den Adressen 1FFC, 1FFD holt. Dasselbe gilt für die Interruptvektoren. Da man für die Funktionsweise des Junior-Computers nicht den ganzen Adreßbereich dekodieren muß, sind die Reset- und Interruptvektoren in Seite 1F im EPROM untergebracht. Das heißt, obwohl die CPU beim Holen der Vektoren die nicht vorhandene Page FF adressiert, bekommt sie die Vektoren von Seite 1F. Das ist allerdings nur für die Grundausstattung des Junior-Computers zutreffend. Bei einer Erweiterung des Computers ist darauf zu achten, daß die Vektoren an den richtigen Speicherzellen am Ende des Speicherbereiches, also in Seite FF abgelegt sind.

Detaillierte Betrachtung von IRQ und NMI

Die Interrupts IRQ und NMI haben denselben technischen Ablauf. Sie unterscheiden sich darin, daß der eine maskiert und der andere nicht maskiert werden kann. Sehen wir uns dazu das praktische Beispiel in Bild 30 an. Dort sind Ausschnitte aus dem gesamten adressierbaren Speicherbereich der CPU 6502 gezeichnet.
In Page 1 ist der Stack untergebracht. Das Programm, das der Prozessor gerade abarbeitet befindet sich in Page 3. Die Vektoren, über die der Prozessor in die IRQ- und NMI-Routine geführt wird, sind am Ende der Page FF abgelegt. Wie aus der Zeichnung ersichtlich ist, zeigt der NMI-Vektor auf die Adresse 2ØØØ und der IRQ-Vektor auf die Adresse 24C3. An diesen beiden Adressen beginnen die Interrupt Routinen. Nehmen wir jetzt an, der Prozessor arbeitet sich in Page 3 durch sein Programm. Das ist für ihn nichts besonderes, denn er muß nur Instruktion für Instruktion ausführen. Irgendwann gelangt er zur Adresse Ø343. An dieser Stelle trifft die CPU wieder auf einen OP-Code. Aus der Zeichnung geht weiter hervor, daß dieser OP-Code eine Länge von drei Bytes hat, denn ihm folgen zwei Operandenbytes. Es könnte aber genau so gut eine Instruktion mit einer Länge von einem oder zwei Bytes sein. Das würde nichts ändern.
Nehmen wir nun an, während sich die CPU um den ersten Operanden

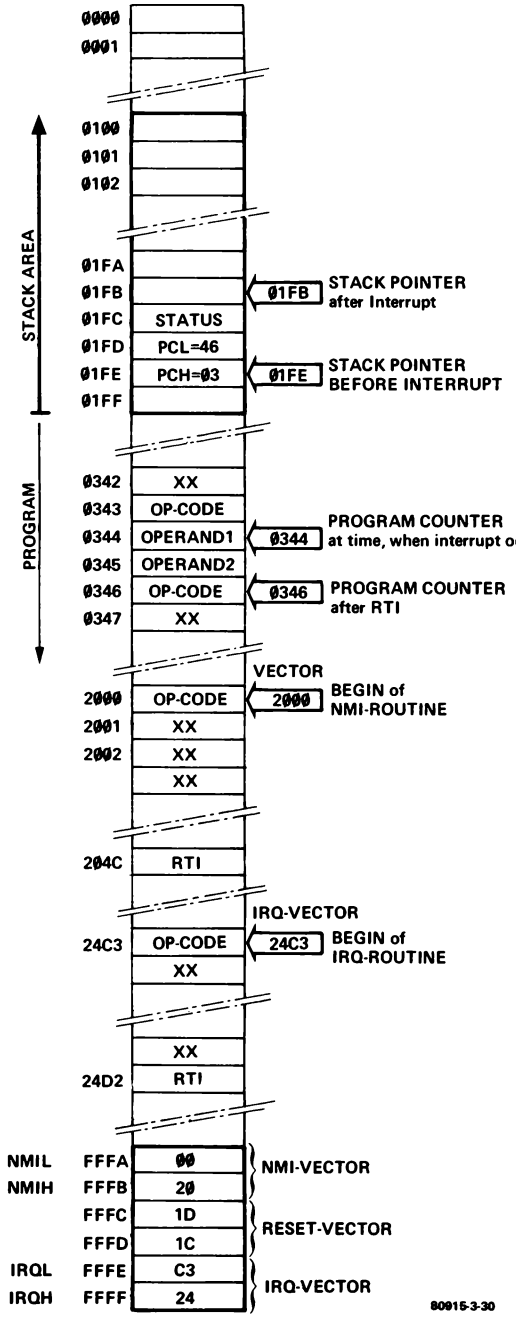


Bild 30. An Hand eines Speicherauszuges ist der Ablauf eines Interrupts leicht zu verfolgen. Auch ist gezeigt, wie der Programmzähler und das Prozessor Statusregister auf dem Stack abgelegt werden, nachdem der Interrupt erkannt ist.

kümmert, ereignet sich ein NMI. Das ist der Interrupt, um den sich die CPU sofort kümmern muß. Was geschieht nun? Die Antwort darauf ist, die CPU führt zuerst die Instruktion aus, die bei der Adresse Ø343 beginnt. Erst dann kümmert sie sich um den Interrupt. Ist die Instruktion ausgeführt, zeigt der Programmzähler auf den nächsten folgenden OP-Code, der an der Adresse Ø346 steht. Erst jetzt kümmert sich die CPU um die NMI-Anforderung. Der Ablauf der Interruptsequenz läßt sich in mehrere Abschnitte zerlegen, die wir nun besprechen:

1) Der Mikroprozessor hält den Programmzähler fest. Sein Inhalt ist die Adresse Ø346.
2) Der Mikroprozessor legt das hoch- und niederwertige Programmzählerbyte auf dem Stack ab. Da der Stack Pointer anfangs auf die Adresse Ø1FF zeigt, adressiert er jetzt die Speicherzelle Ø1FC.
3) Ist die Rückkehradresse der Interruptroutine auf dem Stack abgelegt, rettet die CPU das Prozessor Statusregister auf den Stack. Jetzt zeigt der Stack Pointer auf die Adresse Ø1FB.
4) In diesem Abschnitt holt der Prozessor aus den Speicherzellen FFFA und FFFB den NMI-Vektor und setzt diesen auf den Adreßbus. Die Startadresse 2ØØØ der NMI-Routine ist somit festgelegt.
5) Am Ende der NMI-Routine trifft der Prozessor auf den RTI-Befehl. Dieser Befehl führt ihn wieder ins Hauptprogramm zurück. Der alte Programmzähler und das Prozessor-Statusregister werden wieder in ihren ursprünglichen Zustand versetzt. Die Interruptsequenz ist somit abgeschlossen.

Was geschieht aber, wenn die soeben beschriebene Einleitung eines Interrupts nicht von einem NMI sondern von einem IRQ wird? Wie wir bereits wissen, kann ein IRQ nur dann ausgelöst werden, wenn vorher die I-Flag im Prozessor Statusregister zurückgesetzt wurde. Das heißt, die I-Flag muß Null sein, bevor die IRQ-Leitung auf log. Ø gezogen wird. Die IRQ-Sequenz läuft dann genauso ab, wie die zuvor beschriebene NMI-Sequenz. Auch hier werden wieder in der Reihenfolge PCH, PCL und das P-Register auf dem Stack abgelegt. Aber der Interruptvektor wird nicht wie zuvor von den Adressen FFFA und FFFB geholt, sondern von den Adressen FFFE und FFFF. An diesen Adressen ist nämlich das niederwertige und das hochwertige Byte des IRQ-Vektors abgelegt. Der IRQ-Vektor wird dann in gewohnter Weise auf den Adreßbus gegeben und der Prozessor somit in die IRQ-Routine geleitet.

Somit ist klar, daß während eines IRQs kein weiterer IRQ ausgelöst werden kann, da der Mikroprozessor am Anfang der Interruptsequenz die I-Flag automatisch setzt. Es ist aber sehr wohl möglich, daß während der Zeit, in der die CPU die IRQ-Routine abarbeitet, ein NMI ausgelöst wird, denn ein NMI muß ohne Rücksicht auf die I-Flag angenommen werden. In großen Computersystemen kann es sogar vorkommen, daß während einer IRQ-Routine mehrfach hintereinander NMIs ausgelöst werden. Diese mehrfachen NMIs führen den Mikroprozessor über *einen* NMI-Vektor zu *mehreren* NMI-Routinen. Doch wie das im einzelnen funktioniert, werden wir noch später erfahren, wenn wir den Jump Indirect, oder übersetzt: den indirekten Programmsprung diskutieren.

Merke: Wie innerhalb einer Subroutine mehrere verschiedene Subroutinen angesprungen werden können, lassen sich in einer Interruptroutine durch

Mehrfach-Interrupts verschiedene Interruptroutinen anspringen.
Häufig werden in einer Interruptroutine die CPU-Register Accumulator, X-Register und Y-Register benötigt. Da bei der Erkennung eines Interrupts nur das Prozessor Statusregister automatisch auf den Stack gerettet wird, müssen auch die übrigen Register per Software gerettet werden. Es ist daher empfehlenswert, eine Interruptroutine wie folgt einzuleiten:

```
SAVE   PHA     save Accu on Stack
       TXA  }
       PHA  }  save X-Register on Stack
       TYA  }
       PHA  }  save Y-Register on Stack
         .
         .
         .
         .
  Interruptroutine
         .
         .
         .
         .
RESTO  PLA  }
       TAY  }  restore Y-Register
       PLA  }
       TAX  }  restore X-Register
       PLA     restore Accu
END    RTI     return from interrupt
```

Wie diese Programmsequenz zeigt, werden am Anfang der Interruptroutine alle CPU-Register gerettet und am Ende wieder in ihren ursprünglichen Zustand gebracht. Das Wiederherstellen des ursprünglichen Zustandes aller CPU-Register erfolgt vor dem RTI-Befehl. Wie schon in diesem Abschnitt erwähnt, ist es möglich, durch einen NMI mehrere Interruptroutinen ineinander zu verschachteln. Wie ist das möglich, wenn nur ein NMI-Vektor, abgelegt in den Speicherzellen FFFA und FFFB, vorhanden ist? Möglich wird das durch einen indirekten Adreßsprung. Wie das vor sich geht, zeigen wir im nächsten Abschnitt.

Jump Indirect

Wie wir in früheren Kapiteln indirekte Lade- und Speicheroperationen durchführen, ist es auch möglich, Programme indirekt anzuspringen. Diesen Vorgang zeigt Bild 31. Dort sind bei den Adressen 1F2F und 1F32 die OP-Codes 6C abgelegt. Dieser OP-Code führt einen indirekten Programmsprung aus. Die symbolische Schreibweise ist:
JMP-(IND) OP-Code 6C
Ein indirekter Programmsprung hat eine Länge von 3 Bytes. Erinnern wir uns zuerst an den direkten Programmsprung: JMP OP-Code 4C. Wenn wir sagten, JMP-1CØØ oder 4C ØØ 1C, dann sprangen wir direkt an die Adresse 1CØØ, an der wieder ein OP-Code stand.
Etwas anders ist es beim indirekten Programmsprung. Wenn wir sagen JMP-(1A7A), dann sieht die CPU in der Speicherzelle 1A7A nach, was das niederwertige Adreßbyte der direkten Sprungadresse ist. Aus der folgenden

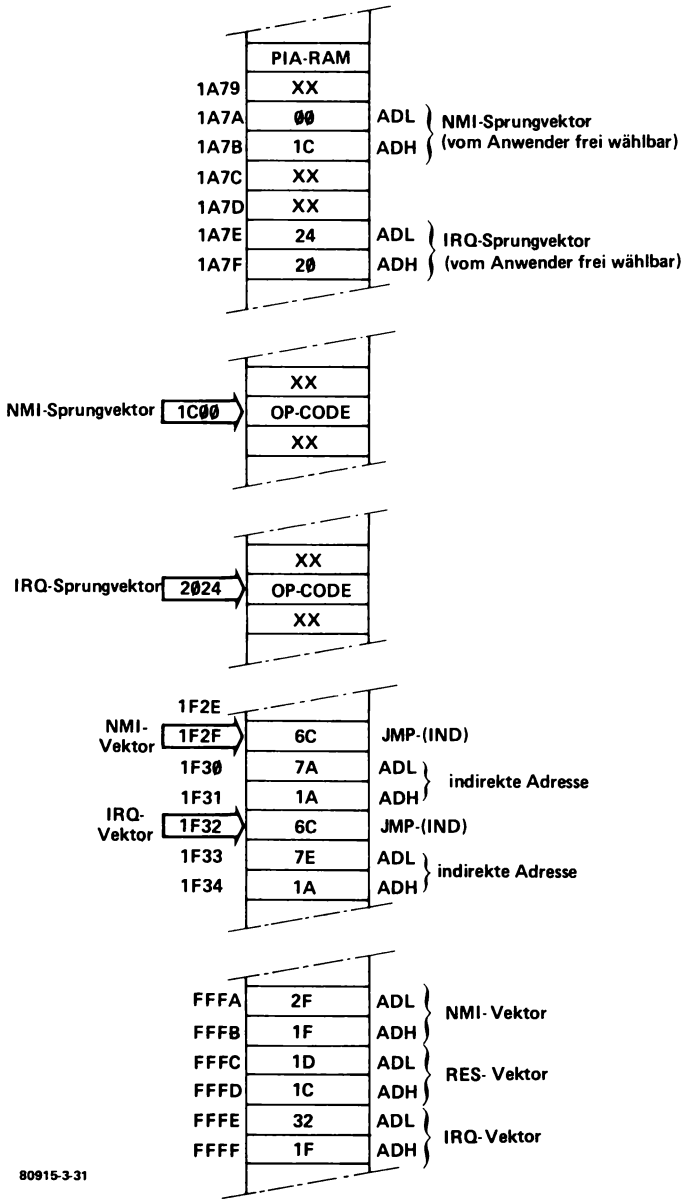


Bild 31. Zeigt der Interrupt Vektor auf eine JMP-(IND)-Instruktion, so lassen sich mit einem und demselben Vektor beliebig viele Interruptroutinen aufrufen. Der direkte Interrupt Vektor muß dann am Ende des PIO-RAMs geladen sein.

Speicherzelle, das ist 1A7B holt sie anschließend das hochwertige Adreßbyte der direkten Sprungadresse. Oder sagen wir anders: Beim indirekten Sprung springt der Mikroprozessor zu zwei aufeinanderfolgende Speicherzellen, aus denen er die direkte Sprungadresse holt. Aus den beiden Adreßbytes in diesen Speicherzellen "baut" er dann einen Sprungvektor zusammen. Dieser Sprungvektor zeigt auf die Adresse, zu der direkt gesprungen werden soll.
Beim Junior-Computer ist an den Adressen 1AØØ . . . 1A7F RAM als Speichermedium vorhanden. Das hat seinen besonderen Grund: In RAM-Zellen läßt sich bekanntlich der Inhalt beliebig verändern. Somit lassen sich auch die effektiven Sprungvektoren, die in den Speicherzellen 1A7A, 1A7B und 1A7E, 1A7F abgelegt sind, beliebig ändern. Genau das benötigen wir, wenn über nur *einen* NMI-Vektor *mehrere* NMI-Routinen bei verschiedenen Startadressen angesprungen werden sollen. Der Programmierer muß dann für folgendes sorgen:

- Bevor ein NMI eintrifft, muß in den Speicherzellen 1A7A, 1A7B ein Sprungvektor abgelegt sein. Dieser Sprungvektor zeigt auf die Startadresse einer Interruptroutine. Zu dieser Startadresse wird dann der Prozessor geführt, wenn ein NMI ausgelöst wird.
- Der NMI-Vektor muß immer auf einen JMP-(IND) OP-Code zeigen, wenn mit *einem* NMI oder IRQ *mehrere* Interruptroutinen, die mit verschiedenen Adressen beginnen, angesprungen werden sollen.

Bei der Programmierung der PIA zeigen wir noch an einem praktischen Beispiel, wie man mittels Programm den NMI-Vektor beliebig verändern kann.
Abschließend sei noch bemerkt, daß ein JMP-(IND) nicht nur im Zusammenhang mit Interrupts Verwendung finden muß: Es ist auch möglich, einen solchen Sprung anstatt eines direkten Sprungbefehles mit OP-Code 4C in ein Programm einzubauen. Das ist beim Testen von großen Programmen von Vorteil, da der Programmierer sein Programm in ein EPROM einbrennen kann, ohne die direkten Sprungadressen zu kennen. Somit lassen sich die direkten Sprungadressen für Testzwecke in RAM-Zellen ablegen und bei Bedarf modifizieren. Ist das Programm im EPROM schließlich getestet, dann können die JMP-(IND) OP-Codes durch normale JMP-Instruktionen ersetzt werden, da direkte wie indirekte Programmsprünge eine Länge von drei Bytes haben.

Das BRK-Kommando

Bisher beschrieben wir nur Interrupts, die extern durch logische Signale ausgelöst wurden. Das heißt, TTL-Signale haben wir an den NMI- oder IRQ-Eingang angelegt. Wir sagten, der Ansprung einer Interruptroutine erfolgte durch Hardware, die an den Junior-Computer angeschlossen war. Es ist bei der CPU 6502 jedoch auch möglich, durch Software einen Interrupt auszulösen. Der Befehl, der diesen Interrupt auslöst ist das BRK-Kommando. Sein OP-Code ist: BRK ——— ØØ. (BRK = BReaK = Unterbrechung).
Trifft der Mikroprozessor auf einen BRK-Befehl, reagiert er genauso, als würde extern ein IRQ-Signal anliegen, da das BRK-Kommando mit dem IRQ-Vektor zusammenarbeitet. Trifft der Mikroprozessor auf den

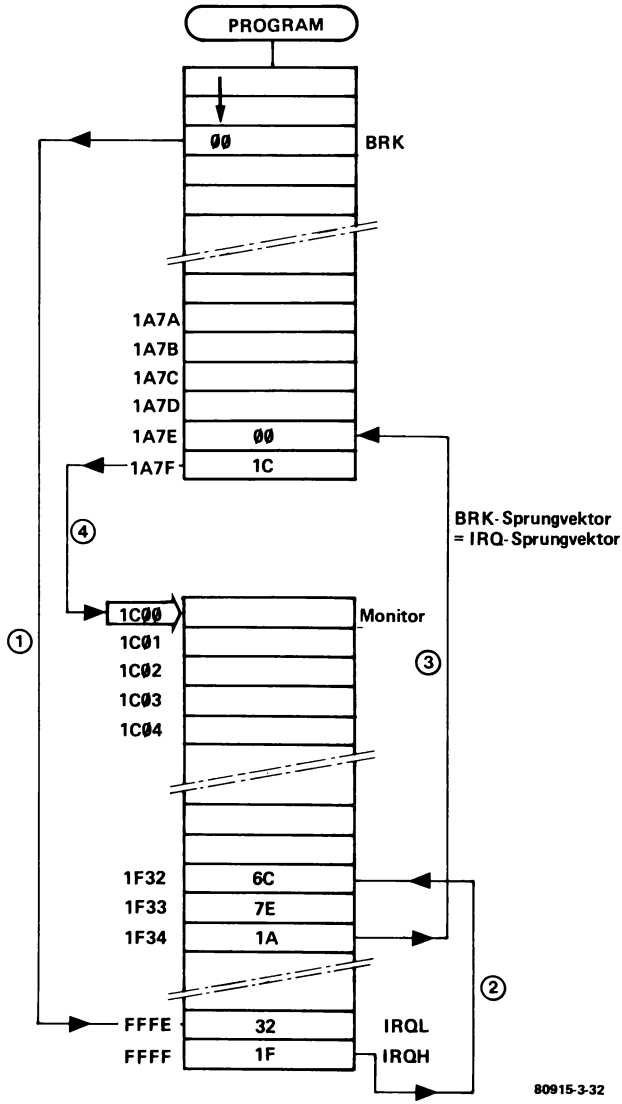


Bild 32. Das BRK-Kommando ist die Softwareversion eines IRQ. In der Standardversion des Junior-Computers sollte der BRK-Vektor (= IRQ-Vektor) immer auf die Adresse 1C00 zeigen. Dort befindet sich im Monitor eine Save-Routine, die den Inhalt aller CPU-Register rettet und anschließend in die Keyboard/Display-Routine verzweigt.

OP-Code ØØ, sieht er in den Speicherzellen FFFE und FFFF nach, zu welcher Programmstelle er springen soll. Das BRK-Kommando führt also einen indirekten Programmsprung aus. Der Unterschied zum indirekten Programmsprung JMP-(IND) mit OP-Code 6C läßt sich wie folgt angeben:

OP-Code	Befehl	Länge	Sprungvektor	Flag
6C	JMP-(IND)	3 Bytes	beliebig	keine
ØØ	BRK	1 Byte	IRQ-Vektor	B-Flag

Bild 32 zeigt ein praktisches Beispiel für ein BRK-Kommando. Der IRQ-Vektor, mit dem das BRK-Kommando zusammenarbeitet, zeigt auf die Adresse 1F32. Dort befindet sich der OP-Code 6C, der einen JMP-(IND) zur Folge hat. Dieser indirekte Programmsprung führt den Prozessor zur Adresse 1A7E und 1A7F. Diese beiden Adressen sind beim Junior-Computer die letzten Speicherzellen im RAM des PIA. Legt dort der Programmierer die Adresse 1CØØ ab, dann führt das BRK-Kommando den Prozessor ins Monitorprogramm des Junior-Computers. Demonstrationsprogramme in diesem Kapitel wurden häufig mit dem BRK-Kommando beendet. Mit Bild 32 wird klar, wie dieser Programmstop ablief.
Im Monitor des Junior-Computers zeigt der IRQ-Vektor auf einen JMP-(IND) Befehl. Da die indirekten Adressen RAM-Zellen sind, in denen die direkten Sprungadressen abgelegt sind, lassen sich mit nur einem Interrupt-Vektor beliebig viele Programmstellen anspringen. Wie bekannt, ist der IRQ-Vektor in den RAM-Zellen 1A7E und 1A7F abgelegt. Diese beiden Speicherzellen sind die indirekten Adressen, in denen der effektive IRQ-Vektor abgelegt ist. Da der effektive IRQ-Vektor in RAM-Zellen abgelegt ist, läßt sich dieser Vektor von Hand oder per Software modifizieren. Übertragen wir das auf das BRK-Kommando, so heißt das, daß mit einem und demselben BRK-Kommando mehrere BRK-Routinen angesprungen werden können, da auch das BRK-Kommando den IRQ-Vektor verwendet.
Somit sind wir am Ende des 3.Kapitels angekommen. Alle Adressierungsarten der CPU 6502 sind nun besprochen. Am Ende brachten wir etwas viel Theorie und nur wenige Übungsprogramme. Der Leser wird aber im folgenden Kapitel entschädigt. Dort sind Ergänzungen zu Kapitel 3 und weitere Übungsprogramme zu finden, die sicherlich sehr interessant sind.

4

Tips und Demonstrationsprogramme

Bevor wir weitere Programme beschreiben, sollen die folgenden Tips helfen, Programme auf den Junior-Computer zu übertragen:

1. Bevor ein Programm in den Junior-Computer getippt wird, sollte zuerst eine grobe (globale) und dann eine detaillierte Flow Chart angefertigt werden. Besonders bei größeren Programmen ist die grobe Flow Chart sehr wichtig, da sich mit ihr das Problem in Worte fassen läßt.
2. Bei Programmen mit bedingten Sprüngen sollten vor der Eingabe die Offsets berechnet werden. Das läßt sich einfach mit der Subroutine BRANCH (Startadresse 1FD5) am Ende des Systemmonitors durchführen. Dabei sind nur die niederwertigen Adreßbytes in den Computer einzugeben.
3. Programme laufen nur dann richtig, wenn sie korrekt im Speicherbereich des Junior-Computers abgelegt sind. Folgende Adressen stehen in der Standardausführung des Junior-Computers zur Verfügung:
   1K RAM mit den Adressen ØØØØ . . . Ø3FF. Das sind vier Seiten mit je 256 Bytes:
   Seite ØØ: ØØØØ . . . ØØFF
   Seite Ø1: Ø1ØØ . . . Ø1FF
   Seite Ø2: Ø2ØØ . . . Ø2FF
   Seite Ø3: Ø3ØØ . . . Ø3FF
   Von diesem zusammenhängenden Speicherbereich dürfen folgende Adressen vom Programmierer nicht verwendet werden, da diese für den Betriebsmonitor reserviert sind:
   In Seite ØØ: die Speicherzellen ØØE1 . . . ØØFF
   In Seite Ø1: die Speicherzellen Ø1F3 . . . Ø1FF
   Die letzten 13 Speicherzellen in Page 1 sind für den System Stack reserviert.
   In der Seite 1A liegt der PIA mit dem Timer. Auch 128 Bytes RAM stehen dort zur freien Verfügung: 1AØØ . . . 1A7F. Dabei sind die Adressen 1A7A, 1A7B sowie 1A7E, 1A7F für den NMI- beziehungsweise IRQ-Vektor vorgesehen und sollten vom Programmierer nicht mit einem Programm überschrieben werden.
4. Nach dem Aufruf des Monitorprogramms (RST-Taste) sollte der Programmierer immer in die Speicherzellen 1A7A, 1A7B sowie 1A7E, 1A7F den IRQ- und NMI-Vektor 1CØØ laden. Der Junior-Computer ist dann immer auf step by step mode und das BRK-Kommando vorbereitet:

	Tasten			Adresse	Daten	
RST				xxxx	xx	
AD				xxxx	xx	
Ø	2	Ø	Ø	Ø2ØØ	xx	
DA		2	Ø	Ø2ØØ	2Ø	JSR- CLEAR1
+		9	4	Ø2Ø1	94	ADL von CLDISP
+		Ø	2	Ø2Ø2	Ø2	ADH von CLDISP
+		2	Ø	Ø2Ø3	2Ø	JSR-
+		8	2	Ø2Ø4	82	ADL von CLB1
+		Ø	2	Ø2Ø5	Ø2	ADH von CLB1
+		2	Ø	Ø2Ø6	2Ø	JSR-
+		8	B	Ø2Ø7	8B	ADL von CLB2
+		Ø	2	Ø2Ø8	Ø2	ADH von CLB2
+		2	Ø	Ø2Ø9	2Ø	JSR- FIRST
+		6	F	Ø2ØA	6F	ADL von KEYDIS
+		Ø	2	Ø2ØB	Ø2	ADH von KEYDIS
+		C	9	Ø2ØC	C9	CMP #
+		1	Ø	Ø2ØD	1Ø	mit 1Ø
+		F	Ø	Ø2ØE	FØ	BEQ
+		F	Ø	Ø2ØF	FØ	FØ Schritte zurück ist CLEAR1
+		C	9	Ø21Ø	C9	CMP #
+		1	2	Ø211	12	mit 12
+		F	Ø	Ø212	FØ	BEQ
+		Ø	6	Ø213	Ø6	(Offset) Ø6 Schritte vorwärts ist PLUS
+		2	Ø	Ø214	2Ø	JSR-
+		4	9	Ø215	49	ADL von SHIFT
+		Ø	2	Ø216	Ø2	ADH von SHIFT
+		4	C	Ø217	4C	JMP- (unmittelbar)
+		Ø	9	Ø218	Ø9	ADL von FIRST
+		Ø	2	Ø219	Ø2	ADH von FIRST
+		2	Ø	Ø21A	2Ø	JSR- PLUS
+		A	A	Ø21B	AA	ADL von STO1
+		Ø	2	Ø21C	Ø2	ADH von STO1
+		2	Ø	Ø21D	2Ø	JSR-
+		9	4	Ø21E	94	ADL von CLDISP
+		Ø	2	Ø21F	Ø2	ADH von CLDISP
+		2	Ø	Ø22Ø	2Ø	JSR- SECOND
+		6	F	Ø221	6F	ADL von KEYDIS
+		Ø	2	Ø222	Ø2	ADH von KEYDIS
+		C	9	Ø223	C9	CMP #
+		1	Ø	Ø224	1Ø	mit 1Ø
+		F	Ø	Ø225	FØ	BEQ
+		Ø	A	Ø226	ØA	ØA Schritte vorwärts (Offset) ist CLEAR2
+		C	9	Ø227	C9	CMP #
+		1	1	Ø228	11	mit 11
+		F	Ø	Ø229	FØ	BEQ
+		Ø	C	Ø22A	ØC	(Offset); ØC Schritte vorwärts ist EQUAL
+		2	Ø	Ø22B	2Ø	JSR-
+		4	9	Ø22C	49	ADL von SHIFT
+		Ø	2	Ø22D	Ø2	ADH von SHIFT
+		4	C	Ø22E	4C	JMP-
+		2	Ø	Ø22F	2Ø	ADL von SECOND
+		Ø	2	Ø23Ø	Ø2	ADH von SECOND

+	2	Ø	Ø231	2Ø	JSR- CLEAR2
+	9	4	Ø232	94	ADL von CLDISP
+	Ø	2	Ø233	Ø2	ADH von CLDISP
+	4	C	Ø234	4C	JMP-
+	2	Ø	Ø235	2Ø	ADL von SECOND
+	Ø	2	Ø236	Ø2	ADH von SECOND
+	2	Ø	Ø237	2Ø	JSR- EQUAL
+	9	D	Ø238	9D	ADL von STO2
+	Ø	2	Ø239	Ø2	ADH von STO2
+	2	Ø	Ø23A	2Ø	JSR-
+	5	9	Ø23B	59	ADL von ADD
+	Ø	2	Ø23C	Ø2	ADH von ADD
+	2	Ø	Ø23D	2Ø	JSR-
+	B	7	Ø23E	B7	ADL von RESDIS
+	Ø	2	Ø23F	Ø2	ADH von RESDIS
+	2	Ø	Ø24Ø	2Ø	JSR-
+	8	2	Ø241	82	ADL von CLB1
+	Ø	2	Ø242	Ø2	ADH von CLB1
+	2	Ø	Ø243	2Ø	JSR-
+	8	B	Ø244	8B	ADL von CLB2
+	Ø	2	Ø245	Ø2	ADH von CLB2
+	4	C	Ø246	4C	JMP-
+	Ø	9	Ø247	Ø9	ADL von FIRST
+	Ø	2	Ø248	Ø2	ADH von FIRST
+	A	Ø	Ø249	AØ	LDY #; Subroutine SHIFT
+	Ø	4	Ø24A	Ø4	Ø4 in Y-Indexregister
+	Ø	6	Ø24B	Ø6	ASLZ SHIFT1
+	F	9	Ø24C	F9	INH mit Adresse ØØF9
+	2	6	Ø24D	26	ROLZ
+	F	A	Ø24E	FA	POINTL mit Adresse ØØFA
+	2	6	Ø24F	26	ROLZ
+	F	B	Ø25Ø	FB	POINTH mit Adresse ØØFB
+	8	8	Ø251	88	DEY dekrementiere Y mit 1
+	D	Ø	Ø252	DØ	BNE
+	F	7	Ø253	F7	F7 Schritte zurück ist SHIFT1
+	Ø	5	Ø254	Ø5	ORAZ
+	F	9	Ø255	F9	Bit für Bit ODER-Funktion mit INH
+	8	5	Ø256	85	STAZ
+	F	9	Ø257	F9	Accuinhalt zu INH (adr ØØF9)
+	6	Ø	Ø258	6Ø	RTS
+	F	8	Ø259	F8	SED dezimal rechnen! Subroutine ADD
+	1	8	Ø25A	18	CLC
+	A	5	Ø25B	A5	LDAZ
+	Ø	Ø	Ø25C	ØØ	ADL von B1Ø (Adresse ØØØØ); B1Ø in Accu
+	6	5	Ø25D	65	ADCZ
+	Ø	3	Ø25E	Ø3	ADL von B2Ø (Adresse ØØØ3); Accu = B1Ø + B2Ø
+	8	5	Ø25F	85	STAZ
+	Ø	6	Ø26Ø	Ø6	ADL von RØ; Accu → RØ
+	A	5	Ø261	A5	LDAZ
+	Ø	1	Ø262	Ø1	ADL von B11 (Adresse ØØØ1); B11 in Accu
+	6	5	Ø263	65	ADCZ

```
+   Ø   4   Ø264  Ø4   ADL von B21 (Adresse ØØØ4);
                       Accu = B11 + B21
+   8   5   Ø265  85   STAZ
+   Ø   7   Ø266  Ø7   ADL von R1 (Adresse ØØØ7);
                       Accu → R1
+   A   5   Ø267  A5   LDAZ
+   Ø   2   Ø268  Ø2   ADL von B12 (Adresse ØØØ2);
                       B12 in Accu
+   6   5   Ø269  65   ADCZ
+   Ø   5   Ø26A  Ø5   ADL von B22 (Adresse ØØØ5);
                       Accu = B12 + B22
+   8   5   Ø26B  85   STAZ
+   Ø   8   Ø26C  Ø8   ADL von R2 (Adresse ØØØ8);
                       Accu → R2
+   D   8   Ø26D  D8   CLD binär rechnen!
+   6   Ø   Ø26E  6Ø   RTS
+   2   Ø   Ø26F  2Ø   JSR-Subroutine  KEYDIS
+   8   E   Ø270  8E   ADL von SCANDS } in Monitor
+   1   D   Ø271  1D   ADH von SCANDS } (1D8E)
+   D   Ø   Ø272  DØ   BNE
+   F   B   Ø273  FB   (Offset); FB Schritte zurück ist
                       KEYDIS
+   2   Ø   Ø274  2Ø   JSR-  KD
+   8   E   Ø275  8E   ADL von SCANDS } in Monitor
+   1   D   Ø276  1D   ADH von SCANDS } (1D8E)
+   F   Ø   Ø277  FØ   BEQ
+   F   B   Ø278  FB   (Offset); FB Schritte zurück ist KD
+   2   Ø   Ø279  2Ø   JSR-
+   8   E   Ø27A  8E   ADL von SCANDS } in Monitor
+   1   D   Ø27B  1D   ADH von SCANDS } (1D8E)
+   F   Ø   Ø27C  FØ   BEQ
+   F   6   Ø27D  F6   (Offset); F6 Schritte zurück ist KD
+   2   Ø   Ø27E  2Ø   JSR-
+   F   9   Ø27F  F9   ADL von GETKEY } in Monitor
+   1   D   Ø28Ø  1D   ADH von GETKEY } (1DF9)
+   6   Ø   Ø281  6Ø   RTS
+   A   9   Ø282  A9   LDA #; Subroutine CLB1
+   Ø   Ø   Ø283  ØØ   ØØ → Accu
+   8   5   Ø284  85   STAZ
+   Ø   Ø   Ø285  ØØ   Accu → BIØ (= ØØ)
+   8   5   Ø286  85   STAZ
+   Ø   1   Ø287  Ø1   ØØ → B11
+   8   5   Ø288  85   STAZ
+   Ø   2   Ø289  Ø2   ØØ → B12
+   6   Ø   Ø28A  6Ø   RTS
+   A   9   Ø28B  A9   LDA #; Subroutine CLB2
+   Ø   Ø   Ø28C  ØØ   ØØ → Accu
+   8   5   Ø28D  85   STAZ
+   Ø   3   Ø28E  Ø3   ØØ → B2Ø
+   8   5   Ø28F  85   STAZ
+   Ø   4   Ø29Ø  Ø4   ØØ → B21
+   8   5   Ø291  85   STAZ
+   Ø   5   Ø292  Ø5   ØØ → B22
+   6   Ø   Ø293  6Ø   RTS
+   A   9   Ø294  A9   LDA #; Subroutine CLDISP
```

+	0	0	0295	00	00 → Accu
+	8	5	0296	85	STAZ
+	F	9	0297	F9	00 → INH (Adresse 00F9)
+	8	5	0298	85	STAZ
+	F	A	0299	FA	00 → POINTL (Adresse 00FA)
+	8	5	029A	85	STAZ
+	F	B	029B	FB	00 → POINTH (Adresse 00FB)
+	6	0	029C	60	RTS
+	A	5	029D	A5	LDAZ; Subroutine STO2
+	F	9	029E	F9	INH (Adresse 00F9) → Accu
+	8	5	029F	85	STAZ
+	0	3	02A0	03	Accu (= INH) → B20 (Adresse 0003)
+	A	5	02A1	A5	LDAZ
+	F	A	02A2	FA	POINTL (Adresse 00FA) → Accu
+	8	5	02A3	85	STAZ
+	0	4	02A4	04	Accu (= POINTL) → B21 (Adresse 0004)
+	A	5	02A5	A5	LDAZ
+	F	B	02A6	FB	POINTH (Adresse 00FB) → Accu
+	8	5	02A7	85	STAZ
+	0	5	02A8	05	Accu (= POINTH) → B22 (Adresse 0005)
+	6	0	02A9	60	RTS
+	A	5	02AA	A5	LDAZ; Subroutine STO1
+	F	9	02AB	F9	INH (Adresse 00F9) → Accu
+	8	5	02AC	85	STAZ
+	0	0	02AD	00	Accu (= INH) → B10 (Adresse 0000)
+	A	5	02AE	A5	LDAZ
+	F	A	02AF	FA	POINTL (Adresse 00FA) → Accu
+	8	5	02B0	85	STAZ
+	0	1	02B1	01	Accu (= POINTL) → B11 (Adresse 0001)
+	A	5	02B2	A5	LDAZ
+	F	B	02B3	FB	POINTH (Adresse 00FB) → Accu
+	8	5	02B4	85	STAZ
+	0	2	02B5	02	Accu (= POINTH) → B12 (Adresse 0002)
+	6	0	02B6	60	RTS
+	A	5	02B7	A5	LDAZ; Subroutine RESDIS
+	0	6	02B8	06	R0 (Adresse 0006) → Accu
+	8	5	02B9	85	STAZ
+	F	9	02BA	F9	Accu (= R0) → INH (Adresse 00F9)
+	A	5	02BB	A5	LDAZ
+	0	7	02BC	07	R1 (Adresse 0007) → Accu
+	8	5	02BD	85	STAZ
+	F	A	02BE	FA	Accu (= R1) → POINTL (Adresse 00FA)
+	A	5	02BF	A5	LDAZ
+	0	8	02C0	08	R2 (Adresse 0008) → Accu
+	8	5	02C1	85	STAZ
+	F	B	02C2	FB	Accu (= R2) → POINTH (Adresse 00FB)
+	6	0	02C3	60	RTS

Bild 1. Tastenprogramm des dezimalen Additionsprogramms aus Kapitel 3. 196 Bytes nimmt dieses Programm ein. Das detaillierte Flußdiagramm ist in den Bildern 20 sowie 21a . . . 21i in Kapitel 3 gezeigt.

RST		AD		XXXX XX
1	A	7	A	1A7A XX
DA		Ø	Ø	1A7A ØØ
+		1	C	1A7B 1C
+				1A7C XX
+				1A7D XX
+		Ø	Ø	1A7E ØØ
+		1	C	1A7F 1C

Jetzt steht nichts mehr im Wege, das Additionsprogramm von Kapitel 3 (Bild 20 und Bild 21) in den Junior-Computer einzugeben. Die Startadresse dieses Programms ist Ø2ØØ, das Tastenprogramm zeigt Bild 1. In der folgenden Übersicht sind die wichtigsten Labels und Programmteile zusammengestellt:

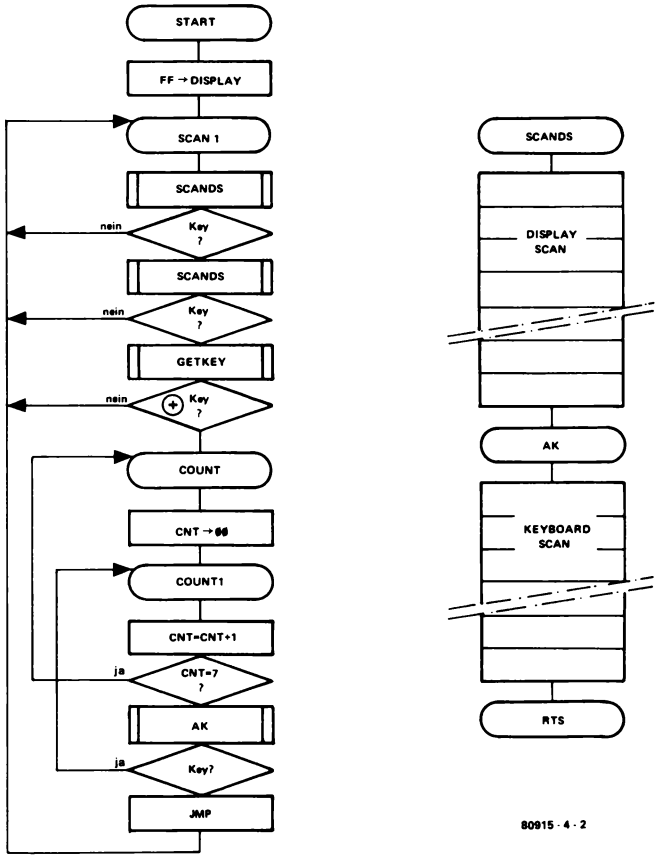


Bild 2. Das globale Flußdiagram des "Software-Würfels". Bei diesem Programm werden verschiedene Subroutinen des Systemmonitors aufgerufen.

Adresse Ø2ØØ ... Ø248: Hauptprogramm aus Bild 2Ø
Adresse Ø249 ... Ø258: Subroutine SHIFT aus Bild 21a
Adresse Ø259 ... Ø26E: Subroutine ADD aus Bild 21b
Adresse Ø26F ... Ø281: Subroutine KEYDIS aus Bild 21c
Adresse Ø282 ... Ø28A: Subroutine CLB1 aus Bild 21d
Adresse Ø28B ... Ø293: Subroutine CB2 aus Bild 21e
Adresse Ø294 ... Ø29C: Subroutine CLDISP aus Bild 21f
Adresse Ø29D ... Ø2A9: Subroutine STO2 aus Bild 21g
Adresse Ø2AA ... Ø2B6: Subroutine STO1 aus Bild 21h
Adresse Ø2B7 ... Ø2C3: Subroutine RESDIS aus Bild 21i

Dann müssen noch die Offsets der bedingten Programmverzweigungen berechnet werden. Dazu starten wir wieder die Subroutine BRANCH mit der Adresse 1FD5 und geben die niederwertigen Adreßbytes von Start- und Zieladresse der Verzweigung ein:

RST	AD			XXXX	XX	
1	F	D	5	1FD5	D8	
GO				ØØØØ	ØØ	
Ø	E	Ø	Ø	ØEØØ	FØ	FØ auf Adresse Ø2ØF
1	2	1	A	121A	Ø6	Ø6 auf Adresse Ø213
2	5	3	1	2531	ØA	ØA auf Adresse Ø226
2	9	3	7	2937	ØC	ØC auf Adresse Ø22A
5	2	4	B	524B	F7	F7 auf Adresse Ø253
7	2	6	F	726F	FB	FB auf Adresse Ø273
7	7	7	4	7774	FB	FB auf Adresse Ø278
7	C	7	4	7C74	F6	F6 auf Adresse Ø27D

RST

Nebenbei sei bemerkt, daß durch Drücken einer beliebigen Funktionstaste während der Routine BRANCH das Display auf ØØØØØØ (= Reset) gesetzt werden kann.

Nachdem das gesamte Additionsprogramm in den Junior-Computer eingetippt ist, läßt es sich mit der GO-Taste starten. Danach ist der Computer für die Zahleneingabe bereit. Die Addition für einige Zahlenbeispiele läßt sich dann wie folgt durchführen:

AD	Ø 2 Ø Ø	Startadresse eingeben	
GO	ØØØØØØ	Programmstart	
2	4 5 6	ØØ2456	1. Zahl
+	1 3 2	ØØØØØØ	+
4	1 3 2	ØØ4132	2. Zahl
DA	(= [=])	ØØ6588	Ergebnis
AD	(= [CLEAR])	ØØØØØØ	Reset
1	9 8 5 3 1	198531	1. Zahl
+		ØØØØØØ	+
8	3 2 7 Ø 2	832702	2. Zahl
DA		Ø31233	Ergebnis
AD		ØØØØØØ	Reset

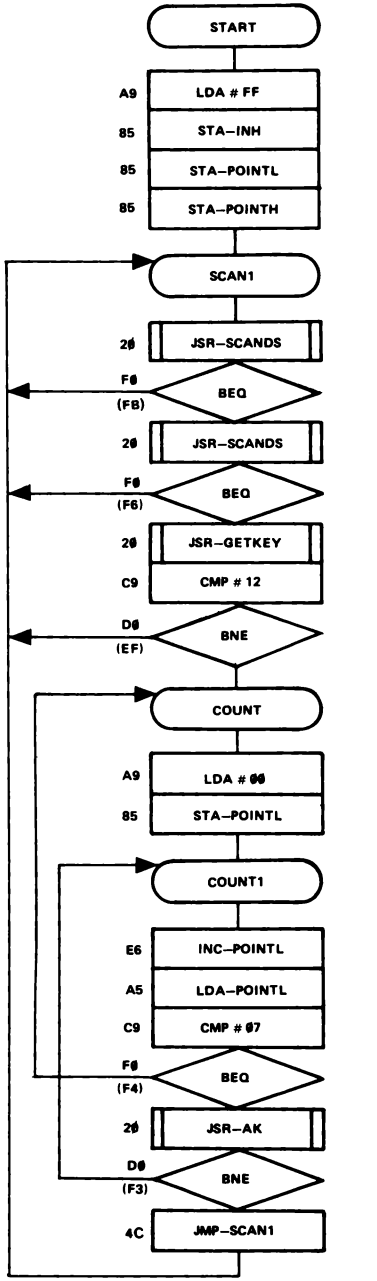


Bild 3. Das detaillierte Flußdiagram des "Software-Würfels".

Die Hauptroutine des Additionsprogramms ist so ausgelegt, daß falsch eingetippte Zahlen sofort korrigiert werden können. Dazu muß nur die CLEAR Taste gedrückt werden. Mit der CLEAR Taste wird die zuletzt eingegebene Zahl gelöscht und der Junior-Computer ist für eine neue Zahleneingabe bereit.

Würfeln mit dem Junior-Computer

Nachdem wir nun gelernt haben, wie die Subroutinen SCANDS, GETKEY und AK in Programme einzubauen sind, fällt es nicht schwer, den Junior-Computer als Würfel zu "mißbrauchen". Doch davor sind noch ein paar Überlegungen anzustellen. Bekanntlich hat ein Würfel sechs Augen. Mit einem Software-Zähler, der von 1 ... 6 zählen kann, lassen sich alle Würfelaugen darstellen. Durch das Drücken der +-Taste soll der Würfelvorgang eingeleitet werden. Solange diese Taste gedrückt ist, soll das Display erlöschen und nach dem Loslassen der Taste soll FF XX FF im Display erscheinen. XX repräsentiert das Würfelergebnis.
Bild 2 zeigt den groben Ablauf des Würfelprogramms. Zunächst werden die Displaybuffer mit $FF geladen und durch die Subroutine SCANDS im Display abgezeigt. Diese Subroutine tastet in gewohnter Weise das Keyboard ab und stellt dabei fest, ob eine Taste niedergedrückt ist. Ist eine Taste betätigt, dann dekodiert die Subroutine GETKEY die gedrückte Taste. Nach der Erkennung der gedrückten Taste fragt das Programm ab, ob die +-Taste gedrückt wurde. Nur die +-Taste wird akzeptiert und jede andere Taste ignoriert.
Der Software-Zähler, der den Würfel nachbildet, startet beim Label COUNT. Zunächst setzt das Programm den Inhalt des Zählers auf Null um ihn anschließend sofort zu inkrementieren. In einer Korrekturschleife fragt dann der Mikrocomputer ab, ob der Inhalt des Zählers bereits Sieben ist. Ist dieser unerlaubte Zählerzustand erreicht, dann setzt das Programm den Zähler wieder auf Eins und ein neuer Zählzyklus kann beginnen.
Jetzt muß nur noch festgesetzt werden, wie lange gezählt werden soll. Solange die +Taste gedrückt ist, soll gezählt werden und — wie zuvor gefordert — das Display erlöschen. Dazu rufen wir die Subroutine AK auf. Diese Subroutine schließt sich direkt an die Subroutine SCANDS an und testet, ob eine Taste gedrückt ist. AK steuert *nicht* das 7-Segment-Display an. Bild 2 zeigt, wie die Subroutinen SCANDS und AK im Systemmonitor des Junior-Computers zusammenhängen.
Aber zurück zum Würfel! Solange die +Taste gedrückt ist, verweilt die CPU in der Zählerschleife COUNT. Da in dieser Programmschleife nur die Subroutine AK aufgerufen wird, bleibt das Display dunkel. Erst wenn die +Taste losgelassen wird, springt der Mikroprozessor nach SCAN1. Dort trifft er wieder auf die Subroutine SCANDS und zeigt im Display die gewürfelte Zahl an.
Bild 3 zeigt die detaillierte Flow Chart des Würfelprogramms. Die Startadressen der einzelnen Subroutinen sind dort ebenfalls angegeben. Das Programm startet wieder bei der Adresse Ø2ØØ und läßt sich mit dem Keyboard leicht in den Junior-Computer eingeben (Bild 4).
Betrachtet man die Programme, die bisher geschrieben wurden, so erscheinen sie beim ersten Hinsehen sinnlos! Ein digitaler Würfel läßt sich einfacher und billiger mit ein paar TTL- oder MOS-ICs aufbauen. Dazu

	Tasten			Adresse	Daten	
RST		AD		xxxx	xx	
Ø	2	Ø	Ø	Ø2ØØ	xx	
DA		A	9	Ø2ØØ	A9	A9 LDA # **START**
+		F	F	Ø2Ø1	FF	FF → Accu
+		8	5	Ø2Ø2	85	STA Z
+		F	9	Ø2Ø3	F9	FF → INH (ØØF9)
+		8	5	Ø2Ø4	85	STA Z
+		F	A	Ø2Ø5	FA	FF → POINTL (ØØFA)
+		8	5	Ø2Ø6	85	STA Z
+		F	B	Ø2Ø7	FB	FF → POINTH (ØØFB)
+		2	Ø	Ø2Ø8	2Ø	JSR- **SCAN1**
+		8	E	Ø2Ø9	8E	ADL } von SCANDS (Monitor)
+		1	D	Ø2ØA	1D	ADH } (Adresse 1D8E)
+		F	Ø	Ø2ØB	FØ	BEQ
+		F	B	Ø2ØC	FB	(Offset); FB Schritte zurück nach
+		2	Ø	Ø2ØD	2Ø	JSR-
+		8	E	Ø2ØE	8E	ADL } von SCANDS (Monitor)
+		1	D	Ø2ØF	1D	ADH } (Adresse 1D8E)
+		F	Ø	Ø21Ø	FØ	BEQ
+		F	6	Ø211	F6	F6 Schritte zurück nach SCAN1
+		2	Ø	Ø212	2Ø	JSR-
+		F	9	Ø213	F9	ADL } von GETKEY (Monitor)
+		1	D	Ø214	1D	ADH } (Adresse 1DF9)
+		C	9	Ø215	C9	CMP #
+		1	2	Ø216	12	mit 12
+		D	Ø	Ø217	DØ	BNE
+		E	F	Ø218	EF	EF Schritte zurück nach SCAN1
+		A	9	Ø219	A9	LDA # **COUNT**
+		Ø	Ø	Ø21A	ØØ	ØØ → Akku
+		8	5	Ø21B	85	STA Z
+		F	A	Ø21C	FA	ØØ → POINTL (ØØFA)
+		E	6	Ø21D	E6	INC Z **COUNT1**
+		F	A	Ø21E	FA	POINTL + 1 → POINTL
+		A	5	Ø21F	A5	LDA Z
+		F	A	Ø22Ø	FA	POINTL → Accu
+		C	9	Ø221	C9	CMP #
+		Ø	7	Ø222	Ø7	mit Ø7
+		F	Ø	Ø223	FØ	BEQ
+		F	4	Ø224	F4	F4 Schritte zurück nach COUNT
+		2	Ø	Ø225	2Ø	JSR-
+		B	1	Ø226	B1	ADL } von AK (Monitor)
+		1	D	Ø227	1D	ADH } (Adresse 1DB1)

```
+          D  Ø   Ø228  DØ  BNE
+          F  3   Ø229  F3  F3 Schritte zurück nach COUNT1
+          4  C   Ø22A  4C  JMP-
+          Ø  8   Ø22B  Ø8  ADL }
+          Ø  2   Ø22C  Ø2  ADH } von SCAN1
AD
Ø     2    Ø  Ø   Ø2ØØ  A9  Startadresse
GO                          Programmstart
+                FFØ4  FF  Der Würfel ist gefallen
+                FFØ1  FF
+                FFØ6  FF
+                FFØ2  FF
usw.
```

Bild 4. Das Tastenprogramm des "Software-Würfels" von Bild 3.

benötigt man keinen Computer. Aber dieses Thema steht hier nicht zur Debatte. Die beschriebenen Programme haben ausschließlich einen edukativen Charakter.
Merke: Subroutinen bringen Übersicht in das Hauptprogramm. Der Ablauf des Hauptprogramms läßt sich somit leicht verfolgen und spätere Modifikationen können leicht eingefügt werden.

Die Subroutinen LENACC (Berechnung der OP-Code-Länge)

Die Subroutine LENACC (Bild 6) ist in jedem Assembler und Disassembler zu finden. Was ein Assembler bzw. ein Disassembler ist, werden wir genau im Buch 2 besprechen. Hier genügt es zu wissen, daß das Schreiben eines Assemblers bzw. Disassemblers viel Programmiererfahrungen und -praxis voraussetzt. Warum stellen wir dann die Subroutine LENACC vor? Ganz einfach: Um zu zeigen, daß wir mit dem erworbenen Wissen aus Kapitel 3 bereits recht intelligente Programme schreiben bzw. verfolgen können.
Was macht nun die Subroutine LENACC? Sie setzt einen beliebigen OP-Code, der im Accu der CPU steht, in eine Bytelänge um und speichert die Längeninformation in der Speicherzelle BYTES.
Wie wir bereits wissen, lassen sich die einzelnen OP-Codes der CPU 6502 in drei Gruppen einteilen:
1. OP-Codes, die *ein* Byte lang sind
2. OP-Codes, die *zwei* Bytes lang sind
3. OP-Codes, die *drei* Bytes lang sind.

Der OP-Code 2Ø = JSR hat eine Länge von 3 Bytes. Wenn wir diesen OP-Code in den Accu laden und anschließend die Subroutine LENACC aufrufen, dann "berechnet" der Mikroprozessor in dieser Subroutine die Länge des OP-Codes 2Ø, der 3 Bytes lang ist. Schreiben wir das in gewohnter Weise an, dann sieht das so aus:

```
LDA #2Ø      lade JSR-OP-Code in den Accu
JSR LENACC   berechne die Länge der JSR-Instruktion
BRK          unterbreche das Programm
```

linke Hexadezimalziffer	rechte Hexadezimalziffer								
	0	1	2	3	4	5	6	7
0	BRK (1)	ORA (IND,X) (2)				ORA Z (2)	ASL Z (2)	
1	BPL (2)	ORA (IND),Y (2)				ORA Z,X (2)	ASL Z,X (2)	
2	JSR (3)	AND (IND,X) (2)			BIT Z (2)	AND Z (2)	ROL Z (2)	
3	BMI (2)	AND (IND),Y (2)				AND Z,X (2)	ROL Z,X (2)	
4	RTI (1)	EOR (IND,X) (2)				EOR Z (2)	LSR Z (2)	
5	BVC (2)	EOR (IND),Y (2)				EOR Z,X (2)	LSR Z,X (2)	
6	RTS (1)	ADC (IND,X) (2)				ADC Z (2)	ROR Z (2)	
7	BVS (2)	ADC (IND),Y (2)				ADC Z,X (2)	ROR Z,X (2)	
8		STA (IND,X) (2)			STY Z (2)	STA Z (2)	STX Z (2)	
9	BCC (2)	STA (IND),Y (2)			STY Z,X (2)	STA Z,X (2)	STX Z,Y (2)	
A	LDY # (2)	LDA (IND,X) (2)	LDX # (2)		LDY Z (2)	LDA Z (2)	LDX Z (2)	
B	BCS (2)	LDA (IND),Y (2)			LDY Z,X (2)	LDA Z,X (2)	LDX Z,Y (2)	
C	CPY # (2)	CMP (IND,X) (2)			CPY Z (2)	CMP Z (2)	DEC Z (2)	
D	BNE (2)	CMP (IND),Y (2)				CMP Z,X (2)	DEC Z,X (2)	
E	CPX # (2)	SBC (IND,X) (2)			CPX Z (2)	SBC Z (2)	INC Z (2)	
F	BEQ (2)	SBC (IND),Y (2)				SBC Z,X (2)	INC Z,X (2)	

Bild 5. OP-Code Tabelle der CPU 6502. Die Spalteninformation, das ist das niederwertige Nibble des OP-Codes, spielt beim Programm LENACC eine wichtige Rolle. In dieser Routine wird der OP-Code in eine Instruktionslänge umgesetzt.

Bevor wir die Subroutine LENACC aufrufen ist der Inhalt der Speicherzelle BYTES beliebig (BYTES = XX = don't care). Nach dem Verlassen der Subroutine steht in der Speicherzelle BYTES die Länge des OP-Codes 2Ø: BYTES = Ø3!
Hätten wir einen anderen OP-Code, zum Beispiel A9 = LDA # in den Accu geladen, bevor wir die Subroutine LENACC anspringen, dann stünde in der Speicherzelle BYTES Ø2, da der OP-Code A9 eine Länge von zwei Bytes hat.
In Bild 5 sind in einer Tabelle alle OP-Codes der CPU 6502 zusammengefaßt. Beim ersten Hinsehen erscheint es fast unmöglich, aus dieser Tabelle die OP-Code-Länge der verschiedenen OP-Codes zu erkennen. Dazu betrachten wir die erste Spalte der OP-Code-Tabelle. Die einzelnen OP-Codes in dieser Tabelle liegen zwischen ØØ . . . FØ. Schreiben wir diese Spalte genauer an, dann erhalten wir folgendes Ergebnis:

OP-Code	Mnemonic	Länge
ØØ	BRK	1 Byte
1Ø	BPL	2 Bytes
2Ø	JSR	3 Bytes
3Ø	BMI	2 Bytes
4Ø	RTI	1 Byte
5Ø	BVC	2 Bytes
6Ø	RTS	1 Byte
7Ø	BVS	2 Bytes
9Ø	BCC	2 Bytes
AØ	LDY#	2 Bytes
BØ	BCS	2 Bytes
CØ	CPY#	2 Bytes
DØ	BNE	2 Bytes
EØ	CPX#	2 Bytes
FØ	BEQ	2 Bytes

Wie wir aus dieser Übersicht leicht ersehen können, sind in der ersten Spalte der OP-Code Tabelle Instruktionen mit einer Länge von einem, zwei

rechte Hexadezimalziffer

linke Hexadezimalziffer	8	9	A	B	C	D	E	F	
Ø	PHP (1)	ORA # (2)	ASL A (1)			ORA ABS (3)	ASL ABS (3)		Ø
1	CLC (1)	ORA ABS,Y (3)				ORA ABS,X (3)	ASL ABS,X (3)		1
2	PLP (1)	AND # (2)	ROL A (1)		BIT ABS (3)	AND ABS (3)	ROL ABS (3)		2
3	SEC (1)	AND ABS,Y (3)				AND ABS,X (3)	ROL ABS,X (3)		3
4	PHA (1)	EOR # (2)	LSR A (1)		JMP ABS (3)	EOR ABS (3)	LSR ABS (3)		4
5	CLI (1)	EOR ABS,Y (3)				EOR ABS,X (3)	LSR ABS,X (3)		5
6	PLA (1)	ADC # (2)	ROR A (1)		JMP IND (3)	ADC ABS (3)	ROR ABS (3)		6
7	SEI (1)	ADC ABS,Y (3)				ADC ABS,X (3)	ROR ABS,X (3)		7
8	DEY (1)		TXA (1)		STY ABS (3)	STA ABS (3)	STX ABS (3)		8
9	TYA (1)	STA ABS,Y (3)	TXS (1)			STA ABS,X (3)			9
A	TAY (1)	LDA # (2)	TAX (1)		LDY ABS (3)	LDA ABS (3)	LDX ABS (3)		A
B	CLV (1)	LDA ABS,Y (3)	TSX (1)		LDY ABS,X (3)	LDA ABS,X (3)	LDX ABS,Y (3)		B
C	INY (1)	CMP # (2)	DEX (1)		CPY ABS (3)	CMP ABS (3)	DEC ABS (3)		C
D	CLD (1)	CMP ABS,Y (3)				CMP ABS,X (3)	DEC ABS,X (3)		D
E	INX (1)	SBC # (2)	NOP (1)		CPX ABS (3)	SBC ABS (3)	INC ABS (3)		E
F	SED (1)	SBC ABS,Y (3)				SBC ABS,X (3)	INC ABS,X (3)		F

oder drei Bytes. Da die Häufigkeit der Instruktionen mit einer Länge von 2 Bytes überwiegen, können wir sagen: Instruktionen die 1 Byte oder 3 Bytes lang sind, bilden in dieser Tabelle eine Ausnahme! Diese Ausnahmen müssen wir in der Subroutine LENACC "herausfiltern". Doch darauf kommen wir noch später zu sprechen.

In der zweiten Spalte der Operation Code Table finden wir 16 Instruktionen der CPU 6502 untereinander gelistet. Alle Instruktionen in dieser Spalte haben als Adressierungsart Indirect Indexed,Y oder Indexed Indirect,X. Wie wir aus Kapitel 3 wissen, haben alle Instruktionen, die sich dieser Adressierungsarten bedienen, eine Länge von 2 Bytes. Wir können also feststellen: OP-Codes, deren niederwertiges Nibble "1" ist, haben immer eine Länge von 2 Bytes. Die Instruktionen in der zweiten Spalte haben folgende OP-Codes:

Ø1, 11, 21, 31, 41, 51, 61, 71, 81, 91, A1, B1, C1, E1, F1. In der dritten Spalte der Operation Code Table finden wir nur eine Instruktion: LDY#. Der OP-Code ist A2. Wir können also feststellen, OP-Codes, deren niederwertiges Nibble "2" ist, haben eine Länge von 2 Bytes.

In den Spalten 4, 8, 12 und 16 befinden sich keine OP-Codes. In der fünften Spalte befinden sich OP-Codes, die als Adressierungsart Zero Page oder Zero Page,X Addressing haben. Wie wir aus Kapitel 3 wissen, haben alle Instruktionen, die sich dieser Adressierungsart bedienen, eine Länge von 2 Bytes. Wir können also feststellen: OP-Codes, deren niederwertiges Nibble "4" ist, haben eine Länge von 2 Bytes.

Die Instruktionen in der fünften Spalte haben folgende OP-Codes: 24, 84, 94, A4, B4, C4, E4.

In der sechsten Spalte finden wir wieder 16 Instruktionen der CPU 6502. Alle Instruktionen in dieser Spalte haben als Adressierungsart Zero Page oder Zero Page,X Addressing. Instruktionen, die sich dieser Adressierungsart bedienen, haben eine Länge von 2 Bytes. Wir können also feststellen: OP-Codes, deren niederwertiges Nibble "5" ist, haben eine Länge von 2 Bytes. Die Instruktionen in der sechsten Spalte haben folgende OP-Codes:

Ø5, 15, 25, 35, 45, 55, 65, 75, 85, 95, A5, B5, C5, D5, E5, F5.

In der siebten Spalte finden wir wieder OP-Codes, für die dasselbe gilt, wie für die OP-Codes in der sechsten Spalte. Sie haben eine Länge von 2 Bytes und haben als Adressierungsart Zero Page, Zero Page,X oder Zero Page,Y.

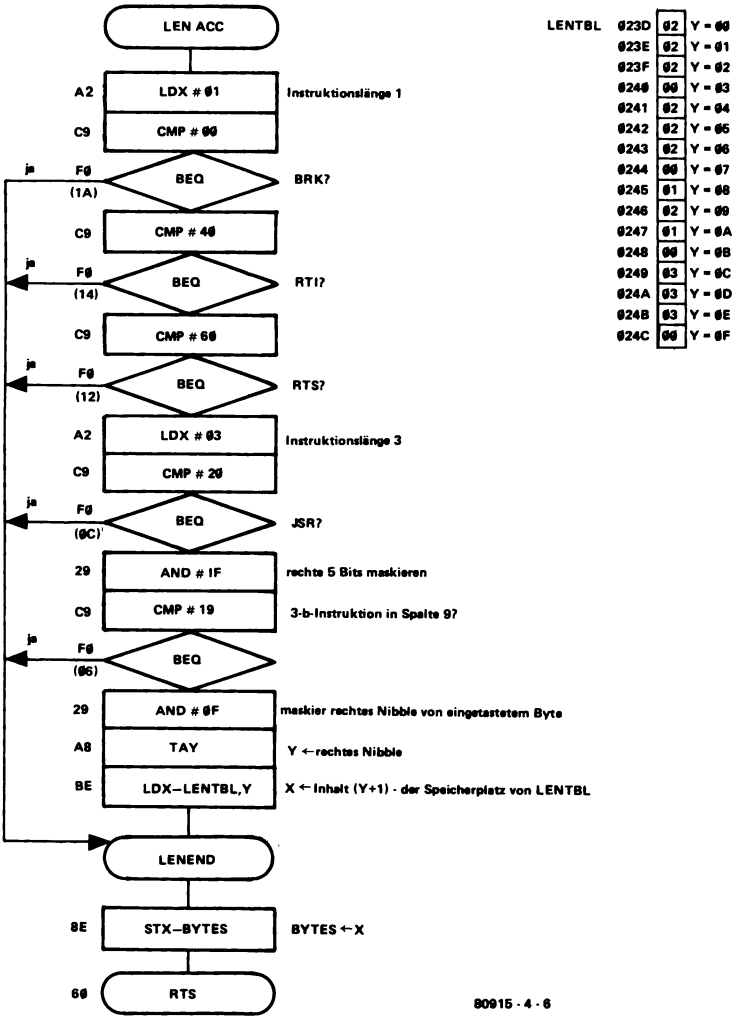


Bild 6. Die Subroutine LENACC berechnet für jede Instruktion der 6502 CPU die Länge. Bekanntlich sind die Instruktionen ein, zwei oder drei Bytes lang. Illegale Instruktionen werden in die Instruktionslänge 00 umgesetzt.

Wir können also wieder sagen: OP-Codes, deren niederwertiges Nibble "6" ist, haben eine Länge von 2 Bytes. Die Instruktionen in der siebten Spalte haben folgende OP-Codes:
06, 16, 26, 36, 46, 56, 76, 86, 96, A6, B6, C6, D6, E6, F6.
In der neunten Spalte finden wir weitere 16 Instruktionen. Alle Instruktionen in dieser Spalte haben als Adressierungsart Implied Addressing.

Instruktionen die sich dieser Adressierungsart bedienen, sind 1 Byte lang. Wir können also sagen: OP-Codes, deren niederwertigen Nibble "8" ist, haben eine Länge von 1 Byte. Die Instruktionen in der neunten Spalte haben folgende OP-Codes:

Ø8, 18, 28, 38, 48, 58, 68, 78, 88, 98, A8, B8, C8, D8, E8, F8.

In der zehnten Spalte treffen wir wieder auf einige Unregelmäßigkeiten. Um das zu verdeutlichen, wollen wir die zehnte Spalte nochmals auflisten:

OP-Code	Mnemonic	Länge
Ø9	ORA#	2 Bytes
19	ORA-ABS,Y	3 Bytes
29	AND#	2 Bytes
39	AND-ABS,Y	3 Bytes
49	EOR#	2 Bytes
59	EOR-ABS,Y	3 Bytes
69	ADC#	2 Bytes
79	ADC-ABS,Y	3 Bytes
99	STA-ABS,Y	3 Bytes
A9	LDA#	2 Bytes
B9	LDA-ABS,Y	3 Bytes
C9	CMP#	2 Bytes
D9	CMP-ABS,Y	3 Bytes
E9	SBC#	2 Bytes
F9	SBC-ABS,Y	3 Bytes

Betrachtet man diese Tabelle global, so ist leicht festzustellen, daß zwei Adressierungsarten abwechselnd aufeinanderfolgen: Immediate Addressing und Absolute Indexed,Y Addressing. Immediate Addressing hat eine Länge von 2 Bytes während Absolute Indexed,Y Addressing eine Länge von 3 Bytes hat. Wie kann man nun in dieser Tabelle feststellen, welche Instruktionen 2 oder 3 Bytes lang sind? Am niederwertigen Nibble des OP-Codes kann nicht erkannt werden, ob die Instruktion 2 oder 3 Bytes lang ist. Aber im hochwertigen Nibble des OP-Codes ist die Längeninformation enthalten! Ist das hochwertige Nibble eine gerade Zahl (das niederwertigste Bit des hochwertigen Nibble ist dann Ø), so handelt es sich um eine Instruktion mit einer Länge von 2 Bytes oder eine Instruktion, die Immediate Addressing hat. Ist das hochwertige Nibble der Instruktion jedoch eine ungerade Zahl, (das niederwertigste Bit des hochwertigen Nibble ist dann 1) so handelt es sich um eine Instruktion mit einer Länge von 3 Bytes oder eine Instruktion, die Absolute Indexed,Y Addressing hat. Durch folgenden Programmablauf lassen sich die 2 Byte Befehle von 3 Byte Befehlen in der zehnten Spalte trennen:

```
LDY # ØØ
LDA - (POINTL),Y     lade den OP-Code in den Accu
AND # 1F             maskiere BitØ . . . Bit4
CMP # 19
BEQ XX               verzweige, wenn 3 Bytes lang
                     verzweige nicht, wenn 2 Bytes lang
```

Das Maskieren mit 1F hat zur Folge, daß bei den OP-Codes Ø9, 29, 49, 69, 89, A9, C9, E9 die Hexzahl Ø9 im Accu steht. Wenn aber Ø9 im Accu der CPU steht, so bedeutet das, die Instruktion hat eine Länge von 2 Bytes.
Bei den OP-Codes 19, 39, 59, 99, B9, D9, F9 steht nach dem Maskieren mit 1F die Hexzahl 19 im Accu. Das bedeutet, es muß sich um eine Instruktion mit Absolute Indexed,Y Addressing handeln. Diese Instruktion ist drei Bytes lang. Somit läßt sich durch das Maskieren mit AND#1F leicht feststellen, ob es sich um Immediate- oder Absolute Indexed,Y Addressing handelt.
Jetzt wird die Subroutine LENACC verständlich. Zuerst wird der OP-Code, dessen Länge zu bestimmen ist, in den Accu der CPU geladen. Dann detektiert das Programm, ob es sich um eine BRK-, eine RTI- oder eine RTS-Instruktion handelt. Diese Befehle haben eine Länge von 1 Byte und bilden die Ausnahme in Spalte 1. Durch das sequenzielle Abfragen mit: CMP#ØØ (BRK-Instruktion?), CMP#4Ø (RTI-Instruktion?) und CMP#6Ø (RTS-Instruktion?) werden die unregelmäßigen Befehle in der ersten Spalte von den regelmäßigen 2 Byte-Instruktionen getrennt.
Anschließend sieht das Programm nach, ob es sich um eine JSR-Instruktion handelt. Diese Instruktion ist 3 Bytes lang und ist ebenfalls eine Unregelmaäßigkeit in der ersten Spalte. Um diese Unregelmaßigkeit festzustellen, fährt das Programm den Vergleich CMP#2Ø durch. Somit sind alle Unregelmaßigkeiten in der ersten Spalte "ausgefiltert". Auch in der zehnten Spalte der Operation Code Table treffen wir wieder auf Unregelmaßigkeiten: Befehle mit einer Länge von 2 Bytes müssen von solchen mit einer Länge von 3 Bytes getrennt werden. Das geschieht, wie zuvor besprochen, durch die Programmsequenz:

```
AND#1F
CMP#19
BEQ LENEND
```

Ist jedoch am Anfang der Subroutine keine BRK-, keine RTI-, keine RTS-, keine JSR- und keine Instruktion, die ABS,Y Addressing hat, in den Accumulator geladen worden, kann es sich nur um eine regelmäßige Instruktion handeln. Wie schon zuvor gesagt, steckt die Längeninformation bei einer regelmäßigen Instruktion im niederwertigen Nibble des OP-Codes. Deshalb sind die Längen der regelmäßigen Instruktionen in einer Lookup Tabelle abgelegt. Um die Längeninformation von diesen Befehlen zu erhalten, ist folgende Programmsequenz nötig:

```
AND#ØF          erzeuge niederwertiges OP-Code Nibble
TAY             verwende Nibble als Index
LDX-LENTBL,Y    hole die Länge aus der Lookup Tabelle
STX-BYTES       lege die Länge in Bytes ab
```

Immer wenn die Subroutine verlassen wird, steht die Länge der Instruktion in der Speicherzelle Bytes. Die Lookup Tabelle LENTBL kann irgendwo im Speicherbereich abgelegt werden.
Merke: Codeumwandlungen lassen sich auf einfache Weise mit einer Lookup Tabelle durchführen. Bei der Subroutine LENACC wird der OP-Code mit einer Lookup Tabelle in eine Längeninformation umgewandelt.
Abschließend schreiben wir noch ein Programm, das die Subroutine LENACC verwendet. Dieses Programm zeigt auf dem 6-stelligen Display

des Junior-Computers den OP-Code und dessen Länge an. In den linken beiden Displays erscheint der eingetippte OP-Code und in den mittleren beiden Displays seine Länge. Die rechten beiden Displays zeigen immer FF an. Unmittelbar nach dem Start wird das Display zurückgesetzt (FFFFFF). Das grobe Flußdiagram der Instruktionslängen-Routine zeigt Bild 7. Am Anfang werden die Displaybuffer mit FF geladen. Dann verzweigt das Programm in den Systemmonitor des Junior-Computers, indem es in die Subroutine GETBYT verzweigt. Diese Subroutine steuert das Keyboard und das Display des Computers und wartet auf die Eingabe eines Datenbytes. Nach der Eingabe eines Datenbytes (Datentasten Ø . . . F) kehrt der Prozessor aus der Subroutine GETBYT mit den Datenbyte als Accuinhalt zurück. Sind nur die Datentasten Ø . . . F gedrückt worden, so ist nach der Rückkehr die N-Flag gesetzt. Bei der Betätigung einer Befehlstaste, wie AD, DA, + usw. kehrt die CPU mit zurückgesetzter N-Flag aus der Subroutine GETBYT zurück. Somit läßt sich leicht feststellen, ob auf dem Keyboard eine Daten- oder Befehlstaste gedrückt wurde.
Betätigte der Programmierer zwei Datentasten, so hat er in den Junior-Computer einen OP-Code eingegeben. Nach dem Verlassen der Subroutine GETBYT wird dieser OP-Code im Displaybuffer POINTH abgelegt. Anschließend verzweigt das Programm in die Subroutine LENACC. Dort berechnet es die Länge des soeben eingegebenen OP-Codes. Nach der Rückkehr von LENACC steht die Länge der Instruktion in BYTES und wird

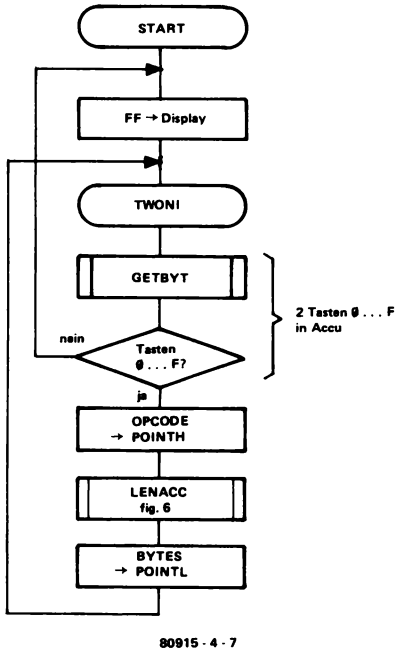


Bild 7. Globales Flußdiagramm der Instruktionslängen-Routine.

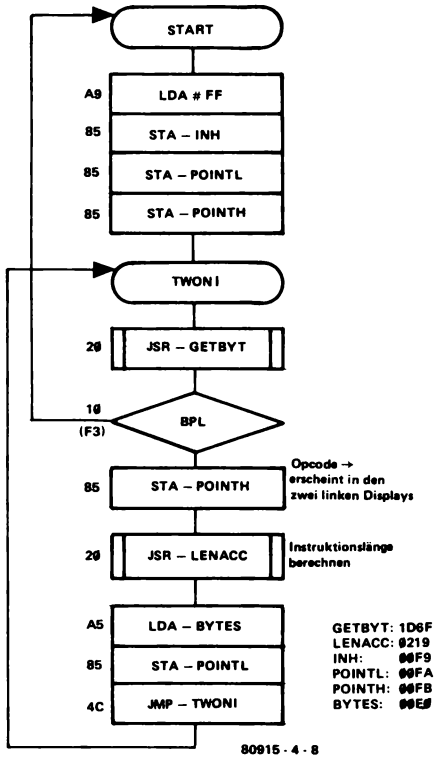


Bild 8. Das detaillierte Flußdiagramm der Instruktionslängen-Routine.

anschließend in den Displaybuffer POINTL kopiert. Dann verzweigt das Programm nach TWONI und wartet auf die Eingabe eines neuen OP-Codes. Wird aber in der Subroutine GETBYT eine Kommando-Taste betätigt, dann setzt das Programm das Display zurück und zeigt wie anfangs FFFFFF an. Der Programmierer kann nun wieder einen neuen OP-Code eingeben.
Bild 8 zeigt das detaillierte Flußdiagramm von Bild 7. Dort sind auch die Startadressen der Subroutinen und der verwendeten Bufferzellen angegeben. Wie sich das Programm aus Bild 8 mit der Subroutine LENACC in den Junior-Computer eingeben läßt, zeigt Bild 9. Die Umwandlung des OP-Codes in eine Instruktionslänge geschieht bekanntlich mit einer Lookup Table. Da in manchen Spalten von Bild 5 keine OP-Codes stehen, ist in der Lookup Table an diesen Stellen die Instruktionslänge ØØ abgelegt. Wird ein illegaler OP-Code eingegeben, so zeigt das Display als Instruktionslänge ØØ an. Das gilt jedoch nur für illegale OP-Codes der Spalten 3, 7, B und F.
Somit sind wir am Ende dieses Buches angelangt. Das nötige Rüstzeug für das Buch Junior-Computer 2 ist somit vorhanden. Dort wird das theo-

retische und praktische Wissen weiter verfeinert. Viele interessante Programme warten dort auf den Leser. Da in Buch 2 die Theorie klein, die Praxis jedoch groß geschrieben ist, dürfte es noch interessanter als Junior-Computer 1 sein. Aber Computer haben es an sich, daß man beim Erlernen des Programmierens besonders am Anfang in einen sauren Apfel beißen muß . . .

	Tasten			Adresse	Daten	
RTS	AD			xxxx	xx	
Ø	2	Ø	Ø	Ø2ØØ	xx	Startadresse
DA		A	9	Ø2ØØ	A9	LDA # START
+		F	F	Ø2Ø1	FF	FF → Accu
+		8	5	Ø2Ø2	85	STAZ
+		F	9	Ø2Ø3	F9	Accu → INH (Adresse ØØF9)
+		8	5	Ø2Ø4	85	STAZ
+		F	A	Ø2Ø5	FA	Accu → POINTL (Adresse ØØFA)
+		8	5	Ø2Ø6	85	STA Z
+		F	B	Ø2Ø7	FB	Accu → POINTH (Adresse ØØFB)
+		2	Ø	Ø2Ø8	2Ø	JSR- TWONI
+		6	F	Ø2Ø9	6F	ADL } von GETBYT (Monitor;
+		1	D	Ø2ØA	1D	ADH } Adresse 1D6F)
+		1	Ø	Ø2ØB	1Ø	BPL; 2 Tasten Ø . . . F betätigt?
+		F	3	Ø2ØC	F3	wenn nicht, F3 Schritte zurück (= Start)
+		8	5	Ø2ØD	85	STA Z (Byte in Accu)
+		F	B	Ø2ØE	FB	Accu (= OP-Code) → POINTH (Adresse ØØFB)
+		2	Ø	Ø2ØF	2Ø	JSR-
+		1	9	Ø21Ø	19	ADL } von LENACC
+		Ø	2	Ø211	Ø2	ADH }
+		A	5	Ø212	A5	LDAZ
+		E	Ø	Ø213	EØ	BYTES (Adresse ØØEØ) → Accu
+		8	5	Ø214	85	STAZ
+		F	A	Ø215	FA	Accu → POINTL (Adresse ØØFA)
+		4	C	Ø216	4C	JMP ABS
+		Ø	8	Ø217	Ø8	ADL } von Sprungadresse
+		Ø	2	Ø218	Ø2	ADH } (TWONI)
+		A	2	Ø219	A2	LDX #
						Subroutine LENACC
+		Ø	1	Ø21A	Ø1	Ø1 → X; Instruktionslänge 1
+		C	9	Ø21B	C9	CMP #
+		Ø	Ø	Ø21C	ØØ	mit ØØ
+		F	Ø	Ø21D	FØ	BEQ BRK?
+		1	A	Ø21E	1A	1A Schritte vor ist LENEND

```
+  C  9  021F  C9  CMP #
+  4  0  0220  40  mit 40
+  F  0  0221  F0  BEQ RTI?
+  1  6  0222  16  16 Schritte vorwärts ist LENEND
+  C  9  0223  C9  CMP #
+  6  0  0224  60  mit 60
+  F  0  0225  F0  BEQ RTS?
+  1  2  0226  12  12 Schritte vor ist LENEND
+  A  2  0227  A2  LDX #
+  0  3  0228  03  03 → X; Instruktionslänge 3
+  C  9  0229  C9  CMP #
+  2  0  022A  20  mit 20
+  F  0  022B  F0  BEQ JSR?
+  0  C  022C  0C  0C Schritte vor ist LENEND
+  2  9  022D  29  AND #
+  1  F  022E  1F  mit 1F, rechte 5 Bits maskieren
+  C  9  022F  C9  CMP #
+  1  9  0230  19  mit 19
+  F  0  0231  F0  BEQ; 3-b-Instruktion in Spalte 9
+  0  6  0232  06  06 Schritte weiter ist LENEND
+  2  9  0233  29  AND #
+  0  F  0234  0F  mit 0F; rechtes Nibble maskieren
+  A  8  0235  A8  TAY; rechtes Nibble → Y
+  B  E  0236  BE  LDX ABS,Y; (Y + 1) Speicherstelle
                  von LENTBL
+  3  D  0237  3D  ADL } von LENTBL
+  0  2  0238  02  ADH } (look up table)
+  8  E  0239  8E  STX ABS          LENEND
+  E  0  023A  E0  ADL BYTES
+  0  0  023B  00  ADH BYTES
+  6  0  023C  60  RTS zurück ins Hauptprogramm
+  0  2  023D  02  Spalte 0; Y = 00  LENTBL
+  0  2  023E  02  Spalte 1; Y = 01
+  0  2  023F  02  Spalte 2; Y = 02
+  0  0  0240  00  Spalte 3; Y = 03
+  0  2  0241  02  Spalte 4; Y = 04
+  0  2  0242  02  Spalte 5; Y = 05
+  0  2  0243  02  Spalte 6; Y = 06
+  0  0  0244  00  Spalte 7; Y = 07
+  0  1  0245  01  Spalte 8; Y = 08
+  0  2  0246  02  Spalte 9; Y = 09
+  0  1  0247  01  Spalte A; Y = 0A
+  0  0  0248  00  Spalte B; Y = 0B
```

+		Ø	3	Ø249	Ø3	Spalte C; Y = ØC
+		Ø	3	Ø24A	Ø3	Spalte D; Y = ØD
+		Ø	3	Ø24B	Ø3	Spalte E ; Y = ØE
+		Ø	Ø	Ø24C	ØØ	Spalte F; Y = ØF
AD						
Ø	2	Ø	Ø			Startadresse
GO						Programmstart
		A	9	A9Ø2	FF	
		Ø	3	Ø3ØØ	FF	
		D	2	D2Ø2	FF	
		9	E	9EØ3	FF	
		D	5	D5Ø2	FF	
		usw.				

Bild 9. Das Tastenprogramm für die Routinen in den Bildern 6 und 8.

1 OP-Codes in hexadezimaler Ordnung

00	BRK	20	JSR	40	RTI	60	RTS
01	ORA (IND,X)	21	AND (IND,X)	41	EOR (IND,X)	61	ADC (IND,X)
02	–	22	–	42	–	62	–
03	–	23	–	43	–	63	–
04	–	24	BIT Z	44	–	64	–
05	ORA Z	25	AND Z	45	EOR Z	65	ADC Z
06	ASL Z	26	ROL Z	46	LSR Z	66	ROR Z
07	–	27	–	47	–	67	–
08	PHP	28	PLP	48	PHA	68	PLA
09	ORA IMM	29	AND IMM	49	EOR IMM	69	ADC IMM
0A	ASL A	2A	ROL A	4A	LSR A	6A	ROR A
0B	–	2B	–	4B	–	6B	–
0C	–	2C	BIT ABS	4C	JMP ABS	6C	JMP IND
0D	ORA ABS	2D	AND ABS	4D	EOR ABS	6D	ADC ABS
0E	ASL ABS	2E	ROL ABS	4E	LSR ABS	6E	ROR ABS
0F	–	2F	–	4F	–	6F	–
10	BPL	30	BMI	50	BVC	70	BVS
11	ORA (IND), Y	31	AND (IND), Y	51	EOR (IND), Y	71	ADC (IND),Y
12	–	32	–	52	–	72	–
13	–	33	–	53	–	73	–
14	–	34	–	54	–	74	–
15	ORA Z,X	35	AND Z,X	55	EOR Z,X	75	ADC Z,X
16	ASL Z,X	36	ROL Z,X	56	LSR Z,X	76	ROR Z,X
17	–	37	–	57	–	77	–
18	CLC	38	SEC	58	CLI	78	SEI
19	ORA ABS,Y	39	AND ABS,Y	59	EOR ABS, Y	79	ADC ABS,Y
1A	–	3A	–	5A	–	7A	–
1B	–	3B	–	5B	–	7B	–
1C	–	3C	–	5C	–	7C	–
1D	ORA ABS,X	3D	AND ABS,X	5D	EOR ABS,X	7D	ADC ABS,X
1E	ASL ABS,X	3E	ROL ABS,X	5E	LSR ABS,X	7E	ROR ABS,X
1F	–	3F	–	5F	–	7F	–

80	–	A0	LDY IMM	C0	CPY IMM	E0	CPX IMM
81	STA (IND,X)	A1	LDA (IND,X)	C1	CMP (IND,X)	E1	SBC (IND,X)
82	–	A2	LDX IMM	C2	–	E2	–
83	–	A3	–	C3	–	E3	–
84	STY Z	A4	LDY Z	C4	CPY Z	E4	CPX Z
85	STA Z	A5	LDA Z	C5	CMP Z	E5	SBC Z
86	STX Z	A6	LDX Z	C6	DEC Z	E6	INC Z
87	–	A7	–	C7	–	E7	–
88	DEY	A8	TAY	C8	INY	E8	INX
89	–	A9	LDA IMM	C9	CMP IMM	E9	SBC IMM
8A	TXA	AA	TAX	CA	DEX	EA	NOP
8B	–	AB	–	CB	–	EB	–
8C	STY ABS	AC	LDY ABS	CC	CPY ABS	EC	CPX ABS
8D	STA ABS	AD	LDA ABS	CD	CMP ABS	ED	SBC ABS
8E	STX ABS	AE	LDX ABS	CE	DEC ABS	EE	INC ABS
8F	–	AF	–	CF	–	EF	–
90	BCC	B0	BCS	D0	BNE	F0	BEQ
91	STA (IND),Y	B1	LDA (IND),Y	D1	CMP (IND),Y	F1	SBC (IND),Y
92	–	B2	–	D2	–	F2	–
93	–	B3	–	D3	–	F3	–
94	STY Z,X	B4	LDY Z,X	D4	–	F4	–
95	STA Z,X	B5	LDA Z,X	D5	CMP Z,X	F5	SBC Z,X
96	STX Z,Y	B6	LDX Z,Y	D6	DEC Z,X	F6	INC Z,X
97	–	B7	–	D7	–	F7	–
98	TYA	B8	CLV	D8	CLD	F8	SED
99	STA ABS,Y	B9	LDA ABS,Y	D9	CMP ABS,Y	F9	SBC ABS,Y
9A	TXS	BA	TSX	DA	–	FA	–
9B	–	BB	–	DB	–	FB	–
9C	–	BC	LDY ABS,X	DC	–	FC	–
9D	STA ABS,X	BD	LDA ABS,X	DD	CMP ABS,X	FD	SBC ABS,X
9E	–	BE	LDX ABS,Y	DE	DEC ABS,X	FE	INC ABS,X
9F	–	BF	–	DF	–	FF	–

In dieser Tabelle sind die OP-Codes in hexadezimaler Ordnung (00 . . . FF) dargestellt. Auch die "Leer-OP-Codes" (zum Beispiel: 1A, 1B usw.) sind in dieser Tabelle aufgenommen. Dem OP-Code folgen die Mnemonics einer bestimmten Instruktion. Sie bestehen aus drei Buchstaben (zum Beispiel: ADC). Hinter den Mnemonics folgen noch weitere Buchstaben, die die Adressierungsart der Instruktion beschreiben. Ihre Bedeutung ist:

IM	Immediate Adressing (# = IM)
ABS	Absolute Addressing
Z	Zero Page Addressing
A	Accumulator Addressing
(IND,X)	Indexed Indirect Addressing
(IND),Y	Indirect Indexed Addressing
Z,X	Zero Page Indexed,X Addressing
Z,Y	Zero Page Indexed,Y Addressing
ABS,X	Absolute Indexed,X Addressing
ABS,Y	Absolute Indexed,Y Addressing
IND	Indirect Addressing

2 INSTRUKTIONS-Übersicht

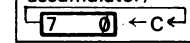
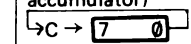
Die folgende Aufstellung zeigt die 56 Instruktionen des Mikroprozessors 6502 in alphabetischer Reihenfolge. Eine Anzahl verschiedener Befehle verfügt über mehrere Adressierungsmöglichkeiten, so daß insgesamt 151 verschiedene OP-Codes zur Verfügung stehen.

Mnemonics + Beschreibung	Adressierungs-methode(n)	OP-Code (hex)	Anzahl der Takt-impulse (N)	Anzahl der Bytes	von der Instruktion beeinflußte Flags
ADC Add memory to accumulator with carry A + M + C → A (1) (4)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	69 6D 65 61 71 75 7D 79	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	NV - - - - ZC
AND "AND" memory with accumulator A ∧ M → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	29 2D 25 21 31 35 3D 39	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
ASL Shift left one bit (accu or memory) C ← [7 0] ← 0	ABS Z A Z,X ABS,X	0E 06 0A 16 1E	6 5 2 6 7	3 2 1 2 3	N - - - - - ZC
BCC Branch on carry clear (2) Branch on C = 0	REL	90	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -
BCS Branch on carry set (2) Branch on C = 1	REL	B0	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -
BEQ Branch on result zero (2) Branch on Z = 1	REL	F0	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -
BIT Test bits in memory: A ∧ M M_7 → N; M_6 → V	ABS Z	2C 24	4 3	3 2	M_7M_6 - - - - Z -
BMI Branch on result minus (2) Branch on N = 1	REL	30	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -

Mnemonics + Beschreibung	Adressierungs-methode(n)	OP-Code (hex)	Anzahl der Takt-impulse (N)	Anzahl der Bytes	von der Instruktion beeinflußte Flags
BNE Branch on result not zero (2) Branch on Z = Ø	REL	DØ	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -
BPL Branch on result plus (2) Branch on N = Ø	REL	1Ø	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -
BRK Force break forced interrupt	IMP	ØØ	7	1	- - - 1 - 1 - - B I
BVC Branch on overflow clear (2) Branch on V = Ø	REL	5Ø	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -
BVS Branch on overflow set (2) Branch on V = 1	REL	7Ø	2 3, wenn Befehl ausgeführt wird	2	- - - - - - - -
CLC Clear carry flag Ø → C	IMP	18	2	1	- - - - - - - Ø C
CLD Clear decimal mode; Ø → D	IMP	D8	2	1	- - - - Ø - - - D
CLI Clear interrupt flag; Ø → 1, Enable IRQ	IMP	58	2	1	- - - - - Ø - - I
CLV Clear overflow flag; Ø → V	IMP	B8	2	1	Ø - - - - - - - V
CMP Compare memory and accumulator A-M	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	C9 CD C5 C1 D1 D5 DD D9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - ZC
CPX Compare memory and index X X-M	IMM ABS Z	E Ø EC E4	2 4 3	2 3 2	N - - - - - ZC
CPY Compare memory and index Y M-Y	IMM ABS Z	C Ø CC C4	2 4 3	2 3 2	N - - - - - ZC

Mnemonics + Beschreibung	Adressierungs-methode(n)	OP-Code (hex)	Anzahl der Takt-impulse (N)	Anzahl der Bytes	von der Instruktion beeinflußte Flags
DEC Decrement memory by one M-1 → M	ABS Z Z,X ABS,X	CE C6 D6 DE	6 5 6 7	3 2 2 3	N - - - - - Z -
DEX Decrement index X by one X-1 → X	IMP	CA	2	1	N - - - - - Z -
DEY Decrement index Y by one Y-1 → Y	IMP	88	2	1	N - - - - - Z -
EOR "Exclusive or" memory with accumulator A ∀ M → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	49 4D 45 41 51 55 5D 59	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
INC Increment memory by one M + 1 → M	ABS Z Z,X ABS,X	EE E6 F6 FE	6 5 6 7	3 2 2 3	N - - - - - Z -
INX Increment index X by one X + 1 → X	IMP	E8	2	1	N - - - - - Z -
INY Increment index Y by one Y + 1 → Y	IMP	C8	2	1	N - - - - - Z -
JMP Jump to new location (PC + 1) → PCL (PC + 2) → PCH	ABS IND	4C 6C	3 5	3 3	- - - - - - - -
JSR Jump to new location saving return address PC + 2 ↓ (PC + 1) → PCL (PC + 2) → PCH	ABS	20	6	3	- - - - - - - -

Mnemonics + Beschreibung	Adressierungs-methode(n)	OP-Code (hex)	Anzahl der Takt-impulse (N)	Anzahl der Bytes	von der Instruktion beeinflußte Flags
LDA Load accumulator with memory M → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	A9 AD A5 A1 B1 B5 BD B9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
LDX Load index X with memory M → X (1)	IMM ABS Z Z,Y ABS,Y	A2 AE A6 B6 BE	2 4 3 4 4	2 3 2 2 3	N - - - - - Z -
LDY Load index Y with memory M → Y (1)	IMM ABS Z Z,X ABS,X	A0 AC A4 B4 BC	2 4 3 4 4	2 3 2 2 3	N - - - - - Z -
LSR Shift right one bit (memory or accumulator) 0 → [7 0] → C	ABS Z A Z,X ABS,X	4E 46 4A 56 5E	6 5 2 6 7	3 2 1 2 3	0 - - - - - ZC N
NOP No operation	IMP	EA	2	1	- - - - - - - -
ORA "OR" memory with accumulator AVM → A	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	09 0D 05 01 11 15 1D 19	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
PHA Push accumulator on stack A↓	IMP	48	3	1	- - - - - - - -
PHP Push processor status on stack; P↓	IMP	08	3	1	- - - - - - - -
PLA Pull accumulator from stack A↑	IMP	68	4	1	N - - - - - Z -

Mnemonics + Beschreibung	Adressierungs-methode(n)	OP-Code (hex)	Anzahl der Takt-impulse (N)	Anzahl der Bytes	von der Instruktion beeinflußte Flags
PLP Pull processor status from stack; P↑	IMP	28	4	1	(ursprünglicher Status-Register-Zustand)
ROL Rotate one bit left (memory or accumulator) 	ABS Z Z,X ABS,X A	2E 26 36 3E 2A	6 5 6 7 2	3 2 2 3 1	N - - - - - ZC
ROR Rotate one bit right (memory or accumulator) 	ABS Z A Z,X ABS,X	6E 66 6A 76 7E	6 5 2 6 7	3 2 1 2 3	N - - - - - ZC
RTI Return from interrupt PC↑ ; P↑	IMP	40	6	1	(ursprünglicher Status-Register-Zustand)
RTS Return from subroutine PC↑; PC + 1 → PC	IMP	60	6	1	- - - - - - - -
SBC Subtract memory from accumulator with borrow (3) A-M-$\overline{C}$ → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	E9 ED E5 E1 F1 F5 FD F9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - ZC
SEC Set carry flag 1 → C	IMP	38	2	1	- - - - - - - 1
SED Set decimal mode	IMP	F8	2	1	- - - - 1 - - - D
SEI Set interrupt 1 → I, Disable IRQ	IMP	78	2	1	- - - - - 1 - - I
STA Store accumulator in memory A → M	ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	8D 85 81 91 95 9D 99	4 3 6 6 4 5 5	3 2 2 2 2 3 3	- - - - - - - -

Mnemonics + Beschreibung	Adressierungs-methode(n)	OP-Code (hex)	Anzahl der Takt-impulse (N)	Anzahl der Bytes	von der Instruktion beeinflußte Flags
STX Store index X in memory; X → M	ABS Z Z,Y	8E 86 96	4 3 4	3 2 2	- - - - - - - -
STY Store index Y in memory; Y → M	ABS Z Z,X	8C 84 94	4 3 4	3 2 2	- - - - - - - -
TAX Transfer accumulator to index X A → X	IMP	AA	2	1	N - - - - - Z -
TAY Transfer accumulator to index Y A → Y	IMP	A8	2	1	N - - - - - Z -
TSX Transfer stack pointer to index X S → X	IMP	BA	2	1	N - - - - - Z -
TXA Transfer index X to accumulator X → A	IMP	8A	2	1	N - - - - - Z -
TXS Transfer index X to stack pointer X → S	IMP	9A	2	1	N - - - - - Z -
TYA Transfer index Y to accumulator Y → A	IMP	98	2	1	N - - - - - Z -

Bemerkungen:

(1) Zu N 1 addieren, wenn die Seitengrenze überschritten wird.
(2) Zu N 1 addieren, wenn der Sprung in derselben Seite stattfindet.
(3) Borrow = non-carry ($\overline{C}$).
(4) Zu N2 addieren, wenn der Sprung in eine andere Seite führt.

3 Hex dump vom Monitorprogramm

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
1C00: 85 F3 68 85 F1 68 85 EF 85 FA 68 85 F0 85 FB 84
1C10: F4 86 F5 BA 86 F2 A2 01 86 FF 4C 33 1C A9 1E 8D
1C20: 83 1A A9 04 85 F1 A9 03 85 FF 85 F6 A2 FF 9A 86
1C30: F2 D8 78 20 88 1D D0 FB 20 88 1D F0 FB 20 88 1D
1C40: F0 F6 20 F9 1D C9 13 D0 13 A6 F2 9A A5 FB 48 A5
1C50: FA 48 A5 F1 48 A6 F5 A4 F4 A5 F3 40 C9 10 D0 06
1C60: A9 03 85 FF D0 14 C9 11 D0 06 A9 00 85 FF F0 0A
1C70: C9 12 D0 09 E6 FA D0 02 E6 FB 4C 33 1C C9 14 D0
1C80: 0B A5 EF 85 FA A5 F0 85 FB 4C 7A 1C C9 15 10 EA
1C90: 85 E1 A4 FF D0 0D B1 FA 0A 0A 0A 0A 05 E1 91 FA
1CA0: 4C 7A 1C A2 04 06 FA 26 FB CA D0 F9 A5 FA 05 E1
1CB0: 85 FA 4C 7A 1C 20 D3 1E A4 E3 A6 E2 E8 D0 01 C8
1CC0: 86 E8 84 E9 A9 77 A0 00 91 E6 20 4D 1D C9 14 D0
1CD0: 2A 20 6F 1D 10 F7 85 FB 20 6F 1D 10 F0 85 FA 20
1CE0: D3 1E A0 00 B1 E6 C5 FB D0 07 C8 B1 E6 C5 FA F0
1CF0: D9 20 5C 1E 20 F8 1E 30 E9 10 3E C9 10 D0 0A 20
1D00: 20 1E 10 C9 20 47 1E F0 C1 C9 13 D0 14 20 20 1E
1D10: 10 BB 20 5C 1E 20 F8 1E A5 FD 85 F6 20 47 1E F0
1D20: A9 C9 12 D0 07 20 F8 1E 30 A0 10 0D C9 11 D0 09
1D30: 20 83 1E 20 EA 1E 4C CA 1C A9 EE 85 FB 85 FA 85
1D40: F9 A9 03 85 F6 20 8E 1D D0 FB 4C CA 1C A2 02 A0
1D50: 00 B1 E6 95 F9 C8 CA 10 F8 20 5C 1E 20 8E 1D D0
1D60: FB 20 8E 1D F0 FB 20 8E 1D F0 F6 20 F9 1D 60 20
1D70: 5C 1D C9 10 10 11 0A 0A 0A 0A 85 FE 20 5C 1D C9
1D80: 10 10 04 05 FE A2 FF 60 A0 00 B1 FA 85 F9 A9 7F
1D90: 8D 81 1A A2 08 A4 F6 A5 FB 20 CC 1D 88 F0 0D A5
1DA0: FA 20 CC 1D 88 F0 05 A5 F9 20 CC 1D A9 00 8D 81
1DB0: 1A A0 03 A2 00 A9 FF 8E 82 1A E8 E8 2D 80 1A 88
1DC0: D0 F5 A0 06 8C 82 1A 09 80 49 FF 60 48 84 FC 4A
1DD0: 4A 4A 4A 20 DF 1D 68 29 0F 20 DF 1D A4 FC 60 A8
1DE0: B9 0F 1F 8D 80 1A 8E 82 1A A0 7F 88 10 FD 8C 80
1DF0: 1A A0 06 8C 82 1A E8 E8 60 A2 21 A0 01 20 B5 1D
1E00: D0 07 E0 27 D0 F5 A9 15 60 A0 FF 0A B0 03 C8 10
1E10: FA 8A 29 0F 4A AA 98 10 03 18 69 07 CA D0 FA 60
1E20: 20 6F 1D 10 21 85 FB 20 60 1E 84 F7 84 FD C6 F7
1E30: F0 12 20 6F 1D 10 0F 85 FA C6 F7 F0 07 20 6F 1D
1E40: 10 04 85 F9 A2 FF 60 20 A6 1E 20 DC 1E A2 02 A0
1E50: 00 B5 F9 91 E6 CA C8 C4 F6 D0 F6 60 A0 00 B1 E6
```

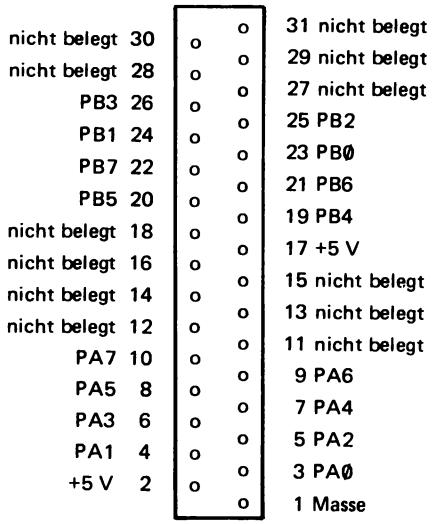
```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
1E60: A0 01 C9 00 F0 1A C9 40 F0 16 C9 60 F0 12 A0 03
1E70: C9 20 F0 0C 29 1F C9 19 F0 06 29 0F AA BC 1F 1F
1E80: 84 F6 60 A5 E6 85 EA A5 E7 85 EB A4 F6 B1 EA A0
1E90: 00 91 EA E6 EA D0 02 E6 EB A5 EA C5 E8 D0 EC A5
1EA0: EB C5 E9 D0 E6 60 A5 E8 85 EA A5 E9 85 EB A0 00
1EB0: B1 EA A4 F6 91 EA A5 EA C5 E6 D0 06 A5 EB C5 E7
1EC0: F0 10 38 A5 EA E9 01 85 EA A5 EB E9 00 85 EB 4C
1ED0: AE 1E 60 A5 E2 85 E6 A5 E3 85 E7 60 18 A5 E8 65
1EE0: F6 85 E8 A5 E9 69 00 85 E9 60 38 A5 E8 E5 F6 85
1EF0: E8 A5 E9 E9 00 85 E9 60 18 A5 E6 65 F6 85 E6 A5
1F00: E7 69 00 85 E7 38 A5 E6 E5 E8 A5 E7 E5 E9 60 40
1F10: 79 24 30 19 12 02 78 00 10 08 03 46 21 06 0E 02
1F20: 02 02 01 02 02 02 01 01 02 01 01 03 03 03 03 6C
1F30: 7A 1A 6C 7E 1A B1 E6 A0 FF C4 EE F0 0D D1 EC D0
1F40: 0A 88 B1 EC AA 88 B1 EC A0 01 60 88 88 88 D0 E9
1F50: 60 38 A5 E4 E9 FF 85 EC A5 E5 E9 00 85 ED A9 FF
1F60: 85 EE 20 D3 1E 20 5C 1E A0 00 B1 E6 C9 FF D0 1D
1F70: C8 B1 E6 A4 EE 91 EC 88 A5 E7 91 EC 88 A5 E6 91
1F80: EC 88 84 EE 20 83 1E 20 EA 1E 4C 65 1F 20 F8 1E
1F90: 30 D3 20 D3 1E 20 5C 1E A0 00 B1 E6 C9 4C F0 16
1FA0: C9 20 F0 12 29 1F C9 10 F0 1A 20 F8 1E 30 E6 A9
1FB0: 03 85 F6 4C 33 1C C8 20 35 1F F0 EE 91 E6 8A C8
1FC0: 91 E6 D0 E6 C8 20 35 1F F0 E0 38 E5 E6 38 E9 02
1FD0: 91 E6 4C AA 1F D8 A9 00 85 FB 85 FA 85 F9 20 6F
1FE0: 1D 10 F2 85 FB 20 6F 1D 10 EB 85 FA 18 A5 FA E5
1FF0: FB 85 F9 C6 F9 4C DE 1F FF FF 2F 1F 1D 1C 32 1F
```

4 Die Anschlüsse der Konnektors

Expansion-Konnektor

32a	Masse	32a	o	o	32c	32c	Masse
31a	RAM-R/W	31a	o	o	31c	31c	nicht belegt
30a	Φ1	30a	o	o	30c	30c	EX_
29a	K1	29a	o	o	29c	29c	R/$\overline{W}$
28a	nicht belegt	28a	o	o	28c	28c	K2
27a	Φ2	27a	o	o	27c	27c	nicht belegt
26a	A1	26a	o	o	26c	26c	AØ
25a	A3	25a	o	o	25c	25c	A2
24a	A5	24a	o	o	24c	24c	A4
23a	A7	23a	o	o	23c	23c	A6
22a	A9	22a	o	o	22c	22c	A8
21a	A11	21a	o	o	21c	21c	AIØ
20a	A13	20a	o	o	20c	20c	A12
19a	A15	19a	o	o	19c	19c	A14
18a	−5 V	18a	o	o	18c	18c	K3
17a	K4	17a	o	o	17c	17c	+12 V
16a	nach 16c	16a	o	o	16c	16c	
15a	K5	15a	o	o	15c	15c	K6
14a	K7	14a	o	o	14c	14c	SO
13a	nicht belegt	13a	o	o	13c	13c	nicht belegt
12a	IRQ	12a	o	o	12c	12c	NMI
11a	nicht belegt	11a	o	o	11c	11c	nicht belegt
10a	D7	10a	o	o	10c	10c	D6
9a	D5	9a	o	o	9c	9c	D4
8a	D3	8a	o	o	8c	8c	D2
7a	D1	7a	o	o	7c	7c	DØ
6a	nicht belegt	6a	o	o	6c	6c	nicht belegt
5a	RES	5a	o	o	5c	5c	RDY
4a	Masse	4a	o	o	4c	4c	Masse
3a	nicht belegt	3a	o	o	3c	3c	nicht belegt
2a	nicht belegt	2a	o	o	2c	2c	nicht belegt
1a	+5 V	1a	o	o	1c	1c	+5 V

Port-Konnektor



Sachwortverzeichnis

Viele verbinden mit dem Stichwort "Computer" die Vorstellung: weniger Arbeit – mehr Freizeit. Aktueller und realistischer ist da schon die Formel: mehr Arbeit in gleicher Zeit. Die steigende Zahl der µComputer-Fans läßt noch einen anderen Schluß zu: mehr Arbeit in der Frei-Zeit!
Vor dem ersten Tastendruck steht allerdings die Qual der Wahl, denn das Angebot an Fix-und-fertig- oder Selbstbau-Cpmputern, an Büchern und Zeitschriften ist sehr groß. Es soll schon Leute gegeben haben, die den Wald vor lauter Bäumen nicht sahen; dann entweder gar nicht anfingen oder zu Hause feststellten, daß der neue Home-Computer wohl doch ein Mißgriff war.

Elektor möchte deshalb allen Newcomern auf diesem Gebiet unter die Arme greifen:
- Mit dem Junior-Computer wird ein preisgünstiger Ein-Platinen-Selbstbau-Mikrocomputer mit einem vierteiligen Lehrbuch als Lernsystem angeboten.
- Das Minimal-System "Junior-Computer" ist darüber hinaus soweit ausbaufähig, wie es die spezielle oder allgemeine Anwendung erfordert.

Also dann: Frisch an's Werk!

Elektor Verlag GmbH, D-5133 Gangelt 1

ISBN 3-921608-18-X