Der einfache Einstieg in die faszinierende Computertechnik

JUNIOR COMPUTER

Buch 2

Elektor Verlag

Junior-Computer 2

Der einfache Einstieg
in die
faszinierende Computertechnik

A. Nachtmann
G.H. Nachbar

ISBN 3-921608-23-6

Elektor Verlag GmbH, 5133 Gangelt 1

© 1980 Elektor Verlag GmbH, 5133 Gangelt 1

Die in diesem Buch veröffentlichten Beiträge, insbesondere alle Aufsätze und Artikel sowie alle Entwürfe, Pläne, Zeichnungen und Illustrationen sind urheberrechtlich geschützt. Ihre auch teilweise Vervielfältigung und Verbreitung ist grundsätzlich nur mit vorheriger schriftlicher Zustimmung des Herausgebers gestattet. Alleiniges Nachdruckrecht für das holländische Sprachgebiet: Elektuur B.V., Beek (L), Holland; für Publikationen in englischer Sprache: Elektor Publishers Ltd., Canterbury, England; für das französische Sprachgebiet: Elektor sarl, Estaires, Frankreich. Der Nachbau der veröffentlichten Schaltungen geschieht außerhalb der Verantwortlichkeit des Herausgebers.

Zwischenmahlzeit!

Wie sind Ihnen die ersten Schritte in die Welt der Computer mit dem Junior-Computer bekommen? Sicherlich gut?! Prima, dann sind Sie ja für weitere Taten bereit. Wie Sie Ihren Bildungshunger stillen können, wollen wir hier kurz aufzählen.
Also: Mahlzeit!
Nach dem Aufbau des Junior-Computers und den ersten Gehversuchen in Kapitel 1 . . . 4 (Buch 1), geht es in diesem Buch ans Eingemachte. Das heißt: Sie erwerben das richtige Handwerkszeug für den Umgang mit dem Junior-Computer. Und das Handwerkszeug bei einem Computer ist nun mal die Programmierung für das entsprechende Programm.
Genau da steigt Kapitel 5 ein. "Editor" und "Assembler" sind wichtige Hilfsmittel für den Programmierer. Eingabefehler lassen sich leicht korrigieren. Sogar fehlende Instruktionen können Sie nachträglich an jede beliebige Stelle des Arbeitsspeichers einfügen, ohne das gesamte Programm neu einzutippen. Schließlich brauchen Sie nicht mehr die Offsets bei bedingten Programmverzweigungen und die absoluten Adressen der Sprungbefehle selbst zu berechnen. Diese stupide Arbeit übernimmt der Assembler im EPROM des Standart-Junior-Computers.
In Kapitel 6 geht's tönend zur Sache. Tönend deshalb, weil einige Zusatzschaltungen zusammen mit dem peripheren Interface Adapter – kurz PIA – den Junior-Computer zur Musikbox machen. Dem Computer ist es schließlich egal, ob er über den PIA einen Lautsprecher oder einen Drucker steuert. Und die Drucker-Schnittstelle ist erst im dritten Buch vorgesehen.
Die EPROM-Intelligenz des Junior-Computers beschreiben die Kapitel 7 (Monitor), Kapitel 8 (Editor) und Kapitel 9 (Assembler). Wenn Sie die Denkweise des Juniors kennen, können sie bei eigenen Programmen darauf zurückgreifen. Die bekannten Subroutinen stehen Ihnen dann bei der "Do-it-yourself-Software" hilfreich zur Seite.
Im Anhang schließlich: Sämtliche Subroutinen und Listings in zusammenhängender Form.
Was nun? Die Programmierung ist soweit klar, wo bleibt der praktische Nutzen? Zwei berechtigte Fragen, auf die es eine Antwort gibt: Machen

Sie mit dem dritten Buch aus Ihrem Junior-Computer einen vollwertigen Personal-Computer, der dann zwar noch Junior heißt, den Kinderschuhen aber längst entwachsen ist.

Die Autoren

P.S. Nichts auf der Welt ist vollkommen. Deshalb sind wir für jede schriftliche Anregung oder Änderung zum Junior mit seinen Büchern recht dankbar.

Hans-Heinrich Behrmann
Aachener Straße 43
4300 Essen 1
Tel. 02 01 / 73 27 77

Inhalt

Kapitel 5 7
Arbeiten mit dem Editor und Assembler — *Editen und assemblieren erleichtern die fehlerfreie Programmeingabe.*

Kapitel 6 33
PIA: Der periphere Interface Adapter — *Bindeglied zwischen Computer und Außenwelt.*

Kapitel 7 99
Das Monitorprogramm — *Unverzichtbare Software.*

Kapitel 8 145
Das Editor-Programm — *Es ist der "Radiergummi" bei fehlerhaft eingegebenen Programmen.*

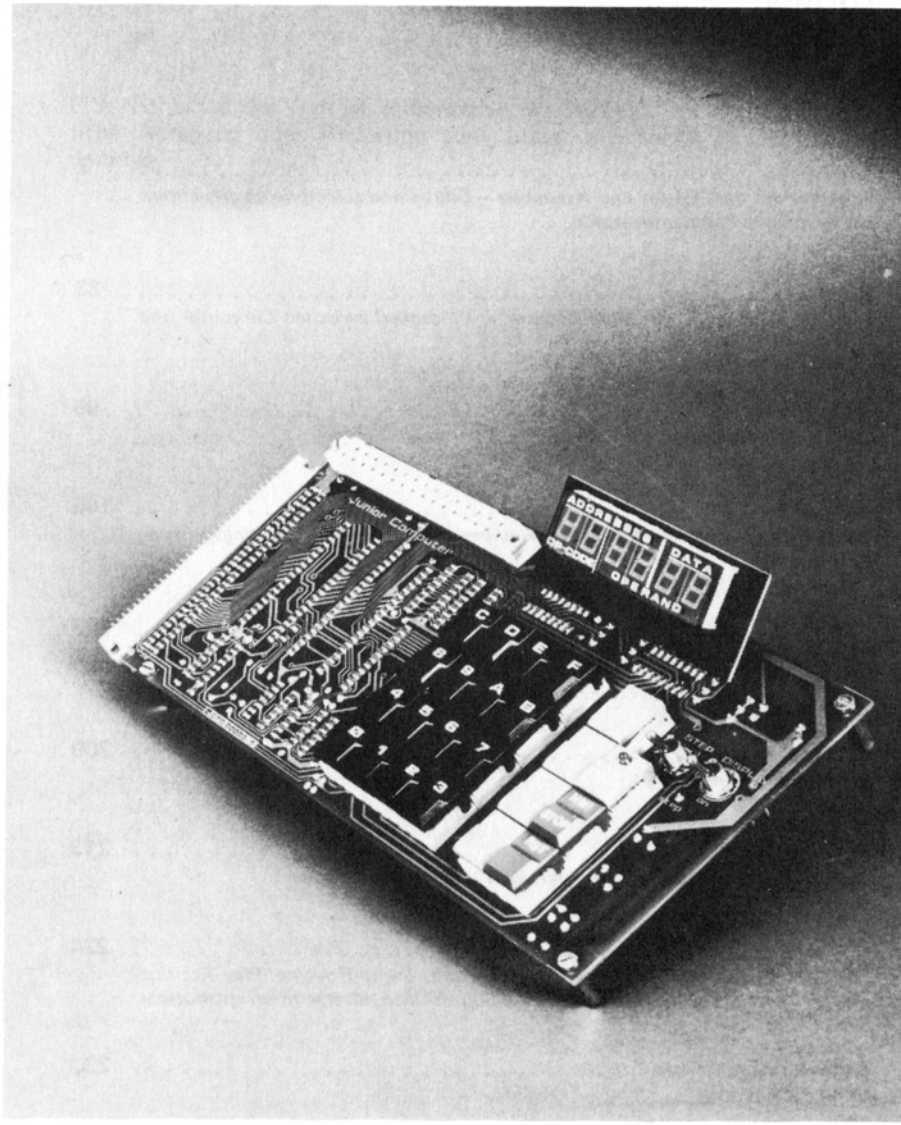
Kapitel 9 183
Das Assembler-Programm — *Paßt das mit dem Editor eingegebene Programm an die CPU an.*

Anhang 1 209
Subroutinen-Summary

Anhang 2 213
Source-Listing — *Editor, Assembler, Monitor, Branch-Routine*

Anhang 3 224
Source-Listing — *Binary Decimal Conversion, Demo-Routine, Play-Routine, Input-Routine, Repeat-Routine, diverse Subroutinen, diverse Interruptroutinen*

Sachwortverzeichnis 233


Junior Computer
ADDRESSES
DATA
OPERAND
STEP
DISPL

5

Arbeiten mit Editor und Assembler

Das Eintasten eines Programms ist eine zeitraubende und langeweilige Angelegenheit. Wie in Buch 1 gezeigt, müssen die einzelnen Instruktionen Byte für Byte in den Computer eingegeben werden. Vor dem Eintippen eines Programms sind auf Papier viele Vorbereitungen vonnöten:

- Berechnung der Startadressen von Subroutinen;
- Berechnung der Offsets bei bedingten Programmverzweigungen;
- Berechnung der absoluten Adressen bei nichtbedingten Programmverzweigungen (Sprung-Instruktionen).

Ist schließlich das ganze Programm in den Computer eingetippt, muß man es auf eventuelle Tippfehler überprüfen. Was tun, wenn bereits zu Programmbeginn ein Byte nicht eingetastet wurde. Das gesamte Programm nochmals neu eintippen? Nein, das ist nicht nötig!

Im EPROM des Junior-Computers befinden sich ein Editor und ein Assembler. Mit dem Editor lassen sich nachträglich Instruktionen in ein Programm einfügen, Instruktionen in einem Programm aufsuchen und Instruktionen aus einem Programm wieder entfernen. Das Tastenfeld und der Editor sind somit "Bleistift und Radiergummi" bei der Eingabe eines Programms in den Junior-Computer.

Mit dem Editor lassen sich auch symbolische Adressen, sog. Labels, in den Computer eingeben. Somit kann der Computer mit dem Assembler die Startadressen von Subroutinen, die Offsets bei bedingten Programm-Verzweigungen und die absolu-

ten Adressen bei Sprung-Instruktionen selbständig berechnen. Programmierfehler werden somit auf ein Minimum reduziert, da der Junior-Computer dem Programmierer bei der Erstellung eines Programms behilflich ist. Wie man mit dem Editor und Assembler arbeitet, besprechen wir in diesem Kapitel.

EDITOR

Aus dem Buch Junior-Computer 1 sind die meisten Befehle und Adressierungsarten der 6502-CPU bekannt. Zahlreiche Übungsprogramme haben gezeigt, wie einfach es ist, den Junior-Computer zu programmieren. Bisher werden die Daten nur in Form von hexadezimalen Zahlen oder Bytes eingegeben. Dabei interpretiert der Computer die mit der AD-Taste eingegebenen Daten als Adresse. Das Drücken der DA-Taste hat zur Folge, daß der Computer die eingegebenen Daten an der angezeigten Adresse ablegt.
Bei kleinen Programmen, wie beispielsweise die Übungsprogramme in Buch 1, ist diese Art der Dateneingabe völlig ausreichend. Bei großen Programmen, mit einer Länge von mehreren hundert Bytes, unterlaufen bei der Eingabe von Daten sehr oft Tippfehler. Was tun, wenn man z.B. in der Mitte des Programms ein paar Bytes irrtümlicherweise vergessen hat? Es ist bisher nur eine Möglichkeit bekannt, nämlich das Programm ab der fehlerhaften Stelle nochmals neu einzugeben. Eine mühselige und zeitraubende Aufgabe! Dabei lassen sich unnütze Instruktionen durch Überschreiben von NOP-Befehlen (Hexcode EA) beseitigen. Das ist aber eine schlechte Ausnutzung des Speichers unseres Junior-Computers. Besonders ärgerlich ist, wenn man ein langes Programm eingetippt hat und am Ende einige Bytes Speicherraum fehlen, weil eben der Speicher nicht optimal genutzt ist.
In Zukunft kann uns das jedoch nicht mehr schrecken, da das Monitorprogramm des Junior-Computers einen EDITOR enthält. Mit Hilfe dieses Editors lassen sich im Speicher Bytes an beliebiger Stelle einfügen oder wieder herausholen. Beim Einfügen von Bytes schiebt der Computer einen Datenblock ab der Stelle nach unten, an der man eine bestimmte Anzahl von Bytes zusätzlich ablegen will. Programme, die einen Blocktransfer durchführen, sind bereits aus Buch 1 – indizierte und indirekte Adressierung – bekannt.
Der Editor nimmt uns also eine Menge Arbeit ab, wenn man Programme nach dem Eintasten korrigieren muß. Wie einfach es ist, mit diesem Editor Programme in den Junior-Computer einzugeben und zu korrigieren, zeigt dieses Kapitels ganz genau.

ASSEMBLER

Ein unentbehrliches Hilfsmittel, Programme einfach, schnell und vor allem fehlerfrei in den Computer zu bekommen, ist der Assembler. Rufen wir uns Buch 1 nochmals ins Gedächtnis! Es sind dort häufig Verzweigungsbefehle (Branch-Instruktionen) verwendet. Um die Offsets dieser Verzweigungen zu berechnen, müssen die Start- und Zieladresse der Verzweigung bekannt sein. Nur dann kann man mit der Monitorroutine BRANCH

(Startadresse 1FD5) das Offset berechnen. Genauso ist es bei den Befehlen JSR und JMP. Auch hier mußte die effektive Sprungadresse bekannt sein. Vor der Programmeingabe müssen also alle absoluten Adressen von Verzweigungs- und Sprungbefehlen bekannt sein. Viel Schreibarbeit ist nötig, um durch Zählen die richtigen Adressen zu erhalten.
Ganz einfach und vor allem fehlerfrei läßt sich diese Aufgabe mit dem Assembler im Monitorprogramm des Junior-Computers lösen. Ein Druck auf die GO-Taste genügt, und der Computer berechnet selbstständig alle Offsets, die Absolutadressen von JMP-Befehlen und die Startadressen von Subroutinen. So unglaublich das klingt, der Junior-Computer nimmt dem Programmierer diese kniffelige Aufgabe in Sekundenbruchteilen ab. Dabei soll nicht unerwähnt bleiben, daß Assembler und Editor neben anderen Programmen in nur 1 KByte EPROM untergebracht sind! Das ist möglich, da die 6502-CPU sehr kräftige Instruktionen und die meisten Adressierungsarten aller auf dem Markt befindlichen Mikroprozessoren hat.
Zusammenfassend kann man also sagen, mit dem Editor und Assembler lassen sich Programme schnell und fehlerfrei erstellen. Dabei übernimmt der Junior-Computer viel Rechenarbeit, um so den Programmierer zu entlasten.

Editen und assemblieren von A bis Z

Bevor es über den Umgang mit Editor und Assembler ins Detail geht, entwickeln wir zuerst ein Programm, das anschließend in den Junior-Computer einzugeben ist. Das Programm soll folgendes bewirken:

- Eine 8 Bit Binärzahl soll in eine Dezimalzahl umgewandelt werden.
- Der Junior-Computer soll die Binärzahl als Hexzahl neben der Dezimalzahl auf dem Display abbilden.
- Die Eingabe der Hexzahl soll über das Keyboard des Junior-Computers erfolgen.
- Für die Scannung des sechsstelligen Displays sind Monitorroutinen zu verwenden.

Das Problem ist also, ein Programm schnell und fehlerfrei zu entwicklen, das diesen Anforderungen genügt. Beginnen wir zuerst, einen Algorithmus zu entwickeln, der eine 8 Bit Binärzahl (ØØ . . . FF) in eine Dezimalzahl ($Ø_{1Ø}$ . . . $255_{1Ø}$ umwandelt. Dazu betrachten wir **Bild 1.** Dort ist der Algorithmus, der diese Umwandlung vollzieht, in Worte gefaßt.
Übrigens ist es bei der Entwicklung eines komplizierten Programms immer gut, das zu lösende Problem zuerst in Worte zu fassen! Anschließend läßt sich leicht eine grobe Flowchart erstellen, die dann zum eigentlichen Programm führt.
Nun zur Programmentwicklung. Bei der Binär-Dezimalumwandlung BINDEC wird zuerst ein Zähler auf Null gesetzt. Anschließend muß man die Hexzahl, die in eine Dezimalzahl umzuwandeln ist, in einem Buffer ablegen. Warum das so ist, macht uns die folgende Aufstellung verständlich:

1) Von der Hexzahl im Buffer wird so oft die Hexzahl ØA = $1Ø_{1Ø}$ abgezogen, bis der Bufferinhalt kleiner als Null ist. Nach jeder Subtraktion inkrementiert der Zähler um eins. Der Inhalt des Zählers zeigt also am Ende an, wieviele Subtraktionen der Computer durchgeführt hat, bis das Ergebnis negativ war.
2) Um eine negative 8 Bit Zahl darstellen zu können, benötigen wir bei

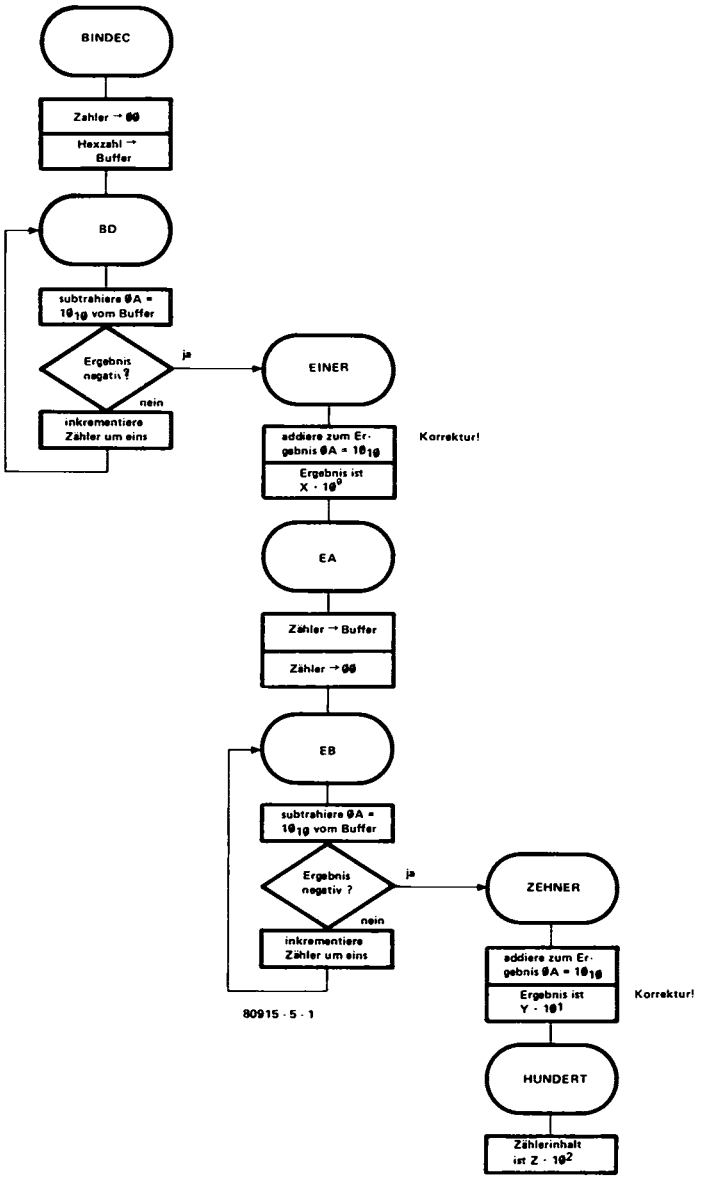


Bild 1. Bevor wir mit dem Editor und Assembler arbeiten, entwickeln wir ein Programm, das eine hexadezimale Zahl in eine Dezimalzahl umwandelt. Somit läßt sich genau zeigen, wie in Zukunft bei der Estellung eines Programms Editor und Assembler als "Programmier-Werkzeug" Verwendung finden. Das Flußdiagramm zeigt den Algorithmus bei der Zahlenumwandlung.

einem 8 Bit Prozessor einen 16 Bit breiten Zahlenbuffer, der auf zwei Speicherzellen verteilt ist.

3) Ist das höchstwertige Bit dieser 16 Bit Zahl log. 1, dann ist das Ergebnis der Subtraktion negativ. Das Programm verzweigt dann zum Label EINER. Dort addiert der Prozessor wieder ØA zum negativen Bufferinhalt. Das Ergebnis ist eine Zahl, die Werte von Ø . . . 9 annehmen kann und die Einer der Dezimalzahl darstellt.
4) Der Zählerinhalt kommt wieder in den 16 Bit breiten Buffer und anschließend wird der Zähler wieder zurückgesetzt. Nun subtrahiert der Prozessor wieder solange ØA vom Bufferinhalt, bis ein negatives Ernis entsteht. Erneut gibt der Zählerinhalt an, um wieviele Subtraktionen es sich dabei handelt.
5) Nach einer abermaligen Addition von ØA zum negativen Bufferinhalt erhalten wir eine Zahl, die Werte von Ø . . . 9 annehmen kann. Es sind die Zehner der Dezimalzahl.
6) Der Zählerinhalt ist jetzt eine Zahl, die Werte von Ø . . . 2 annehmen kann und stellt die Hunderter der Dezimalzahl dar.

Daß dieser Algorithmus tatsächlich funktioniert, zeigt ein praktisches Beispiel. Dazu soll die Hexzahl ØØ91 in eine Dezimalzahl umgewandelt werden und der zuvor entwickelte Programmablauf (Bild 1) Verwendung finden.

Zahlenbuffer ist:	ØØ91	Zähler ist ØØ
	− ØA	Zähler ist Ø1
Zahlenbuffer ist:	ØØ87	
	− ØA	Zähler ist Ø2
Zahlenbuffer ist:	ØØ7D	
	− ØA	Zähler ist Ø3
Zahlenbuffer ist:	ØØ73	
	− ØA	Zähler ist Ø4
Zahlenbuffer ist:	ØØ69	
	− ØA	Zähler ist Ø5
Zahlenbuffer ist:	ØØ5F	
	− ØA	Zähler ist Ø6
Zahlenbuffer ist:	ØØ55	
	− ØA	Zähler ist Ø7
Zahlenbuffer ist:	ØØ4B	
	− ØA	Zähler ist Ø8
Zahlenbuffer ist:	ØØ41	
	− ØA	Zähler ist Ø9
Zahlenbuffer ist:	ØØ37	
	− ØA	Zähler ist ØA
Zahlenbuffer ist:	ØØ2D	
	− ØA	Zähler ist ØB

Zahlenbuffer ist: ØØ23
— ØA Zähler ist ØC

Zahlenbuffer ist: ØØ19
— ØA Zähler ist ØD

Zahlenbuffer ist: ØØØF
— ØA Zähler ist ØE

Zahlenbuffer ist: ØØØ5
— ØA Zähler unverändert

Das Ergebnis ist negativ und wird durch die Addition von ØA wieder positiv, nämlich 5. Berechnet ist also:

$$\$91 = X\ Y\ 5_{10}$$

Der Inhalt des Zählers ist jetzt ØE und wird zum Zahlenbuffer gebracht. Anschließend setzt das Programm den Zähler wieder zurück. Folgender Zustand stellt sich nun ein:

Zahlenbuffer ist: ØØØE Zähler ist ØØ
— ØA Zähler ist Ø1

Zahlenbuffer ist: ØØØ4
— ØA Zähler unverändert

negatives Ergebnis, das durch Addition von ØA wieder 4 ist. Berechnet ist also:

$$\$91 = X\ 4\ 5_{10}$$

Der Inhalt des Zählers ist jetzt 1 und stellt die Hunderter der Dezimalzahl dar. Somit läßt sich die Dezimalzahl wie folgt zusammensetzen:

$$\$91 = 1\ 4\ 5_{10}$$

Wie sich die Zahlensymbole der Dezimalzahl 145 in den Displaybuffer des Junior-Computers packen lassen, ist noch aus Buch 1 bekannt.
Somit ist der Algorithmus für die Binär-Dezimalumwandlung entwickelt und wir können das grobe Flußdiagramm unseres Programms zeichnen, das in **Bild 2** dargestellt ist. Zunächst wird der Displaybuffer mit Nullen gefüllt. Dann fragt der Computer das Keyboard ab und bedient das Display. Die erforderliche Subroutine des Monitorprogramms kennen wir ebenfalls aus Buch 1: GETBYT mit der Startadresse 1D6F. Ist eine Kommandotaste gedrückt, kehrt der Prozessor aus dieser Subroutine mit zurückgesetzter N-Flag ins Hauptprogramm zurück. Somit läßt sich das Display durch Drücken einer beliebigen Kommandotaste immer auf Null setzen.
Nach Drücken von zwei Datentasten kehrt der Mikroprozessor aus der Subroutine GETBYT zurück. Das eingetippte Datenbyte steht im Accu der CPU und wird zum Displaybuffer INH gebracht. Anschließend erfolgt in einer Subroutine die Binär-Dezimalumwandlung gemäß dem eben beschriebenen Algorithmus. Die so gewonnene Dezimalzahl transportiert der Prozessor ebenfalls zum Displaybuffer POINTH und POINTL. Dann kümmert sich der Computer wieder um die Abfrage des Keyboards und zeigt in der Subroutine GETBYT die Hexzahl und die Dezimalzahl an.
Das detaillierte Flußdiagramm des Programms zeigt **Bild 3a**. Der Display-

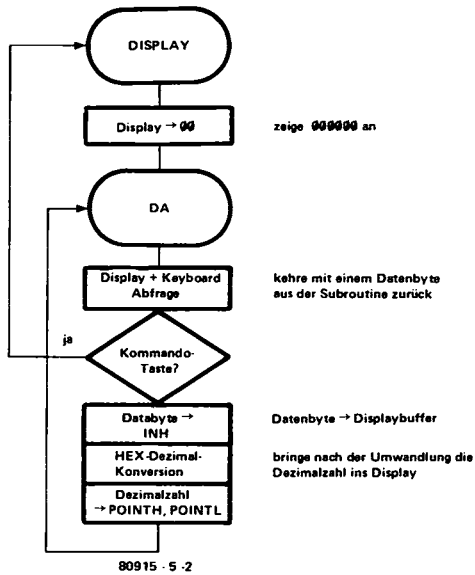


Bild 2. So sieht das globale Flußdiagramm für die Steuerung des Displays und die Abfrage des Tastenfeldes bei der Umwandlung einer hexadezimalen Zahl aus.

buffer hat die Adressen ØØF9 . . . ØØFB. Der Daten oder Zahlenbuffer beansprucht die Adressen ØØD7 und ØØD8. Die für das Programm reservierten Speicherzellen lassen sich wie folgt anschreiben:

```
INH     * $ØØF9  DISPLAY BUFFERS
POINTL  * $ØØFA
POINTH  * $ØØFB
HEXL    * $ØØD7  DATA BUFFERS
HEXH    * $ØØD8
GETBYT  * $1D6F  MONITOR SUBROUTINE
```

Als Zähler für das Subtrahieren verwenden wir keine Speicherzelle, sondern das Y-Register.
Wie aus Bild 3 ersichtlich ist, hat das Hauptprogramm DISPL die beiden Subroutinen GETBYT und HEXDEC. Diese beiden Subroutinen unterscheiden sich dadurch, daß von GETBYT die Startadresse bekannt ist, nicht jedoch von HEXDEC. Das ist von nun an nicht mehr störend, da in Zukunft der Computer selbständig unbekannte Startadressen berechnet. Wir müssen also HEXDEC keine Adresse zuordnen. Außerdem enthält die Hauptroutine noch einen JMP-Befehl und einen Verzweigungsbefehl. Zu welcher Adresse gesprungen oder verzweigt werden soll, berechnet der Computer von sich aus, ohne daß sich der Programmierer darüber den Kopf zerbrechen muß.
Die Subroutine HEXDEC (**Bild 3b**), das ist die praktische Ausführung des zuvor entwickelten Algorithmus, enthält die beiden Subroutinen

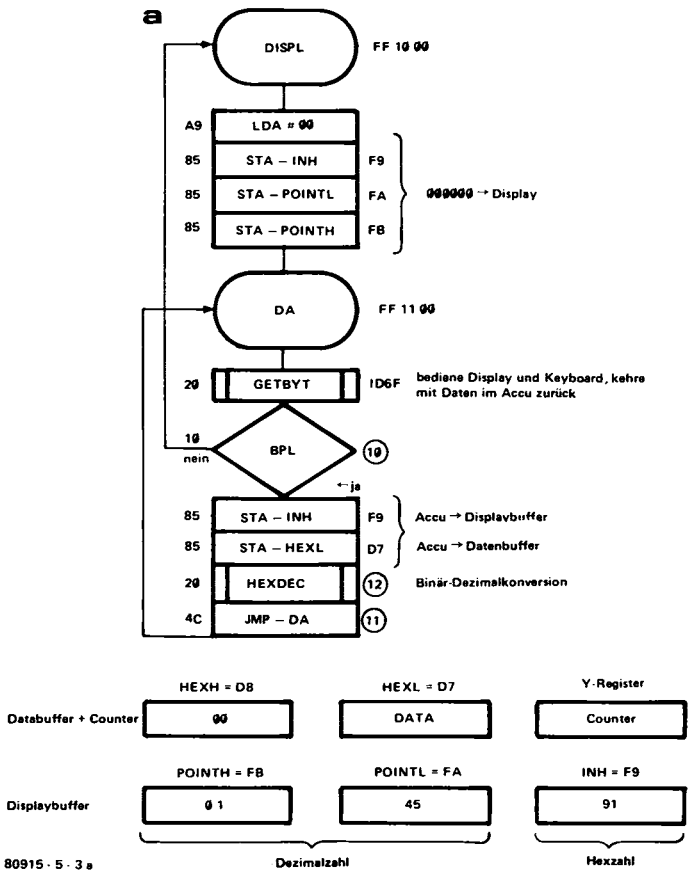


Bild 3a. Das detaillierte Flußdiagramm von Bild 2. Dabei steht die hexadezimale Zahl, die in eine Dezimalzahl umgewandelt werden soll, im Displaybuffer INH. Nach erfolgter Umwandlung befindet sich die Dezimalzahl in den beiden Displaybuffern POINTH und POINTL. Die Speicherzellen HEXH und HEXL sowie das Y-Register finden als temporärer Datenspeicher Verwendung.

COMNUM und SUBTRA **(Bild 3c und Bild 3b)** sowie JMP- und Branchbefehle. Auch hier interessiert uns nicht, welche Startadressen die Subroutinen und welche Offsets die Branchbefehle haben. Diese mühseligen und zeitraubenden Berechnungen überlassen wir dem Computer, der sich nicht irren kann, da sein Monitorprogramm fehlerfrei ist!
Der Teil des Monitorprogramms, der die Offsets von Verzweigungsbefehlen und die Absolutadressen von Sprungbefehlen berechnet, heißt Assembler. Seine Startadresse ist 1F51. Ein Drücken der GO-Taste startet in gewohnter Weise den Assembler.

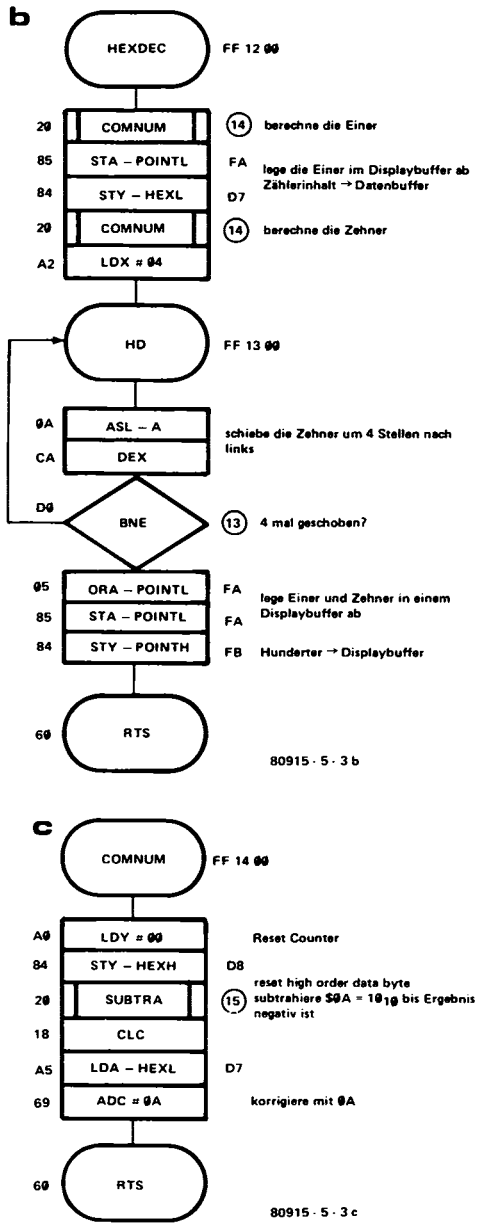


Bild 3b und Bild 3c. In der Subroutine HEXDEC erfolgt die Umwandlung der eingegebenen hexadezimalen Zahl in eine Dezimalzahl. Zuerst werden mit Hilfe der Subroutine COMNUM die Einer, dann die Zehner und Hunderter der Dezimalzahl berechnet.

Damit der Assembler arbeiten kann, benötigt der Computer ein Werkzeug, mit dem sich Programme assemblergerecht eingeben lassen. Dieses Werkzeug ist der Editor, der dem Computer ein eingetipptes Programm "mundgerecht" aufbereitet. Wie zuvor schon erwähnt, soll nun der Computer Adressen und Offsets selbst berechnen. Wie weiß er, wohin er springen oder verzweigen soll, wenn der Programmierer weder Start- noch Zieladresse eingibt? Die Antwort darauf ist: Der Programmierer gibt in den Computer **symbolische Adressen** ein. In Zukunft sagen wir also nicht mehr JSR-Ø23B, sondern JSR-SUBTRA. Beides hat für den Computer dieselbe Bedeutung! Die Aufgabe des Assemblers besteht nun darin, der symbolischen Adresse SUBTRA die absolute Adresse Ø23B zuzuordnen. Das Wort SUBTRA können wir jedoch in den Junior-Computer nicht eingeben, da nur ein hexadezimales Keyboard und kein alphanumerisches zur Verfügung steht. Das heißt, ein Label kann nur hexadezimale Zahlensymbole enthalten. Im Programmspeicher können hexadezimale Zahlen sowohl OP-Codes wie Operanden sein. Der Computer kann also von sich aus nicht unterscheiden, ob es sich um ein Label oder einen OP-Code handelt. Wie läßt sich aber ein Label von einem OP-Code unterscheiden? Sehen wir uns dazu den Anhang von Buch 1, Seite 154 an. Dort sind die OP-Codes der

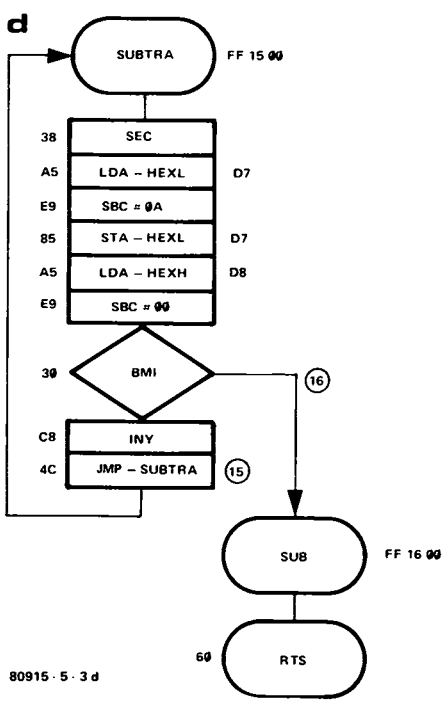


Bild 3d. In der Subroutine SUBTRA führt der Prozessor eine 16-Bit-Subtraktion durch. Dabei wird von der hexadezimalen Zahl sooft ØA subtrahiert, bis ein negatives Ergebnis entsteht. Das Y-Register wird dabei als Subtraktions-Zähler verwendet.

6502-CPU in hexadezimaler Reihenfolge zusammengestellt. Wie aus dieser Tabelle ersichtlich ist, sind nicht allen hexadezimalen Zahlen von ØØ . . . FF OP-Codes zugeordnet. Die Tabelle hat also Lücken. Auch der Hexzahl FF ist kein OP-Code zugeordnet. Deshalb wollen wir dieser Hexzahl einen Pseudo-OP-Code zuordnen, einen OP-Code, den der Mikroprozessor nicht kennt. Somit ist es möglich, die Hexzahl FF als Labelindikator zu verwenden. Wir hätten auch jede andere Zahl aus dieser Tabelle verwenden können, zum Beispiel Ø4, D3, F7 usw. Aber FF läßt sich am einfachsten eintippen und das ist der Grund, warum diese Hexzahl als Labelindikator Verwendung findet.
Verweilen wir noch einen Augenblick beim Label SUBTRA in **Bild 3d.** SUBTRA ist der Startpunkt oder genauer die symbolische Startadresse einer Subroutine. Wollen wir diese Subroutine aufrufen, heißt das: JSR-SUBTRA (sprich: springe zu Subroutine SUBTRA). Da beim Editor der Junior-Computers nur hexadezimale Zahlensymbole verwendet werden, muß der Subroutine SUBTRA eine Nummer zugeordnet werden. Diese Nummer kann eine Zahl von ØØ . . . FF sein. Somit lassen sich 256 unterschiedliche Labels in einem Programm kennzeichnen. Nun dürfte die Idee, die hinter dem Editor und Assembler des Junior-Computers steckt, klar sein. Mit ein paar praktischen Beispielen soll die bisherige Theorie in Fleisch und Blut übergehen, denn in Zukunft verwenden wir bei der Eingabe von Programmen ausschließlich den Editor und Assembler im Monitorprogramm des Junior-Computers.

Der Editor

1) Aufbau eines Labels:

Beispiel:	FF	15	ØØ
allgemein:	FF	XX	ØØ
	Label-Indikator	Label-Nummer	Label-Begrenzer

Ein Label ist also drei Bytes lang. Um ein Label in den Junior-Computer einzugeben, müssen insgesamt sechs Tasten gedrückt werden. Wie aus Bild 3 ersichtlich, ist der Label SUBTRA durch FF 15 ØØ ersetzt worden. Somit können wir nach dem Start des Editors Labels in den Computer eingeben. Eine ASCII-Tastatur (Schreibmaschinentastatur) ist dazu nicht erforderlich.
Weitere **gültige** Labeleingaben sind:

Tastenfeld	Display	
F F 3 7 Ø Ø	FF 37 ØØ	Label-Nummer ist 37
F F F A Ø Ø	FF FA ØØ	Label-Nummer ist FA
F F Ø Ø Ø Ø	FF ØØ ØØ	Label-Nummer ist ØØ

Ungültige Label-Eingaben sind:

Tastenfeld	Display	
F F 2 1	FF 21 XX	Begrenzer fehlt
F F 5 6 F F	FF 56 FF	falscher Begrenzer
F F	FF 41 ØØ	obwohl im Display nach der Eingabe des Label-Indikators der richtige Begrenzer im Display erscheint, muß er durch ØØ überschreiben werden. Erst wenn

die gesamte Label-Eingabe abgeschlossen ist, wird der Displaybuffer in den Speicher kopiert.

2) Aufbau einer JSR-Instruktion:

Beispiel:	20	14	00
allgemein:	20	XX	00
	OP-Code von JSR	Subroutinen-Nummer	Begrenzer

Der Aufruf einer Subroutine ist also, wie gewohnt, drei Bytes lang. Nehmen wir wieder **Bild 3b** zur Hand. In der Subroutine HEXDEC wird zweimal die Subroutine COMNUM aufgerufen. Normalerweise würde man schreiben: JSR-COMNUM. Der Subroutine COMNUM haben wir die Nummer 14 zugeordnet. Wir hätten jedoch auch jede andere Nummer wählen können. 20 14 00 heißt für den Junior-Computer: Springe zur Subroutine mit der Nummer 14. Oder genauer gesagt: Springe in eine Subroutine, die bei der Label-Nummer 14 anfängt. Dabei ist zu bedenken, daß beispielsweise FF 14 00 und COMNUM für das Flußdiagramm ein und dieselbe Bedeutung haben.

Weitere **gültige** Eingaben für eine JSR-Instruktion sind:

Tastenfeld	Display	Bemerkung
2 0 3 9 0 0	20 39 00	springe zur Subroutine Nummer 39
2 0 9 A 0 0	20 9A 00	springe zur Subroutine Nummer 9A
2 0 2 0 0 0	20 20 00	springe zur Subroutine Nummer 20

Ungültige JSR-Instruktionen sind:

Tastenfeld	Display	Bemerkung
2 0 1 1	20 11 XX	Begrenzer fehlt
2 0 7 1 9 F	20 71 9F	falscher Begrenzer
2 0	20 28 00	obwohl im Display nach der Eingabe der JSR-OP-Code **zufällig** die gewünschte Subroutinen-Nummer und der richtige Begrenzer erscheinen, muß die gesamte Instruktion eingegeben werden. Erst wenn die gesamte JSR-Instruktion eingegeben ist,wird der Inhalt des Displaybuffers in den Speicher des Computers kopiert.

3) Aufbau einer JMP-Instruktion:

Beispiel:	4C	11	00
allgemein:	4C	XX	00
	OP-Code von JMP	Label-Nummer	Begrenzer

Auch die JMP-Instruktion ist wie gewohnt drei Bytes lang. In der Hauptroutine DISPL finden wir am Ende eine Sprung-Instruktion: JMP-DA, oder springe zum Label DA. Für den Junior-Computer heißt 4C 11 00: springe zum Label mit der Nummer 11. Auch hier gilt wieder für das Flußdiagramm: Der Label DA und FF 11 00 sind identisch.

Weitere **gültige** Eingaben für eine JMP-Instruktion sind:

Tastenfeld	Display	Bemerkung
4 C 7 7 0 0	4C 77 00	springe zum Label Nummer 77
4 C 2 9 0 0	4C 29 00	springe zum Label Nummer 29
4 C 0 0 0 0	4C 00 00	springe zum Label Nummer 00

Ungültige JMP-Instruktionen sind:

Tastenfeld	Display	Bemerkung
4 C 3 2	4C 32 XX	Begrenzer fehlt
4 C Ø 8 A A	4C Ø8 AA	falscher Begrenzer
4 C	4C 24 ØØ	obwohl im Display nach der Eingabe des JMP-OP-Codes **zufällig** die gewünschte Labelnummer, zu der gesprungen werden soll und der richtige Begrenzer erscheinen, muß die gesamte Instruktion eingegeben werden. Erst wenn die gesamte JMP-Instruktion eingegeben ist, wird der Display-Buffer in den Speicher des Computers kopiert.

Die Hexzahlen hinter JSR- und JMP-Befehlen sind also keine Absolutadressen, sondern Label-Nummern, die der Programmierer frei wählen kann. Die wahren Absolutadressen, zu denen gesprungen werden soll, berechnet der Junior-Computer beim Assemblieren selbständig. JSR- und JMP-Instruktionen sind, wie bereits bekannt, drei Bytes lang **und benötigen das Begrenzerbyte ØØ**. Diese Besonderheit ist bei der Eingabe eines Programms in den Junior-Computer **unbedingt** zu beachten!

4) Der Aufbau einer Branch-Instruktion:

Die 6502-CPU kennt acht Branch-Instruktionen:

BPL, BMI, BEQ, BNE, BCC, BCS, BVC, BVS mit den OP-Codes 1Ø, 3Ø, FØ, DØ, 9Ø, BØ, 50, 70. Für diese Instruktionen berechnet der Assembler im Monitorprogramm des Junior-Computers selbstständig die Offsets. Eine Branch-Instruktion baut sich wie folgt auf:

Beispiel:	3Ø	16
allgemein:	3Ø	XX
	OP-Code von BMI	Label-Nummer

Andere Beispiele von Branch-Instruktionen sind:

1Ø 34	BPL zur Label-Nummer 34
3Ø F6	BMI zur Label-Nummer F6
FØ 19	BEQ zur Label-Nummer 19
DØ 56	BNE zur Label-Nummer 56
9Ø 9D	BCC zur Label-Nummer 9D
BØ 21	BCS zur Label-Nummer 21

usw.

Die Zahlen hinter den Branch-Befehlen sind also keine Offsets, sondern Label-Nummern, die der Programmierer frei wählen kann. Die wahren Offsets berechnet der Junior-Computer beim Assemblieren selbständig. Branch-Instruktionen sind zwei Bytes lang und benötigen **nicht** das Begrenzerbyte ØØ.

5) Weitere Besonderheiten:

Bei der Eingabe eines Programms in Editor-Mode ist darauf zu achten, daß Labels eindeutig definiert sind. Das heißt, ein und dieselbe Label-Nummer darf **nur einmal** einem Label zugeordnet werden. Die Label-Nummern darf man in beliebiger Reihenfolge in den Computer eingeben. Bei der Eingabe von Branch-Instruktionen ist ferner zu beachten, daß die

Verzweigung nicht über den erlaubten Bereich von −128 . . . +127 Schritten hinausführt. Der Assembler erzeugt dann **keine** Error-Meldung, sondern berechnet das **falsche** Offset.

6) JMP, JSR mit definierten Adressen:

Häufig soll eine JMP- oder JSR-Instruktion zu einer bestimmten Adresse führen. Das heißt, der Computer soll den JMP- oder JSR-Befehl nicht assemblieren. In **Bild 3a** springen wir im Hauptprogramm zur Subroutine GETBYT mit der Adresse 1D6F. Diese Adresse ist im Monitorprogramm des Junior-Computers definiert und darf sich beim assemblieren nicht ändern! Im Assembler ist eine wirkungsvolle Sperre eingebaut, um bestimmte JMP- und JSR-Instruktionen nicht zu assemblieren:

- Eine JMP- oder JSR-Instruktion wird vom Assembler nur dann assembliert, wenn ihr eine Labelnummer zugeordnet ist.
- Eine JMP- oder JSR-Instruktion wird **nicht** assembliert, wenn ihr nicht eindeutig eine Label-Nummer zugeordnet ist.

Beispiel: JSR-GETBYT = 2Ø 6F 1D
OP-Code von JSR ADL ADH

Das niederwertige Adreßbyte der JSR-Instruktion ist ADL = 6F. Im Programmbeispiel von **Bild 3** kommt die Label Nummer 6F nicht vor. Deshalb ordnet der Assembler JSR-GETBYT keine Adresse zu, sondern läßt diese Instruktion unberücksichtigt.

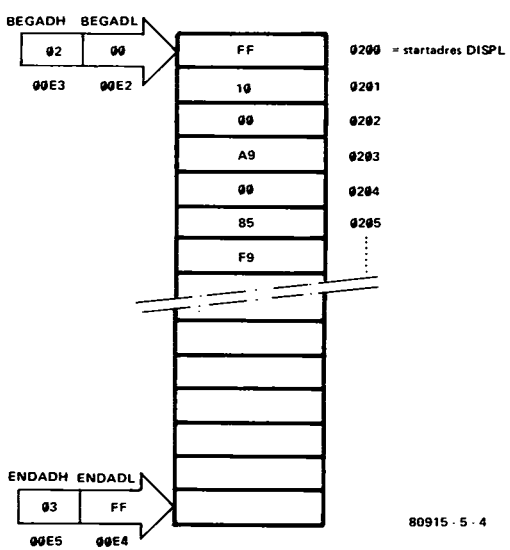


Bild 4. So legt der Editor die ersten Instruktionen des Programms DISPL von Bild 3a im Arbeitsspeicher des Junior-Computers ab. BEGAD und ENDAD sind Pointer, die den Arbeitsspeicher "aufspannen".

7) Die Eingabe der übrigen Instruktionen:

Als Beispiel wählen wir die LDA-Instruktion.

Tastenfeld	Display	Bemerkung	Adressing Mode
A 9 7 F	A9 7F	LDA #7F	Immediate
A D 8 2 1 A	AD 82 1A	LDA-1A82	Absolute
A 5 E 6	A5 E6	LDAZ-E6	Zero Page
A 1 F A	A1 FA	LDA-(FA,X)	Indexed Indirect
B 1 F A	B1 FA	LDA-(FA),Y	Indirect Indexed
B D Ø Ø Ø 2	BD ØØ Ø2	LDA-Ø2ØØ,X	Absolute Indexed,X
B 9 D 1 2 2	B9 D1 22	LDA-22D1,Y	Absolute Indexed,Y
B 5 3 1	B5 31	LDAZ-31,X	Zero Page Indexed,X

Was für die LDA-Instruktion gültig ist, trifft natürlich auch für alle anderen Instruktionen der 6502-CPU zu. Für jede eingegebene Instruktion berechnet der Junior-Computer selbstständig die Länge. Dazu verwendet er die bekannte Subroutine OPLEN im Systemmonitor. Diese Subroutine ist fast identisch mit LENACC aus Buch 1. Somit steuert der Computer das Display in Abhängigkeit der Instruktionslänge an. Instruktionen mit Implied Addressing sind bekanntlich nur 1 Byte lang. Deshalb leuchtet bei der Eingabe einer solchen Instruktion nur das OP-Code-Feld auf dem Display auf. Das Operanden-Feld des Displays ist also ausgeschaltet.
Instruktionen, die wie die LDA-Instruktion zu Punkt (7) gehören, kann **keine symbolische Adresse** wie den JMP-, JSR- und Branch-Instruktionen zugeordnet werden. Das heißt, die Adressen, von denen aus geladen oder in die geschrieben werden soll, muß der Programmierer vor der Eingabe des Programms definieren.

8) Definition des Speicherbereichs:

Bevor man den Editor startet, muß der Speicherbereich klar sein, in dem das Programm abgelegt werden soll. Dazu sind zwei Adreßpointer erforderlich, die in Page Ø des Junior-Computers abgelegt sind. Der Pointer BEGADH, BEGADL (= Begin Address High, Begin Address Low) ist in folgenden Zellen gespeichert:

BEGADL * $ØØE2
BEGADH * $ØØE3

Der Pointer ENDADH, ENDADL (= End Address High, End Address Low) ist in folgenden Speicherzellen abgelegt:

ENDADL * $ØØE4
ENDADH * $ØØE5

BEGAD zeigt also immer auf die Anfangsadresse und ENDAD auf die Endadresse des Speicherbereichs, in dem das Programm abgelegt werden soll. Beim Junior-Computer stehen zwei zusammenhängende Seiten, Page 2 und Page 3, mit den Adressen Ø2ØØ . . . Ø3FF zur Verfügung. Das ist 1/2 KByte-Speicher, der für die Eingabe der meisten Programme mehr als ausreichend ist. Es ist also sinnvoll, diesen großen, zusammenhängenden Speicherraum durch die Adreßpointer zu definieren, bevor man ein Programm eintippt:

BEGADH, BEGADL = Ø2 ØØ und
ENDADH, ENDADL = Ø3 FF

Mit dem Keyboard lassen sich die Pointer folgendermaßen festlegen:

AD	Ø	Ø	E 2
DA	Ø	Ø	ADL des BEGAD-Pointers
+	Ø	2	ADH des BEGAD-Pointers
+	F	F	ADL des ENDAD-Pointers
+	Ø	3	ADH des ENDAD-Pointers

Die Pointer sind in Page Ø des Junior-Computers abgelegt. Der Speicherbereich, in dem das Programm laufen soll, ist somit definiert: Ø2ØØ . . . Ø3FF (Bild 4).
Es lassen sich mit diesen Pointern auch andere Speicherbereiche in der Grundausführung des Junior-Computers festlegen:

in Page Ø	$ØØØØ . . . ØØEØ
in Page 1	$Ø1ØØ . . . Ø1F2
in Page 1A	$1AØØ . . . 1A79.

In diesen Seiten steht nur ein begrenzter Speicherbereich zur Verfügung, da ein Teil der Speicherzellen vom Monitorprogramm und vom Stack beansprucht wird. Überschreitet man den zuvor definierten Speicherbereich, macht der Junior-Computer eine Errormeldung: EEEEEE.

9) Das Starten des Editors:

Nachdem der Speicherbereich definiert ist, läßt sich der Editor starten. Seine Startadresse ist 1CB5. Um den Editor zu starten, sind folgende Tasten zu drücken:

AD 1 C B 5
GO

Auf dem Display erscheint dann im OP-Code-Feld 77. Die Displays für den Operanden sind ausgeschaltet. Somit teilt der Computer mit, daß der Editor arbeitet und bereit ist, eingegebene Instruktionen im zuvor definierten Arbeitsspeicher abzulegen.

10) Die Kommando-Tasten des Editors:

Sobald der Editor gestartet ist, verlieren die Aufschriften AD, DA, +, PC und GO auf den Kommandotasten ihre Gültigkeit. Von nun an gilt die zweite Aufschrift der Kommando-Tasten: SEARCH, INPUT, INSERT, DELETE und SKIP. Mit diesen Kommandos lassen sich Instruktionen in den Junior-Computer eingeben (INSERT, INPUT), wieder herausholen (DELETE) oder auch aufsuchen (SEARCH). Ferner ist es möglich, von Instruktion zu Instruktion zu springen. Das Kommando hierfür ist SKIP. Diese Kommandos wollen wir nun nacheinander besprechen:

INSERT

Mit dem INSERT-Kommando läßt sich eine neue Instruktion vor der auf dem Display angezeigten Instruktion einfügen. Immer, wenn auf dem Display der Pseudo-OP-Code 77 erscheint, **muß unbedingt** die INSERT-Taste gedrückt werden, wenn man anschließend Daten in Form von Instruktionen in den Computer eingeben will. Die INSERT-Taste muß man deshalb immer zuerst drücken, wenn nach dem Start des Editors ein Label oder eine Instruktion in den Computer eingegeben werden soll.

INPUT

Mit dem INPUT-Kommando läßt sich eine neue Instruktion hinter der auf dem Display angezeigten Instruktion einfügen. Zwischen den Kommandos INPUT und INSERT kann der Programmierer ständig wechseln. Es ist immer die zuletzt gedrückte Kommandotaste gültig.
Allgemeines zu INSERT und INPUT:
Bei der Eingabe von Labels oder Instruktionen mit den Kommando-Tasten INSERT und INPUT, werden die gedrückten Daten-Tasten byteweise zum Displaybuffer gebracht. Erst wenn die gesamte Instruktion in den Computer eingetippt ist, gelangt diese vom Displaybuffer in den zuvor definierten Speicherbereich. Somit ist es möglich, falsch eingetastete Instruktionen durch abermaliges Drücken der INSERT- oder INPUT-Taste zu überschreiben. INSERT setzt dabei die berichtigte Instruktion vor die auf dem Display zuletzt angezeigte Instruktion. INPUT setzt dagegen die neu eingetastete Instruktion hinter die zuletzt angezeigte Instruktion. Die Funktionsweise der Kommandotasten INSERT und INPUT bleibt somit voll erhalten. Das ist eine einfache, jedoch sehr wirkungsvolle Methode, Programme zu editen oder in den Computer einzugeben.

DELETE

Mit dem DELETE-Kommando läßt sich die gerade im Display angezeigte Instruktion aus dem Speicher des Junior-Computers entfernen. Die somit im Speicher freigewordenen Zellen füllt der Computer von sich aus auf. Dazu wird der Datenblock, welcher der herausgeholten Instruktion folgt, um die Anzahl der entfernten Bytes nach oben transportiert. Beim Entfernen einer Instruktion aus den Speicher entstehen somit keine "Löcher" im Programm. Nach dem Drücken der DELETE-Taste ist deshalb die Instruktion im Display zu sehen, die der herausgeholten Instruktion folgt. Durch abermaliges Drücken der INSERT- oder INPUT-Taste lassen sich erneut Instruktionen vor der oder hinter der auf dem Display angezeigten Instruktion eingeben.

SEARCH

Mit dem SEARCH-Kommando läßt sich ein beliebiges Zweibyte-Muster im Speicherbereich aufsuchen. Drücken wir beispielsweise die Tasten SEARCH F F 1 1, sucht der Computer die Label-Nummer 11 im Speicher auf und zeigt diese im Display an. Mit dem SEARCH-Kommando läßt sich der Display-Pointer CURAD, der ebenfalls in Page Zero abgelegt ist, manipulieren. Der Display-Pointer CURAD zeigt immer auf den OP-Code, den gerade das Display des Junior-Computers anzeigt. CURADH, CURADL ist wie folgt definiert:

CURADL * $00E6
CURADH * $00E7

Das SEARCH-Kommando führt unter Software-Steuerung den Display-Pointer CURAD zur Anfangsadresse des zuvor definierten Speicherbereichs. Da jede Instruktion eine definierte Länge hat, die der Computer selbständig berechnen kann, wird CURAD von OP-Code zu OP-Code (nicht von Speicherzelle zu Speicherzelle!) bewegt! Durch einfache Vergleichs-Operationen stellt der Computer fest, ob das vom Programmierer

gewünschte 2 Byte-Muster gefunden ist. Somit lassen sich mit dem SEARCH-Kommando nicht nur Labels im Speicherbereich aufsuchen, sondern beliebige Instruktionen:
SEARCH A 9 Ø Ø oder
SEARCH 4 C 1 1 (Bild 3a) oder
SEARCH 6 9 Ø A (Bild 3c)
Bemerkung: Ist beispielsweise mehrmals hintereinander die Instruktion A9 ØØ vorhanden, so läßt sich mit dem SEARCH-Kommando nur die erste A9 ØØ – Instruktion, beginnend am Anfang des Speicherbereichs auffinden. Das heißt, mit dem SEARCH-Kommando lassen sich **nicht** zwei identische, aufeinanderfolgende Instruktionen auffinden!

SKIP

Mit dem SKIP-Kommando läßt sich von Instruktion zu Instruktion springen. Dieses Kommando wird benötigt, um ein eingetipptes Programm schrittweise zu durchlaufen. Ein schnelles Überprüfen auf eventuelle Tippfehler kann man mit dem SKIP-Kommando leicht durchführen.
Bemerkung: Das INPUT-Kommando ist eine Kombination des SKIP- plus INSERT-Kommandos. Dabei bestehen folgende Analogien:

SKIP INSERT XX	= INPUT XX	(1 Byte)
SKIP INSERT XXXX	= INPUT XXXX	(2 Bytes)
SKIP INSERT XXXXXX	= INPUT XXXXXX	(3 Bytes)

11) Cold Start

11) Cold Start Entry/Warm Start Entry:
Der Editor läßt sich von zwei Adressen starten:

Cold Start Entry = 1CB5 und
Warm Start Entry = 1CCA.

Cold Start Entry ist bereits bekannt. Wird der Editor an dieser Adresse gestartet, setzt er einige Pointer in Page Zero in Abhängigkeit des vom Programmierer gewählten Speicherbereichs. Bevor ein Programm überhaupt in den Computer eingegeben werden kann, ist der Editor an der Adresse 1CB5 zu starten.
Warm Start Entry ermöglicht dem Programmierer nach Drücken der RST-Taste wieder in den Editor zu kommen. Alle Pointer in Page Ø werden dann **nicht** neu gesetzt. Um nach dem Drücken der RST-Taste auf diese Weise wieder in den Editor zu kommen, sind folgende Tasten zu drücken:

AD 1 C C A
GO

Auf dem Display erscheint dann die zuletzt angezeigte Instruktion. Die Kommandotasten INSERT, INPUT, DELETE, SEARCH und SKIP behalten auch bei Warm Start Entry ihre normale Wirkungsweise.

12) Eingabe des Programms von Bild 3:

Drücke Taste:		Display:	Bemerkung:
RST		XX XX XX	Initialisierung
AD	Ø Ø E 2	ØØ E2 XX	setzen des BEGAD-Pointers
DA	Ø Ø	ØØ E2 ØØ	
+	Ø 2	ØØ E3 Ø2	
+	F F	ØØ E4 FF	setzen des ENDAD-Pointers
+	Ø 3	ØØ E5 Ø3	
AD	1 C B 5	1C B5 2Ø	
GO		77	starten des Editors (Cold Start Entry)
INSERT	F F 1 Ø Ø Ø	FF 1Ø ØØ	Eingabe der Label-Nummer 1Ø
INPUT	A 9 Ø Ø	A9 ØØ	LDA # ØØ
INPUT	8 5 F 9	85 F9	STAZ-INH
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 5 F B	85 FB	STAZ-POINTH
INPUT	F F 1 1 Ø Ø	FF 11 ØØ	Label-Nummer 11
INPUT	2 Ø 6 F 1 D	2Ø 6F 1D	JSR-GETBYT (6F ist keine Label-Nummer!)
INPUT	1 Ø 1 Ø	1Ø 1Ø	BPL zur Label-Nummer 1Ø
INPUT	8 5 E	85 XX	Tippfehler! Instruktion ist noch nicht im Speicher
INPUT	8 5 F 9	85 F9	STAZ-INH
INPUT	8 5 D 7	85 D7	STAZ-HEXL
INPUT	2 Ø 1 2 Ø Ø	2Ø 12 ØØ	JSR-HEXDEC (HEXDEC hat die Nummer 12)
INPUT	4 C 1 1 Ø Ø	4C 11 ØØ	JMP-DA (DA hat die Nummer 11)
INPUT	F F 1 2 Ø Ø	FF 12 ØØ	HEXDEC hat die Labelnummer 12)
INSERT			Tippfehler! INSERT statt INPUT
INPUT	2 Ø 1 4 Ø Ø	2Ø 14 ØØ	JSR-COMNUM (COMNUM hat die Nummer 14)
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 4 D 7	84 D7	STYZ-HEXL
INPUT	2 Ø 1 4 Ø Ø	2Ø 14 ØØ	JSR-COMNUM (COMNUM hat die Nummer 14)
INPUT	A 2 Ø 4	A2 Ø4	LDX # Ø4
INPUT	F F 1 3 Ø Ø	FF 13 ØØ	HD (HD hat die Label-Nummer 13)
INPUT	Ø A C A	ØA EEEEEE	Kommando-Taste drücken, sonst Error!
INPUT	C A	CA	DEX
INPUT	D Ø 1 3	DØ 13	BNE zur Label-Nummer 13
INPUT	Ø 5 F A	Ø5 FA	ORAZ-POINTL
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 4 F B	84 FB	STYZ-POINTH
INPUT	6 Ø	6Ø	RTS
SEARCH	F F 1 3	FF 13 ØØ	suche Label-Nummer 13
SKIP		ØA	
SKIP		CA	
SKIP		DØ 13	
SKIP		Ø5 FA	
SKIP		85 FA	
SKIP		84 FB	
SKIP		6Ø	
SKIP		77	77 heißt: INSERT-Taste drücken!!!!!!!!!!
INSERT	F F 1 4 Ø Ø	FF 14 ØØ	COMNUM hat die Label-Nummer 14
INPUT	8 4 D 8	84 D8	STYZ-HEXH

Bemerkung: Aus Versehen wurde die Instruktion LDA #ØØ vergessen. Diese Instruktion läßt sich leicht einfügen. Dazu muß nur das INSERT-Kommando verwendet werden. Also:

```
INSERT  A 9 Ø Ø   A9 ØØ  LDA #ØØ
SKIP              84 D8  STYZ-HEXH
```

Die Instruktion LDA #ØØ wurde tatsächlich vor die Instruktion STYZ-HEXH eingefügt. Das war eine Menge Arbeit für den Computer, nicht jedoch für den Programmierer. Nun können wir wieder mit dem INPUT-Kommando weitermachen, denn hinter 84 D8 soll eine neue Instruktion eingefügt werden. Also:

```
INPUT   2 Ø 1 5 Ø Ø   2Ø 15 ØØ   JSR-SUBTRA (SUBTRA hat die Nummer 15)
INPUT   1 8           18         CLC
INPUT   A 5 D 7       A5 D7      LDAZ-HEXL
INPUT   6 9 Ø A       69 ØA      ADC # ØA
INPUT   6 Ø           6Ø         RTS;
INPUT   F F 1 5 Ø Ø   FF 15 ØØ   SUBTRA hat die Label-Nummer 15
INPUT   3 8           38         SEC
INPUT   A 5 D 7       A5 D7      LDAZ-HEXL
INPUT   E 9 Ø A       E9 ØA      SBC # ØA
INPUT   8 5 D 7       85 D7      STAZ-HEXL
INPUT   A 5 D 8       A5 D8      LDAZ-HEXH
INPUT   E 9 Ø Ø       E9 ØØ      SBC # ØØ
INPUT   3 Ø 1 6       3Ø 16      BMI zur Label-Nummer 16
INPUT   C 8           C8         INY
INPUT   4 C 1 5 Ø Ø   4C 15 ØØ   JMP-SUBTRA (SUBTRA hat die Nummer 15)
INPUT   F F 1 6 Ø Ø   FF 16 ØØ   SUB hat die Label-Nummer 16
INPUT   6 Ø           6Ø         RTS
```

Das Programm ist somit in den Junior-Computer eingegeben. Es ist empfehlenswert, das eingetippte Programm auf eventuelle Tippfehler zu überprüfen. Dazu geht man folgendermaßen vor:

```
SEARCH F F 1 Ø
SKIP
SKIP
  .
  .
  .
  .
SKIP
```

Erst wenn das eingetippte Programm mit dem Flußdiagramm übereinstimmt, kann man den Assembler starten.

Der Assembler

Mit dem Editor haben wir nun ein Programm in den Junior-Computer eingegeben, das symbolische Adressen enthält. Diese symbolischen Adressen sind Label-Nummern, Nummern von Subroutinen oder Labels, zu denen gesprungen oder verzweigt werden soll. Ein Programm mit diesen symbolischen Adressen ist für den Junior-Computer nicht geeignet, da der Mikroprozessor damit nichts anfangen kann. Der Assembler muß also das eingegebene Programm so gestalten, daß der 6502-Prozessor damit arbeiten kann. Somit läßt sich die Aufgabe des Assemblers wie folgt defnieren:

- Alle Labels müssen aus dem Programm entfernt werden.
- JSR-Befehlen müssen die tatsächlichen Absolutadressen eindeutig zugeordnet sein.
- Dasselbe gilt für JMP-Befehle.
- Branch-Befehle müssen das tatsächliche Offset erhalten.
- Am Schluß der Assemblierung muß das Programm in einer für die 6502-CPU verständlichen Weise im Programmspeicher des Computers stehen.

Wie steht das Programm vor der Assemblierung im Speicher?

Als Startadresse für unser Programm ist Ø2ØØ festgelegt. Bild 5 zeigt, wie der Editor das Programm im Speicher des Junior-Computers abgelegt hat. Da für jeden Label drei Speicherplätze reserviert sind, ist das eingetippte Programm wesentlich länger als das assemblierte Programm. Nun ist es höchste Zeit, den Assembler zu starten. Seine Startadresse ist 1F51:

RST	Aufruf des Monitor Programms
AD 1 F 5 1	Assembler Startadresse
GO	Starten des Assemblers

Nach dem Niederdrücken der GO-Taste wird das Display für einige Sekundenbruchteile dunkel. Das ist die Zeit, die der Junior-Computer benötigt, um das Programm zu assemblieren. Bei langen Programmen kann das Display für einige Sekunden dunkel sein. Sobald das Programm assembliert ist, meldet sich der Computer über das Monitor Programm zurück. Nun sind wieder die Tasten AD, DA, +, PC und GO gültig.

Wie steht das Programm nach der Assemblierung im Speicher?

Der Junior-Computer hat einen Two Pass Assembler. Was heißt das? Die Assemblierung des Programmes vollzieht der Computer nicht in einem Schritt, sondern er benötigt dazu zwei Schritte oder genauer zwei Durchläufe:

1. Schritt

Der Computer liest das eingetippte Programm **(Bild 5)**. Für ihn sind jetzt **nur** die Labels, die der Programmierer eingetippt hat, interessant. Alle anderen Instruktionen werden ignoriert. Trifft der Computer im Programm auf einen Label (ein Label ist durch FF gekennzeichnet), speichert er die Label-Nummer und die Adresse, an der sich der Label befindet auf einem Symbol-Stack ab. Dieser Symbol-Stack ist nichts anderes als eine Lookup Table, die sich der Computer selbstständig anfertigt. Der Programmierer muß sich darum nicht kümmern.

Nachdem die Label-Nummer und die Absolutadresse, an der sich der Label befindet, auf den Symbol-Stack gerettet sind, entfernt der Computer den Label aus dem Programm. Wie funktioniert das? Ein Label ist bekanntlich

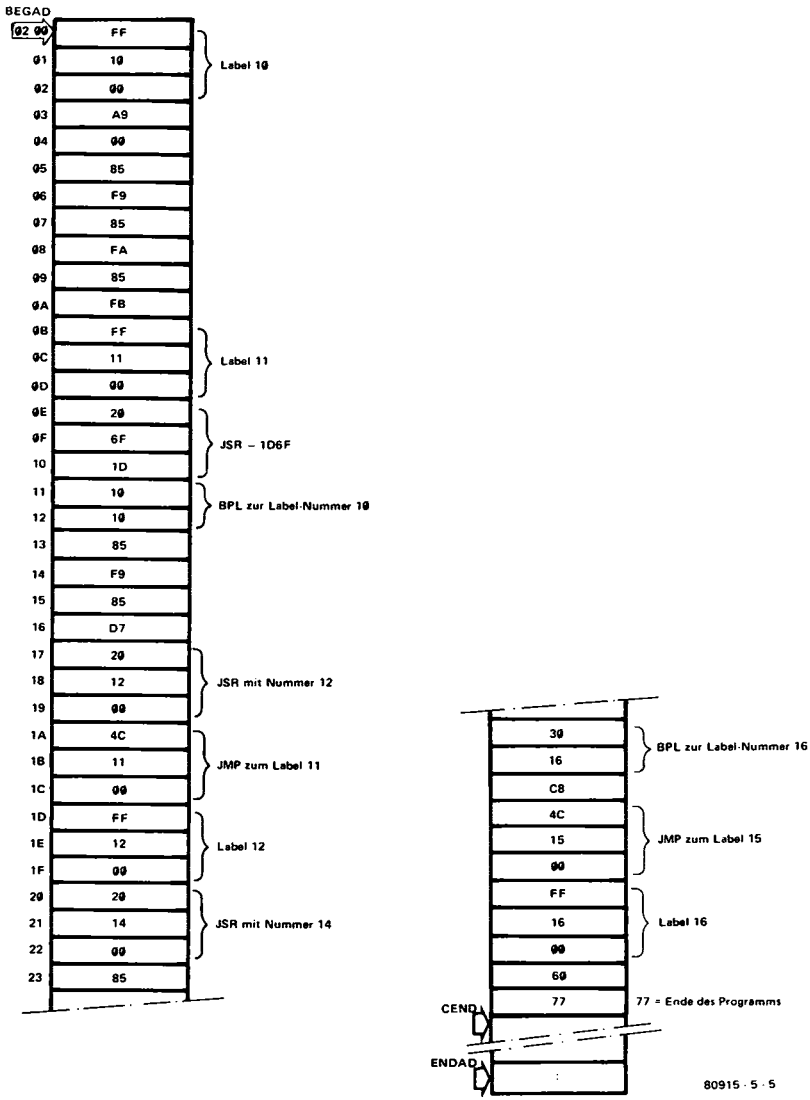


Bild 5. Der Arbeitsspeicher beginnt bei BEGAD und endet bei ENDAD. Der Speicherraum zwischen BEGAD und CEND enthält das eingetippte Programm, das im Beispiel mit dem Label FF 10 00 beginnt und mit dem EOF-Charakter 77 endet. Der Speicherraum zwischen BEGAD und CEND heißt File. Hier enthält die File alle im Programm verwendeten Instruktionen, auch die Adressen, die sog. Labels mit dem Pseudo-OP-Code FF. Ein Programm kann in dieser Form vom Junior-Computer nicht "verdaut" werden, da die Offsets und die absoluten Sprungadressen noch nicht bekannt sind. Erst die anschließende Assemblierung macht das mit dem Editor eingetippte Programm der 6502-CPU "mundgerecht".

drei Bytes lang. Der Computer schiebt deshalb von unten das gesamte Programm um drei Bytes nach oben. Die Instruktionen, die dem Label folgen, vernichten denn diesen durch Überschreiben. Der Label ist somit aus dem Programm verschwunden.
Dann wiederholt sich der gleiche Vorgang. Der Prozessor arbeitet sich wieder von oben nach unten durchs Programm. Dabei startet er in unserem Beispiel immer an der Adresse Ø2ØØ. Er sucht solange weiter, bis er wieder einen Label findet. Auch dieser kommt mit Label-Nummer und Absolutadresse auf den Symbol-Stack. Dieser Vorgang wiederholt sich sooft, bis alle Labels aus dem eingetippten Programm verschwunden sind. Alles, was der Computer über die Labels wissen muß, ist jetzt auf dem Symbol-Stack abgelegt. Dieser Symbol-Stack ist ein 256 Byte langes Software Register, auf dem sich maximal 256:3 = 85 Labels ablegen lassen. Somit können mit dem Editor Programme eingegeben werden, die bis zu 85 Labels enthalten! Das ist mehr als genug, um Programme schrittweise zu assemblieren.

2. Schritt
Jetzt durchläuft der Assembler nochmals von oben, beginnend bei der Adresse Ø2ØØ, das gesamte Programm. Wir erinnern uns, alle Labels sind bereits entfernt. Der Assembler kümmert sich nun **ausschließlich** um folgende Maschinenbefehle:

- JMP-
- JSR-
- Branchbefehle wie BPL, BMI, BEQ, BNE, BCC, BCS, BVC und BVS.

Hinter diesen Befehlen **stehen noch** die Label-Nummern, zu denen gesprungen oder verzweigt werden soll. Trifft nun der Computer auf einen JMP- oder JSR-Befehl, sieht er nach, welche Labelnummer hinter dem Befehl steht. Zu jeder Label-Nummer gehört eine Absolutadresse. Diese Absolutadresse holt sich der Computer vom Symbol-Stack. Wie kann er jedoch auf dem Symbol-Stack diese Absolutadresse finden? Wir erinnern uns: Im ersten Schritt der Assemblierung rettete der Computer Label-Nummer und Absolutadresse des Labels auf den Symbol-Stack. Jetzt sucht der Computer auf dem Symbol-Stack die Label-Nummer auf, die hinter dem **momentan assemblierten** JMP- oder JSR-Befehl steht. Findet er auf dem Symbol-Stack diese Label-Nummer, holt er das hoch- und niederwertige Adreßbyte, das zur Label-Nummer gehört, vom Symbol-Stack und setzt diese hinter dem **momentan assemblierten** JMP- oder JSR-Befehl ein. Das heißt, die Label-Nummer und der Begrenzer, die beide dem Befehl folgen, werden vom hoch- und niederwertigen Adreßbyte der Absolutadresse überschrieben. Dieser Vorgang wiederholt sich sooft, bis alle JMP- und JSR-Befehle assembliert sind.
Auch für die Branch-Befehle müssen noch die Offsets berechnet werden. Deshalb sucht der Mikrocomputer im 2. Schritt auch alle Branchbefehle auf, die der Programmierer zuvor eingegeben hat. Hinter diesen Branchbefehlen stehen noch die Label-Nummern, zu denen die Verzweigung führen soll. Der Computer kann die Adresse, an welcher der Branchbefehl steht, leicht berechnen. Außerdem steht die Adresse, zu der verzweigt werden soll, auf dem Symbol-Stack. Für das Offset gilt:
Destination – Source + 2 = Offset oder übersetzt:
Adresse des Labels, zu dem verzweigt werden soll minus Adresse, an der

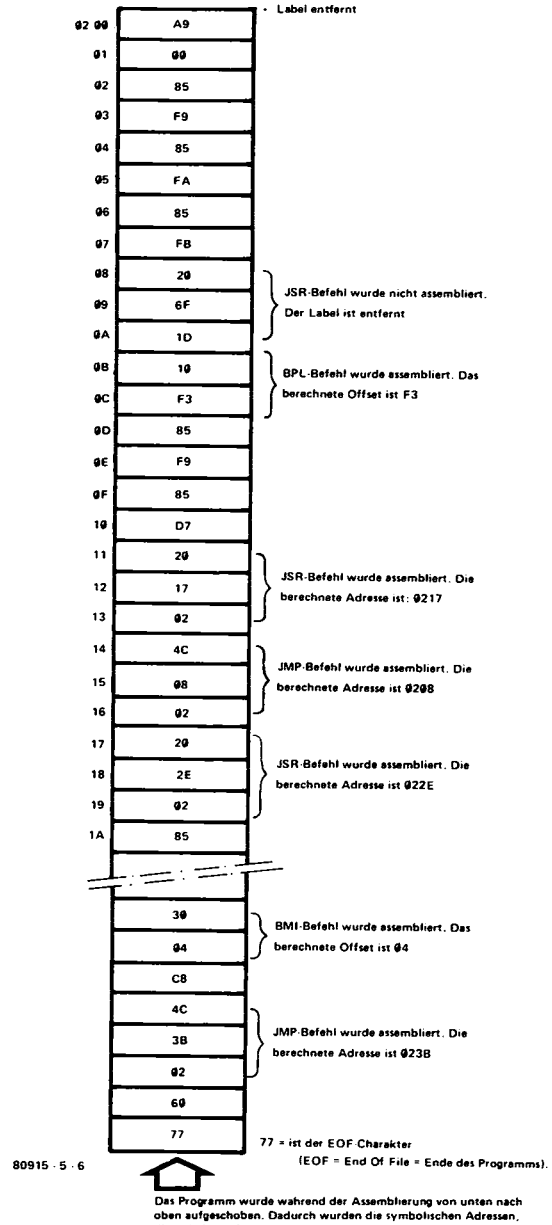


Bild 6. Nach der Assemblierung sind alle Labels aus der File verschwunden. Die File ist deshalb auch kürzer geworden. Hinter den OP-Codes der Sprung-Instruktionen stehen jetzt die richtigen absoluten Adressen. Auch den Branch-Instruktionen wurden die richtigen Offsets zugeordnet. In dieser Form kann die 6502-CPU das mit dem Editor eingegebene Programm verarbeiten.

sich Branchbefehl befindet plus zwei ist das Offset. Das assemblierte Programm von Bild 3 zeigt **Bild 6.**
Somit ist auch die Bedienung des Assemblers und sein Mechanismus besprochen. Ein **assembliertes Programm** läßt sich übrigens auch mit dem Editor durchlaufen. Dabei sollte man folgendes beachten:

- Der Editor sollte **nur** über Warm Start Entry gestartet werden.
- Die Pointer in Page Ø müssen manuell gesetzt werden:
  BEGAD = CURAD
  ENDAD = CEND (CEND = Current End Address)

Auf das Beispiel von Bild 3 übertragen, sind folgende Tasten zu drücken, um mit dem Editor ein bereits **assembliertes Programm** zu überstreichen:

```
AD Ø Ø E 6
DA Ø Ø          setze CURAD-Pointer auf Ø2ØØ
 + Ø 2
 + F F          setze CEND-Pointer auf Ø3FF
 + Ø 3
AD 1 C C A      Warm Start Entry
GO              Start des Editors
```

Durch diese Prozedur läßt sich ein assembliertes Programm mit der SKIP-Taste von oben nach unten durchlaufen. Mit dem SEARCH-Kommando hat der Programmierer die Möglichkeit, bestimmte Instruktionen im assemblierten Programm aufzusuchen. Die Tasten INSERT, INPUT und DELETE sollten nur dann verwendet werden, wenn durch Einfügen oder Herausholen von Instruktionen aus dem Speicher die vom Computer berechneten absoluten Adressen und die Länge der Verzweigungen unberührt bleiben.

Wichtige Adressen für Editor und Assembler

1) Editor:
Cold Start Entry : $1CB5
Warm Start Entry : $1CCA
Die Pointer des Editors:

BEGADL * $ØØE2 BEGADH * $ØØE3	}	Begin Address Pointer
ENDADL * $ØØE4 ENDADH * $ØØE5	}	End Address Pointer
CURADL * $ØØE6 CURADH * $ØØE7	}	Current Address Pointer
CENDL * $ØØE8 CENDH * $ØØE9	}	Current End Address Pointer

2) Assembler:
Startadresse: $1F51

3) Weitere Möglichkeiten für Editor und Assembler:
Die ST-Taste auf dem Keyboard des Junior-Computers ist mit der NMI-Leitung verbunden. Diese Leitung arbeitet mit den NMI-Vektor zusammen, den der Programmierer frei wählen kann. Der NMI-Vektor ist in folgenden Speicherzellen abgelegt:

```
NMIL * $1A7A    niederwertiges Adreßbyte
NMIH * $1A7B    hochwertiges Adreßbyte
```

Durch Laden der Startadressen in diese Speicherzellen kann der Programmierer den Editor **oder** Assembler mit der ST-Taste starten:

```
AD 1 A 7 A   Cold Start Entry via ST-Taste
DA B 5
 + 1 C
oder
AD 1 A 7 A
DA C A       Warm Start Entry via ST-Taste
 + 1 C
oder
AD 1 A 7 A   Start des Assemblers via ST-Taste
DA 5 1
 + 1 E
```

Das fünfte Kapitel hat uns also gezeigt, mit dem Editor und Assembler des Junior-Computers zu arbeiten. Durch diese Besonderheit des Monitorprogramms ist es möglich, Programme schnell und fehlerfrei in den Computer einzugeben. Der Computer nimmt dabei dem Programmierer eine Menge Arbeit ab. Erst durch den Editor und Assembler ist es möglich, Programme lückenlos im Speicher des Junior-Computers abzulegen. Das ist besonders wichtig, wenn nur ein begrenzter Speicherraum zur Verfügung steht.

6

PIA:

Der periphere Interface Adapter

Der Junior-Computer besitzt einen peripheren Interface Adapter, der wie die CPU in einem 40-Pin-Gehäuse untergebracht ist. Über diesen Adapter, kurz PIA genannt, läuft der gesamte Datenaustausch zur Außenwelt des Junior-Computers. Die Außenwelt, von der der Computer Daten empfängt oder zu der er Daten senden will, kann ein vielfältiges Gesicht haben: Ein hexadezimales Keyboard, ein vielstelliges 7-Segment-Display, ein ASCII-Keyboard, ein Videodisplay wie das ELEKTERMINAL, ein Drucker oder auch das Servo in einem Modellflugzeug. Wie der Computer mit Hilfe des PIA angeschlossene Geräte steuert oder Daten empfängt und weiterverarbeitet, beschreibt dieses Kapitel.

Ein Computer ohne Ein- und Ausgänge ist eine nutzlose Maschine. Der Mensch, dessen Arbeit der Computer erleichtern soll, kann am Programmablauf in der Maschine nicht teilnehmen oder eingreifen, wenn keine Ein- und Ausgänge vorhanden sind. Der Computer teilt sich also nicht mit. Das ist ähnlich wie bei Lehrern und Professoren, die einen hohen Wissensstandard erreicht haben, ihr Wissen jedoch nicht an die Schüler oder Studenten weitergeben können oder wollen. Beim Computer ist es dasselbe: je weniger ein Computer auf die Ungeschicktheit und Unkenntnis des Programmierers eingeht, desto unbrauchbarer ist er für den Menschen.
Wie kann sich nun ein Computer der Außenwelt mitteilen? Der Datenaustausch zwischen Mensch und Computer ist eine exakte Kopie des Informationsaustausches zwischen zwei Menschen. Um im Gehirn gespeicherte Gedanken oder Ideen weitergeben zu können, bedient sich der Mensch der Sprache. Auch beim Computer sind "Gedanken" und "Ideen" in Form von bearbeiteten Daten im Speicher abgelegt. Um diese Daten an die Außenwelt weitergeben zu können, bedient sich der Computer eines Druckers oder im einfachsten Falle eines sechsstelligen Displays, wie beim Junior-Computer.

Ein Gespräch zwischen zwei Menschen ist nur dann sinnvoll, wenn nicht nur gesprochen, sondern auch zugehört wird; denn zuhören ist genauso wichtig wie sprechen! Hören heißt jedoch nichts anderes, als vom Gesprächspartner Information zu empfangen. Damit ein Computer von der Außenwelt Daten empfangen kann, bedient er sich eines Keyboards. Bei großen Computern ist es eine ASCII-Tastatur, beim Junior-Computer ist es eine HEX-Tastatur.
Wie der Mensch mit dem Mund spricht und mit den Ohren hört, so sendet und empfängt der Junior-Computer über den peripheren Interface Adapter. Senden und Empfangen heißt jedoch, daß der periphere Interface Adapter – kurz PIA – Ein- und Ausgänge haben muß: Eingänge, um Daten zu empfangen; Ausgänge, um Daten zu senden. Unter Daten sind natürlich wieder elektrische Spannungen zu verstehen, die entweder log. Ø oder log. 1 sind und sich von Zeit zu Zeit ändern. Beim Junior-Computer ist ein 6532 als PIA eingesetzt. Diese LSI-Schaltung enthält neben dem peripheren Interface Adapter 128 RAM-Zellen, einen Timer, der sich universell programmieren läßt, ein FLAG-Register und einen Flanken-Detektor, der auf positive und negative Pulsflanken reagiert. Diese Register im IC 6532 präsentieren sich dem Programmierer wie normale Speicherzellen, aus denen der Mikroprozessor sowohl lesen oder in die er schreiben kann. Im ganzen gesehen enthält das IC 6532 neben den 128 RAM-Zellen noch 19 **besondere Register**, deren Programmierung und Funktionsweise im Verlauf dieses Kapitels genau beschrieben wird.
Die folgende Tabelle soll darüber Auskunft geben, wie diese 19 besonderen Register heißen:

1. Port A und Port B
   - PAD : Port A Data Register (Daten-Register)
   - PADD : Port A Data Direction Register (Datenrichtungs-Register)
   - PBD : Port B Data Register
   - PBDD : Port B Data Direction Register
2. Start des Timers, Timer-IRQ **ist nicht möglich**
   - CNTA : CLK1T (Teilungsfaktor ÷ 1)
   - CNTB : CLK8T (Teilungsfaktor ÷ 8)
   - CNTC : CLK64T (Teilungsfaktor ÷ 64)
   - CNTD : CLK1KT (Teilungsfaktor ÷ 1024)
3. Das Flag-Register und die Timer-Zellen
   - RDFLAG : lese das Flag-Register
   - RDTEN : lese die Timer-Zelle, enable Timer-IRQ
   - RDTDIS : lese die Timer-Zelle, disable Timer-IRQ

   - CNTE : CLK1T (Teilungsfaktor ÷ 1)
   - CNTF : CLK8T (Teilungsfaktor ÷ 8)
   - CNTG : CLK64T (Teilungsfaktor ÷ 64)
   - CNTH : CLK1KT (Teilungsfaktor ÷ 1024)
4. PA7 als Flankendetektor
   - EDETA : sensibel auf ↓ -Flanke, PA7-IRQ **ist nicht möglich**
   - EDETB : sensibel auf ↑ -Flanke, PA7-IRQ **ist nicht möglich**

– EDETC : sensibel auf ↓ -Flanke, PA7-IRQ **ist möglich**

– EDETD : sensibel auf ↑ -Flanke, PA7-IRQ **ist möglich**

Auch wenn die meisten Begriffe in dieser Tabelle noch unbekannt sind, lassen sich aus ihr die Namen der 19 besonderen Speicherzellen und der Aufbau dieses Kapitels entnehmen.
(Bemerkung: Manche Speicherzellen können übereinander liegen. Das heißt, daß zwei Speicherzellen mit verschiedenen Namen an ein und derselben Adresse liegen. Das kann zu Verwirrungen führen. Deshalb ist jedem symbolischen Namen einer Speicherzelle eine seperate Adresse zu geordnet. Verwechslungen von Speicherzellen und mögliche Doppeladressierungen sind somit ausgeschlossen).

Port A und Port B

Das IC 6532 hat zwei Ports: Port A und Port B. Beide Ports bilden zusammen den peripheren Interface Adapter, genannt PIA. Dieser periphere Interface Adapter besteht aus 16 Leitungen, die auf 16 Pins des ICs 6532 herausgeführt sind. Der Programmierer kann eigene Geräte, wie zum Beispiel einen Drucker, an den PIA anschließen. Beim Junior-Computer ist eine Steckerleiste vorgesehen, auf der die Leitungen des PIA herausgeführt sind: der Port-Konnektor. Aus Bild 1 ist ersichtlich, wie Keyboard und Display an den PIA angeschlossen sind. Es stehen dem Programmierer nicht alle 16 Leitungen des PIA zur Verfügung, wenn Keyboard und Display weiterhin verwendet werden sollen. Zur freien Verfügung stehen noch die Leitungen PB7 . . . PB5 sowie PBØ und PA7. Werden aber Keyboard und das Display des Junior-Computers in Programmen nicht mehr verwendet, stehen dem Programmierer alle Portleitungen zur Verfügung.

Das Programmier-Modell des PIA

Wie in Buch 1 (Kapitel 3, Bild 1b) ein Programmier-Modell für den Mikroprozessor gezeigt ist, so läßt sich auch ein Programmier-Modell für den PIA angeben. Dazu verwenden wir Bild 2.
Der PIA ist über drei Busse mit dem Mikroprozessor verbunden: dem Adreßbus, Datenbus und Controlbus. Er ist also wie ein normales Speicherelement an die CPU angeschlossen. Diese drei Busse haben folgende Funktionsweise:

– Der Adreßbus wählt interne Speicherzellen aus.
– Über den Datenbus läuft in zwei Richtungen der Datenaustausch zwischen CPU und PIA. Die Richtung des Datenaustausches steuert wieder die R/$\overline{W}$-Leitung des Prozessors.
– Über den Controlbus sendet die CPU zum PIA folgende Signale:
  * das R/$\overline{W}$-Signal
  * das Taktsignal $\Phi 2$
  * das CS-Signal
  * und das RES-Signal (Reset)

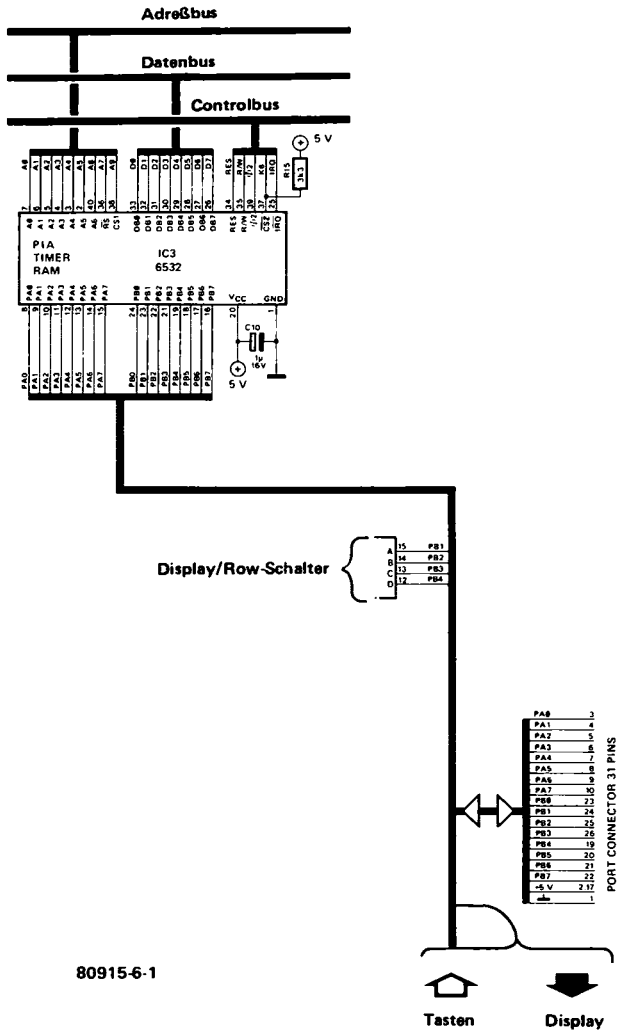
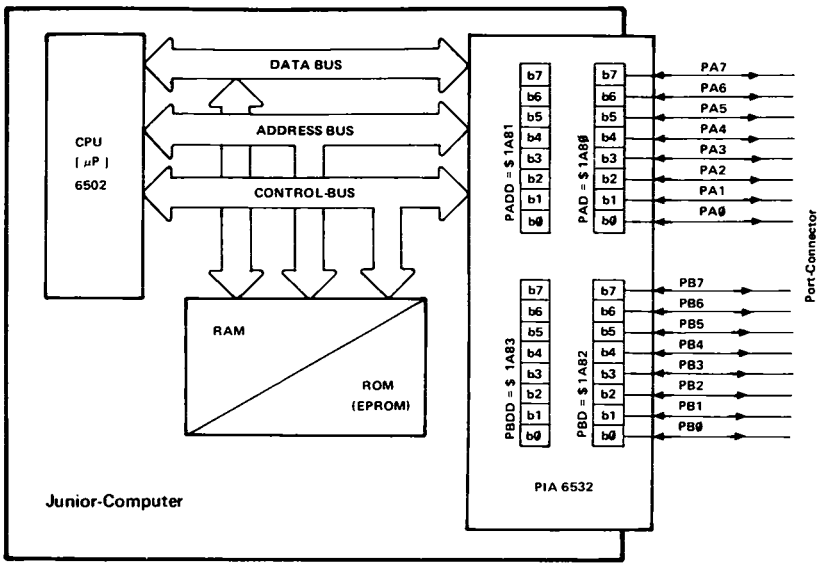


Bild 1. Der PIA ist über den Adreß- Daten- und Controlbus mit der CPU verbunden. Der Adreßbus wählt verschiedene Speicherzellen im IC 6532 an. Nicht alle Port-Leitungen stehen dem Programmierer zur Verfügung: An Port A wie Port B sind das Tastenfeld und das Display angeschlossen. Alle 16 Port-Leitungen sind auf den Port-Konnektor herausgeführt.

Auch die IRQ-Leitung des PIA ist über den Controlbus mit der CPU des Junior-Computers verbunden. Der PIA kann also über die IRQ-Leitung einen IRQ auslösen. Wie das im einzelnen funktioniert, besprechen wir beim Timer und bei der Programmierung von PA7 (Punkt 2, 4 und 5).



PAD = Port A Data Register
PADD = Port A Data Direction Register
PBD = Port B Data Register
PBDD = Port B Data Direction Register (Übersetzung siehe Text)

Bild 2. Der PIA (= Peripherer Interface Adapter) besteht aus vier Registern und den beiden Ports PORT A und PORT B. Ein Port ist ein Leitungsbündel von acht Leitungen, über die der Mikrocomputer Daten an die Außenwelt weiterleitet oder über die er Daten von der Außenwelt empfängt. Jede Port-Leitung läßt sich unabhängig von der anderen sowohl als Eingang wie als Ausgang programmieren. Dabei bestimmt das Bitmuster in den Datenrichtungs-Registern PADD und PBDD, ob eine Port-Leitung als Eingang oder Ausgang erklärt ist. Eine "Ø" an einer bestimmten Bitposition im Datenrichtungs-Register erklärt die korrespondierende Portleitung als Eingang, eine "1" als Ausgang. Über den Datenbus kann der Prozessor die Daten-Register lesen oder in diese schreiben. Bei einer Leseoperation liest der Prozessor das Bitmuster auf den Port-Leitungen. Schreibt die CPU in das Daten-Register PAD oder PBD ein Bitmuster, erscheint das Bitmuster auf den zu Ausgängen deklarierten Port-Leitungen. Nach der Schreiboperation bleibt das Bitmuster auf den Port-Leitungen erhalten (Latchfunktion einer als Ausgang deklarierten Port-Leitung). Der PIA ist über den Adreßbus, Datenbus und den Controlbus mit der CPU verbunden.

Somit kennen wir alle elektrischen Verbindungen zwischen PIA und Mikroprozessor und einer Besprechung des Programmier-Modelles steht nichts mehr im Wege. Bei der Beschreibung des peripheren Interface Adapters lassen sich Hardware und Software nicht mehr trennen. Deshalb präsentiert sich der PIA dem Programmierer folgendermaßen:

– **Hardware:** 16 Leitungen, die sich unabhängig voneinander als Eingänge oder Ausgänge programmieren lassen. Diese 16 Leitungen sind in zwei Ports aufgeteilt: Port A und Port B. Beide Ports bestehen aus je 8 Lei-

tungen. Man sagt auch, Port A wie Port B sind 8 Bit breit. Die Leitungen der Ports haben die Namen PAØ . . .PA7 und PBØ . . . PB7.
Auf die einzelnen Port-Leitungen ist der Datenbus herausgeführt. Da zwei Ports zur Verfügung stehen, läßt sich der Datenbus wahlweise auf Port A oder Port B durchschalten.
Da der Datenbus bidirektional ist, kann die CPU über den Datenbus Daten auf die Port-Leitungen schreiben oder Daten, die als Spannungen auf den Port-Leitungen liegen (log. Ø oder log. 1), lesen. Schreibt der Mikroprozessor auf Port-Leitungen, die zu Ausgängen erklärt wurden, ein bestimmtes Bitmuster, bleibt dieses Bitmuster auf den Ausgangs-Port-Leitungen auch nach der Schreibe-Operation erhalten (**Latch-Funktion** einer Ausgangs-Leitung).
Liest der Mikroprozessor eine oder mehrere Port-Leitungen, die zu Eingängen erklärt wurden, liest er das momentane Spannungsniveau auf der Port-Leitung. Ist die Spannung an der entsprechenden Port-Leitung kleiner als 0,4 V, liest die CPU ein log. Ø-Niveau; ist die Spannung an der Port-Leitung größer als 2,4 V, handelt es sich um ein log. 1-Niveau (**keine Latch-Funktion** einer Eingangsleitung).

– **Software**: 4 Register, in die sich Daten schreiben, oder aus denen sich Daten lesen lassen.
Die Namen der vier internen Register sind:
PAD = Port A Data Register (Daten-Register)
PADD = Port A Data Direction Register (Datenrichtungs-Register)
PBD = Port B Data Register (Daten-Register)
PBDD = Port B Data Direction Register (Datenrichtungs-Register)
Wie aus den Bezeichnungen dieser Register ersichtlich ist, besteht jeder Port aus zwei Speicherzellen:
1. Einem Daten-Register und
2. einem Datenrichtungs-Register.
Das Datenrichtungs-Register steuert für jeden Port die Flußrichtung der Daten auf den Portpins. Da dieses Register 8 Bit breit ist, kann die CPU über den Datenbus ein Wort in das Datenrichtungs-Register schreiben. Dieses Wort ist ein Bitmuster aus Nullen und Einsen. Dieses Null-Eins-Muster im Datenrichtungs-Register hat auf die Port-Leitungen folgende Auswirkung:
  – Eine Null in einem beliebigen Bit des Datenrichtungs-Register programmiert die korrespondierende Port-Leitung als Eingang.
  – Eine Eins in einem beliebigen Bit des Datenrichtungs-Registers programmiert die korrespondierende Port-Leitung als Ausgang.

Beiden Registern eines Ports sind Adressen zugeordnet. Beim Junior-Computer liegen die Port-Register an folgenden Adressen:
PAD hat die Adresse 1A8Ø
PADD hat die Adresse 1A81
PBD hat die Adresse 1A82
PBDD hat die Adresse 1A83
Folgende Beispiele sollen uns mit der Praxis der Port-Programmierung vertraut machen. Zunächst sollen einige Port-Leitungen als Eingang oder Ausgang programmiert werden:
1. Alle Port-Leitungen von Port A sollen Eingänge, alle Port-Leitungen von Port B sollen Ausgänge sein. Das Programm sieht dann so aus:

LDA # ØØ alle Bits im Accu sind null
STA-PADD die Leitungen PAØ . . . PA7 sind Eingänge
LDA # FF alle Bits im Accu sind eins
STA-PBDD die Leitungen PBØ . . . PB7 sind Ausgänge.
Wie bereits bekannt ist, bestimmen die einzelnen Bits im Datenrichtungsregister der Ports, ob eine Port-Leitung ein Eingang oder ein Ausgang ist. Im Datenrichtungs-Register PADD sind alle Bits null. Deshalb sind alle Leitungen von Port A als Eingänge programmiert. Die CPU kann jetzt die Spannungen, die auf den einzelnen Port-Leitungen stehen als Null-Eins-Muster in den Accu, das X-Register oder das Y-Register lesen. Doch das besprechen wir zu einem späteren Zeitpunkt.
Im Datenrichtungs-Register PBDD sind alle Bits eins. Deshalb sind alle Leitungen von Port B als Ausgänge programmiert. Die CPU kann nun über den Datenbus ein beliebiges Bitmuster auf die Port-Leitungen schreiben. Auch diese Funktionsbeschreibung folgt zu einem späteren Zeitpunkt.
2. Die Port-Leitungen PA4 und PBØ sollen Ausgänge sein. Alle anderen Port-Leitungen sollen als Eingänge wirken. Dazu schreiben wir die Bits in eine Zeile, wie sie in der beiden Datenrichtungs-Registern stehen müssen:

PADD	b7	b6	b5	b4	b3	b2	b1	bØ	
	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	= $1Ø
PBDD	b7	b6	b5	b4	b3	b2	b1	bØ	
	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	= $Ø1

Das Programm sieht dann so aus:

LDA # 1Ø	lade Accu mit Bitmuster
STA-PADD	die Leitung PA4 ist Ausgang
LDA # Ø1	lade Accu mit Bitmuster
STA-PBDD	die Leitung PBØ ist Ausgang, alle übrigen Leitungen von Port A und Port B sind als Eingänge programmiert.

Lesen von und Schreiben auf die Port-Leitungen

Bis jetzt wissen wir erst, daß die Port-Leitungen mit Hilfe der Datenrichtungs-Register zu Eingängen oder zu Ausgängen programmiert werden können. Wie kann nun der Mikroprozessor Daten (in Form von Nullen und Einsen) von den Port-Leitungen lesen oder auf die Port-Leitungen schreiben? Dafür sind die beiden Daten-Register PAD und PBD vorgesehen. Die Adresse von PAD ist 1A8Ø und die von PBD ist 1A82.

Lesen der Port-Leitungen PAØ . . . PA7 und PBØ . . . PB7 (Input Mode "I")

Beim Lesen der Daten-Register PAD oder PBD wird das Bitmuster, das an den Port-Leitungen anliegt, über den Datenbus in eines der internen CPU-Register transportiert. Das heißt, die Port-Leitungen sind eine Verlängerung des Datenbusses. **Will der Prozessor eine oder mehrere Port-Leitungen lesen,**

so müssen diese als Eingänge programmiert sein. Das folgende Beispiel macht das deutlich:

```
LDA # ØØ
STA-PADD  PAØ . . . PA7 sind als Eingänge programmiert
LDX-PAD   das Bitmuster, das an den Port-Leitungen anliegt, gelangt
          ins X-Register der CPU.
```

Liegt an Port A das Bitmuster 1Ø1Ø111Ø an,

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ	Port A Leitungen
1	Ø	1	Ø	1	1	1	Ø	Bitmuster an PA7 . . .PAØ

dann steht im X-Register AE. Ist die Spannung an einer Port-Leitung kleiner als 0,4 V, interpretiert sie der Mikroprozessor als log. Ø, ist die Spannung jedoch größer als 2,4 V, handelt es sich um log. 1.

Bemerkung: Wird eine als Eingang programmierte Port-Leitung von einem Gatter, einem Transistor oder einem anderen Logikbaustein angesteuert, müssen diese Bausteine einen Strom von mindestens 1,6 mA liefern können. Das ist besonders beim Anschluß diverser CMOS-Schaltungen zu beachten. Liest der Mikroprozessor die Port-Leitungen, gelangt **das momentan anliegende Bitmuster** in eines der internen CPU-Register. Als Eingänge programmierte Port-Leitungen haben **keine** Latch-Funktion.

Schreiben auf die Port-Leitungen PAØ . . . PA7 oder PBØ . . .PB7 (Output Mode "0")

Beim Schreiben in die Daten-Register PAD oder PBD, wird das Bitmuster, das in einem der CPU-Register steht, auf die Port-Leitungen geschrieben. Auch bei einer Schreibeoperation sind die Port-Leitungen eine Verlängerung des Datenbusses. **Will der Prozessor auf eine oder mehrere Port-Leitungen schreiben, so müssen diese als Ausgänge programmiert sein.**

Beispiel:

```
LDA # FF
STA-PADD  PAØ . . . PA7 sind als Ausgänge programmiert
LDX # C3
STX-PAD   das Bitmuster im X-Register wird auf die Port A-Leitungen
          geschrieben.
```

Der Inhalt des X-Registers ist C3 = 11ØØØØ11. Dieses Bitmuster steht jetzt als Null-Eins-Muster auf den Leitungen PA7 . . . PAØ, also

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ	Port A Leitungen
1	1	Ø	Ø	Ø	Ø	1	1.	

Auf den Port-Leitungen bleibt die Hexzahl C3 auch nach der Schreibeoperation erhalten (Latch-Funktion).

Bemerkung: Steuert eine als Ausgang programmierte Port-Leitung ein Gatter, einen Transistor oder einen anderen Logikbaustein an, dann dürfen diese Bausteine nicht mehr als 1,6 mA Eingangsstrom "schlucken". Die Port-Leitungen können also eine Standard TTL-Last treiben. Verzichtet man auf TTL-Kompatibilität und steuert mit den Leitungen PBØ . . . PB7

Transistoren, können diese Port-Leitungen einen Ausgangsstrom von 3 mA bei einer Ausgangsspannung von 1,5 V liefern (Current Source).

Achtung!!! Die Port-Leitungen des ICs 6532 sind gegen Überspannung und Überstrom **nicht** geschützt. Beim direkten Anschluß von externen Geräten, wie einem Drucker, ist darauf zu achten, daß die Spannung an den Port-Leitungen niemals +7 V übersteigt oder gar negativ wird. Sonst wird das IC 6532 zerstört.

Die ersten Schritte

Wir haben jetzt alles wissenwerte über die Port-Leitungen erfahren. Es ist nun höchste Zeit, ein kleines Übungsprogramm zu schreiben, in dem die vier I/O-Register des ICs 6532 Verwendung finden. Das Programm soll folgendes bewirken:

1. An Port A soll die Schalterstellung von acht Schaltern in die CPU gelesen werden.
2. Diese Schalterstellung soll der Junior-Computer in eine hörbare Frequenz umwandeln.
3. Die Frequenz soll an PBØ zur Verfügung stehen. Ein Verstärker ist vorzusehen, damit sich ein kleiner 8-Ohm-Lautsprecher direkt ansteuern läßt.
4. Ein Schaltbild ist zu zeichnen, das diesen Anforderungen genügt.
5. Es ist ein Flußdiagramm zu zeichnen, das mit dem Editor in den Junior-Computer eingegeben und anschließend assembliert wird.

Die Punkte 1 . . . 3 sind Forderungen, die das Programm erfüllen soll. Die Lösung dieses Problemes beginnt bei Punkt 4: Zeichnen eines Schaltbildes. Dieses Schaltbild ist in Bild 3a dargestellt. An die Leitungen PAØ . . . PA7

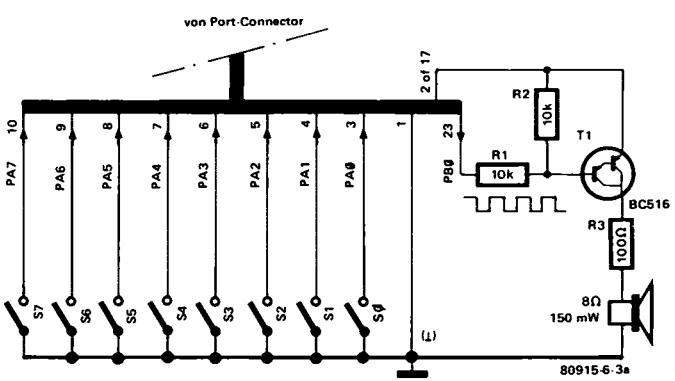


Bild 3a. Die Schalterstellung der Schalter SØ . . .S7 repräsentiert ein Bitmuster, das über PORT A in den Junior-Computer eingelesen wird. Alle Port-Leitungen von PORT A sind deshalb zu Eingängen programmiert. Der Computer setzt die Schalterstellungen in eine hörbare Frequenz um. An PBØ ist ein Verstärker, der einen Lautsprecher treibt, angeschlossen. PBØ ist deshalb als Ausgang programmiert. Auf dieser Portleitung erscheint die hörbare Rechteckspannung.

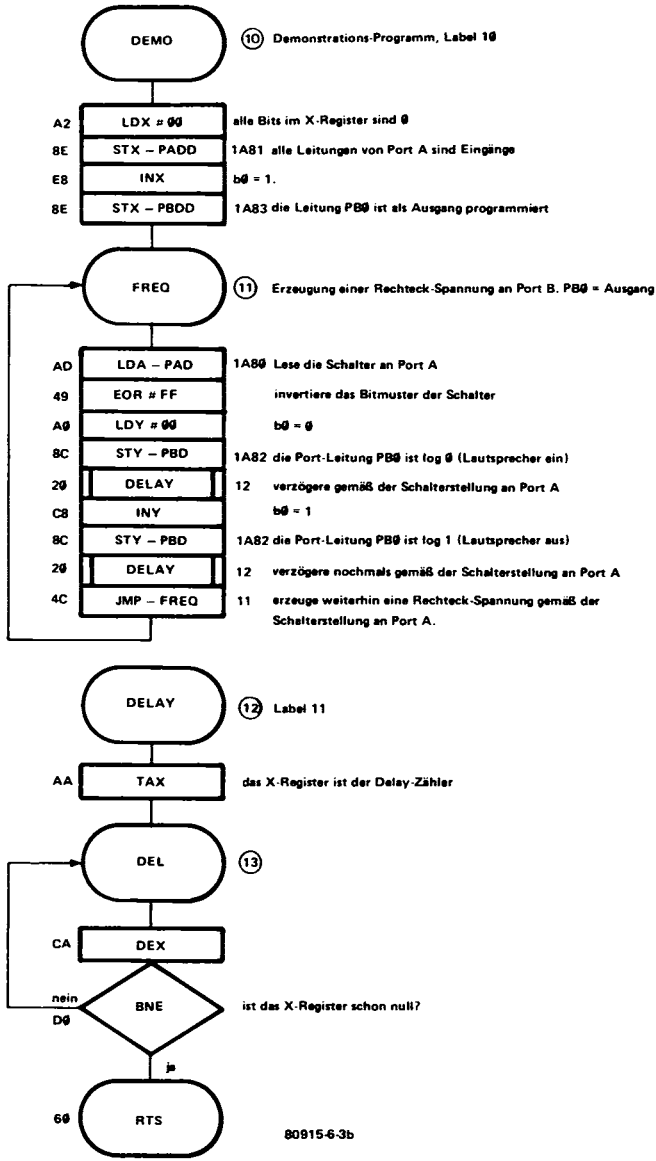


Bild 3b. Das Programm DEMO liest die Schalterstellungen der Schalter S0 . . . S7 über Port A in den Junior-Computer. Ein geschlossener Schalter entspricht einem log. 1-Niveau, ein offener Schalter einem log. 0-Niveau. Ist die Schalter-Stellung in den Computer eingelesen, wandelt diese das DEMO-Programm in eine hörbare Frequenz um. Der Computer sendet diese Frequenz über PB0 und einen Verstärker an die Außenwelt.

sind die acht Schalter angeschlossen: SØ . . . S7. Da der Junior-Computer diese Schalterstellungen lesen soll, sind die Leitungen von Port A als Eingang zu programmieren. An der Leitung PBØ soll eine hörbare Frequenz erscheinen. Deshalb ist PBØ als Ausgang zu programmieren. Damit diese Frequenz mit einem Lautsprecher hörbar wird, ist der Transistor T1 vorgesehen. Die benötigte Hardware zur Lösung des Problemes ist somit entwickelt.
Als nächstes folgt die Entwicklung der Software: Das Programm (Punkt 5). Ein Programm, das eine bestimmte Schalterstellung in eine Frequenz bzw. Rechteckspannung umwandelt, zeigt Bild 3b. Es ist darauf hinzuweisen, daß dieses Problem nicht optimal geschrieben ist. Aber es soll ja nicht schön sein, sondern die Steuerung der vier Register PAD, PADD, PBD und PBDD demonstrieren und sie besonders leicht verständlich machen.
Am Anfang des Programms DEMO werden zunächst die Port-Leitungen PAØ . . .PA7 als Eingänge, die Port-Leitung PBØ als Ausgang programmiert: Alle Bits im Datenrichtungs-Register PADD sind null, bØ im Datenrichtungs-Register PBDD ist eins. Daran wird sich im Verlauf des Programms nichts mehr ändern.
Es folgt die Frequenzschleife unseres Programms (Label FREQ). Dort liest der Prozessor die Stellungen der Schalter SØ . . . S7 in den Accu. Dazu müssen wir noch definieren, ob ein geschlossener Schalter einem log. Ø- oder log. 1-Niveau entspricht. Es gilt:
– ein geschlossener Schalter ist log. 1
– ein offener Schalter ist log. Ø
Nachdem das Bitmuster auf den Port-Leitungen in den Mikro-Prozessor gelesen ist (LDA-PAD), muß es im Accu invertiert werden: EOR # FF. Somit ist die Definition erfüllt. In Abhängigkeit der Schalterstellung soll nun ein hörbarer Ton erzeugt werden. Das läßt sich am einfachsten dadurch verwirklichen, indem die als Ausgang deklarierte Port-Leitung PBØ für eine bestimmte Zeit auf log. Ø und anschließend auf log. 1 gesetzt wird. An PBØ entsteht dann eine Rechteckspannung, die im angeschlossen Lautsprecher hörbar ist.
Das Setzen und Zurücksetzen der Leitung PBØ steuert das Y-Register: Das niederwertigste Bit bØ wird abwechselnd null oder eins gesetzt und dann ins Daten-Register PBD geschrieben. Wie lange PBØ null oder eins ist, bestimmt die Subroutine DELAY. Dort kopiert die CPU das Bitmuster der Schalterstellung ins X-Register (TAX). Dann wird das X-Register solange dekrementiert, bis es null geworden ist. Dieses Dekrementieren dauert in Abhängigkeit der Schalterstellungen kürzer oder länger. Somit erhalten wir an PBØ eine höhere oder niedrigere Frequenz.
Bemerkung: Wird ein bestimmtes Bitmuster auf eine oder mehrere Port-Leitungen geschrieben, bleibt es auch nach der Schreibeoperation erhalten (Latch-Funktion einer als Ausgang programmierten Port-Leitung). Erst ein abermaliges Schreiben eines neuen Bitmusters in das Daten-Register PAD oder PBD verändert das Bitmuster auf den Port-Leitungen.

Das Editen und Assemblieren des DEMO-Programms

Nachdem das Flußdiagramm für DEMO gezeichnet ist, können wir das Programm in den Junior-Computer eingeben. Wir wollen dieses Programm

in Page Ø assemblieren. Dort stehen die Speicherzellen mit den Adressen ØØØØ . . . ØØEØ zur freien Verfügung. Deshalb setzen wir den BEGAD-Pointer auf ØØØØ und den ENDAD-Pointer auf ØØEØ. Den Assembler starten wir mit der ST-Tase, das heißt, über einen NMI wird der Editor verlassen und der Assembler gestartet. Deshalb muß vor dem Editen der NMI-Vektor auf die Assembler-Startadresse 1F51 gesetzt werden. Das Tastenprogramm in Bild 3c veranschaulicht die Eingabe des DEMO-Programms.

Drücke Taste:		Display:	Bemerkung:
RST			
AD	Ø Ø E 2	ØØE2 XX	
DA	Ø Ø	ØØE2 ØØ	BEGAD = ØØØØ
+	Ø Ø	ØØE3 ØØ	
+	E Ø	ØØE4 EØ	ENDAD = ØØEØ
+	Ø Ø	ØØE5 ØØ	
AD	1 A 7 A	1A7A XX	NMI-Vektor
DA	5 1	1A7A 51	= 1F51
+	1 F	1A7B 1F	Startadresse Assembler
AD	1 C B 5	1CB5 2Ø	Startadresse Editor
GO		77	
INSERT	F F 1 Ø Ø Ø	FF 1Ø ØØ	Label 1Ø: DEMO
INPUT	A 2 Ø Ø	A2 ØØ	LDX # ØØ
INPUT	8 E 8 1 1 A	8E 81 1A	STX-PADD
INPUT	E 8	E8	INX
INPUT	8 E 8 3 1 A	8E 83 1A	STX-PBDD
INPUT	F F 1 1 Ø Ø	FF 11 ØØ	Label 11: FREQ
INPUT	A D 8 Ø 1 A	AD 8Ø 1A	LDA-PAD
INPUT	4 9 F F	49 FF	EOR # FF
INPUT	A Ø Ø Ø	AØ ØØ	LDY # ØØ
INPUT	8 C 8 2 1 A	8C 82 1A	STY-PBD
INPUT	2 Ø 1 2 Ø Ø	2Ø 12 ØØ	JSR-DELAY (Label Nummer 12)
INPUT	C 8	C8	INY
INPUT	8 C 8 2 1 A	8C 82 1A	STY-PBD
INPUT	2 Ø 1 2 Ø Ø	2Ø 12 ØØ	JSR-DELAY (Label Nummer 12)
INPUT	4 C 1 1 Ø Ø	4C 11 ØØ	JMP-FREQ (Label Nummer 11)
INPUT	F F 1 2 Ø Ø	FF 12 ØØ	Label 12: DELAY
INPUT	A A	AA	TAX
INPUT	F F 1 3 Ø Ø	FF 13 ØØ	Label 13: DEL
INPUT	C A	CA	DEX
INPUT	D Ø 1 3	DØ 13	BNE zum Label 13
INPUT	6 Ø	6Ø	RTS
ST		XX XX XX	Start Assembler via NMI
AD	Ø Ø Ø Ø	ØØØØ A2	Startadresse DEMO
GO			Start DEMO

Bild 3c. So läßt sich das DEMO-Programm von Bild 3a mit dem Editor in den Junior-Computer eingeben und anschließend assemblieren. Das DEMO-Programm wird dabei in Page Ø abgelegt. Der Start des Assemblers erfolgt über den NMI-Vektor durch Drücken der ST-Taste. Sind die Schalter SØ . . . S7 und der Verstärker am Portkonnektor angeschlossen, dann läßt sich das DEMO-Programm an der Adresse ØØØØ starten. Sind alle Schalter geschlossen, dann ist ein tiefer Tor hörbar.

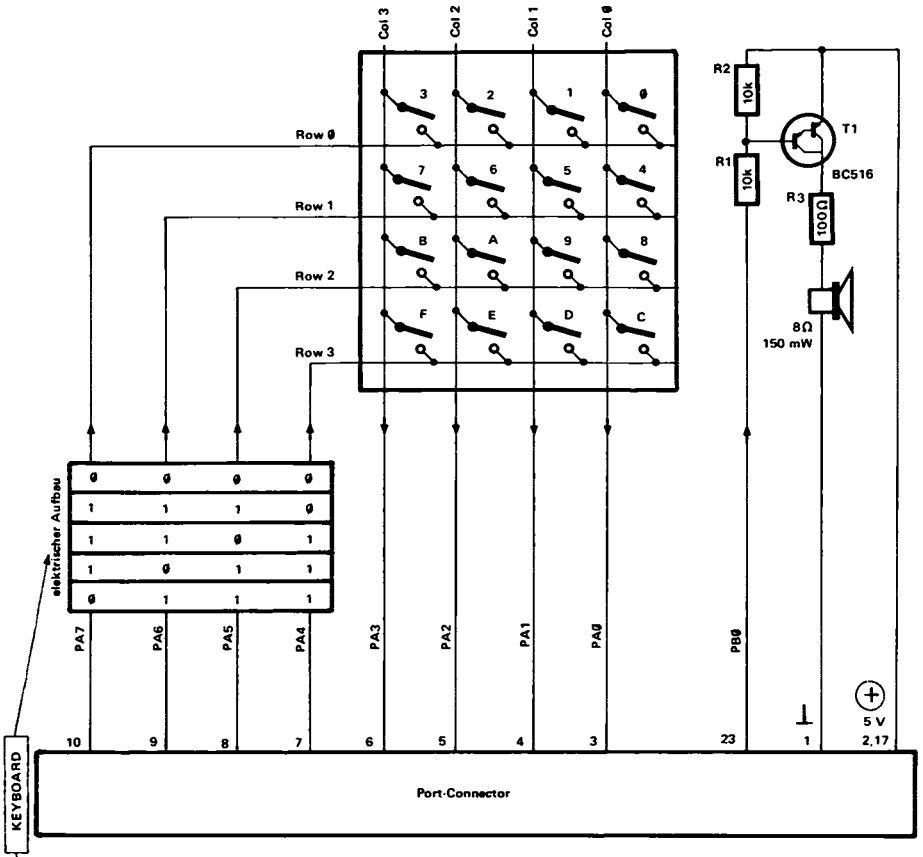
Musik auf dem Junior-Computer

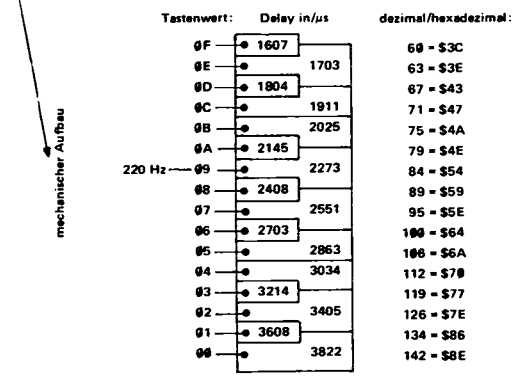
Mit dem Junior-Computer lassen sich auch Melodien spielen. Dazu schließen wir eine kleine Klaviatur, bestehend aus Schaltern, an den Port-Leitungen an. Die gespielte Melodie soll über einen Lautsprecher hörbar sein. Folgende Forderungen sollen dabei erfüllt sein:

1. Die Klaviatur soll, wie ein normales Piano-Keyboard, weiße und schwarze Tasten haben. Das an den Computer angeschlossene Tastenfeld soll mit dem Original übereinstimmen.
2. Elektrisch sollen die Tasten in einer Matrix angeordnet sein. Das gesamte Tastenfeld soll an Port A angeschlossen werden.
   Für das Betreiben des Lautsprechers ist ein Verstärker vorzusehen.
   Faßt man diese Forderungen zusammen, dann läßt sich eine Schaltung gemäß Bild 4a angeben. Dort finden wir die Anordnung
   - einer Piano-Klaviatur
   - einer Tasten-Matrix mit den Anschlüssen an Port A
   - einer Verstärkerschaltung, die an die Port-Leitung PBØ angeschlossen ist.
3. Da die Tasten (Bild 4a) in einer Matrix angeordnet sind, läßt sich jeder Taste eindeutig ein bestimmter Wert zuordnen. Das Format der Matrix ist 4 mal 4. Die Reihen haben die Bezeichnungen ROWØ . . . ROW3 und die Spalten COLØ . . . COL3. Auch die Werte der einzelnen Tasten sind in der Matrix angegeben. Bild 4b zeigt den praktischen Aufbau der Klaviatur mit Digitaster.
4. Da alle Tasten an nur einen Port (PORT A) angeschlossen sind müssen einige Port-Leitungen als Eingang, andere als Ausgang programmiert sein. Es soll gelten:
   - Die Spalten COLØ . . . COL3 sind mit PAØ . . . PA3 verbunden. Da der Computer zur Berechnung des Tastenwerts das Spalten-Muster liest, sind die Port-Leitungen PAØ . . . PA3 als Eingänge zu programmieren.
   - Die Zeilen ROWØ . . . ROW3 sind mit PA4 . . . PA7 verbunden. Diese Zeilen müssen nacheinander log. Ø werden, damit der Computer eine gedrückte Taste erkennen und berechnen kann. Da der Computer auf die Matrix-Reihen ROWØ . . . ROW3 ein bestimmtes Bitmuster schreibt, sind die Port-Leitungen PA7 . . . PA4 als Ausgänge zu programmieren.
5. Die Steuerung eines Lautsprechers mit einem Verstärker ist bereits bekannt. Die Port-Leitung PBØ ist als Ausgang zu programmieren.

Nun ist der elektrische und mechanische Aufbau der Piano-Klaviatur klar. Jetzt müssen wir nur noch ein kleines Programm entwickeln, damit sich auf der Klaviatur eine Melodie spielen läßt. Auch hier gehen wir wieder schrittweise vor und stellen die Programme zusammen, die benötigt werden:

- Ein Programm, das die Tastenwerte der 16 Klaviatur-Tasten berechnet.
- Ein Programm, das feststellt, ob irgendeine Klaviatur-Taste gedrückt ist.
- Ein Programm, das für die Entprellung der Tasten sorgt.
- Ein Programm, das jeder gedrückten Klaviatur-Taste eine bestimmte Frequenz zuordnet.
- Eine Lookup Table, in der die 16 Frequenzen für die Klaviatur-Tasten abgelegt sind.


Col 3
Col 2
Col 1
Col 0
Row 0
Row 1
Row 2
Row 3
3
2
1
0
7
6
5
4
B
A
9
8
F
E
D
C
R2
10k
R1
10k
T1
BC516
R3
100Ω
8Ω
150 mW
elektrischer Aufbau
PA7
PA6
PA5
PA4
PA3
PA2
PA1
PA0
PB0
10
9
8
7
6
5
4
3
23
1
5 V
2,17
KEYBOARD
Port-Connector


Tastenwert:
Delay in/µs
dezimal/hexadezimal:
0F
1607
60 = $3C
0E
1703
63 = $3E
0D
1804
67 = $43
0C
1911
71 = $47
0B
2025
75 = $4A
0A
2145
79 = $4E
220 Hz
09
2273
84 = $54
08
2408
89 = $59
07
2551
95 = $5E
06
2703
100 = $64
05
2863
106 = $6A
04
3034
112 = $70
03
3214
119 = $77
02
3405
126 = $7E
01
3608
134 = $86
00
3822
142 = $8E
mechanischer Aufbau

80915-6-4a

Ein Programm, das die Tastenwerte der 16 Klaviatur-Tasten berechnet, zeigt Bild 4c. Da dieses Programm eine Abfrage der Tasten-Matrix von Bild 4a durchführt, müssen die Port-Leitungen von Port A definiert sein, bevor der Prozessor in die Subroutine KEYVAL springt. Wie bekannt, sind die Port-Leitungen PA7 . . . PA4 als Ausgänge und PA3 . . . PAØ als Eingänge programmiert. Deshalb muß das Datenrichtungs-Register von PORT A das Bitmuster 1111ØØØØ = FØ enthalten. Diese Forderung läßt sich wie folgt erfüllen:

LDA # FØ
STA-PADD.

Alle Port-Leitungen sind jetzt definiert und die Subroutine KEYVAL kann aufgerufen werden. Bei der Besprechung nehmen wir die Tasten-Matrix von Bild 4a und das Flußdiagramm von Bild 4c zur Hand:
Am Anfang des Programms enthält die Speicherzelle ROW F7 = 1111Ø111. Das X-Register arbeitet als Reihenzähler, das heißt, sein Inhalt gibt darüber Auskunft, welche Reihe der Keyboard-Matrix über PORT A in den Computer eingelesen wird:

X = Ø3 entspricht ROWØ
X = Ø2 entspricht ROW1
X = Ø1 entspricht ROW2
X = ØØ entspricht ROW3

Nach jeder Dekrementierung des X-Registers schiebt der Prozessor den Inhalt der Speicherzelle ROW um eine Bitposition nach links und speichert diesen im Daten-Register ab. Das hat zur Folge, daß die Zeilen der Matrix hintereinander log. Ø werden:

X-Register:	PA7 . . . PA4:
X = Ø4	1111
X = Ø3	111Ø
X = Ø2	11Ø1
X = Ø1	1Ø11
X = ØØ	Ø111

Da die einzelnen Matrixreihen nacheinander auf log. Ø gehen, kann der Prozessor leicht feststellen, ob in der Reihe eine Taste gedrückt ist. Ein Lesen der Port-Leitungen PA3 . . . PAØ (Lesen der Spalten) gibt darüber Aufschluß:

– Ist keine Taste gedrückt, dann gilt:
  PA3 . . . PAØ ist 1111. Das niederwertigste Nibble des Datenbytes enthält also nur Einsen.

Bild 4a. Beim PLAY-Programm (Bild 4g) schließen wir ein echtes Piano-Keyboard an den Junior-Computer an. Für jede gedrückte Klaviaturtaste muß der Computer eine Frequenz erzeugen. Der untere Teil des Bildes zeigt die Anordnung der weißen und schwarzen Pianotasten. Die Zahlen auf den Tasten sind die halben Periodendauern des zu erzeugenden Tones in µs. Jeder Klaviaturtaste ist ein bestimmter Tastenwert zugeordnet (ØØ . . . ØF). Der rechte Teil des Bildes zeigt die Anordnung der Tastenmatrix. Diese Tastenmatrix ist elektrisch mit PORT A verbunden. Dabei sind die Leitungen PAØ . . . PA3 als Eingänge und die Leitungen PA4 . . . PA7 als Ausgänge deklariert. Der Lautsprecher ist wieder über einen Verstärker an PBØ angeschlossen.

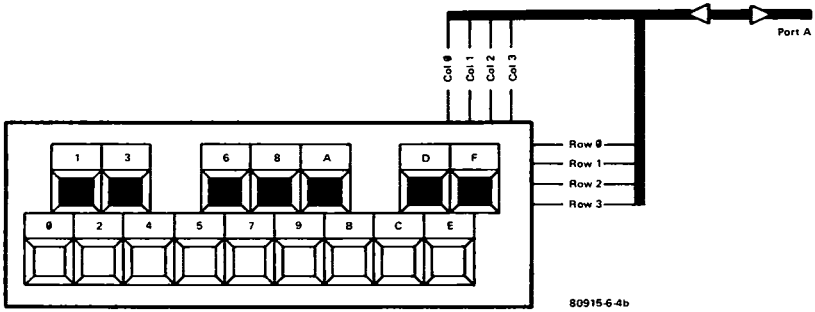


Bild 4b. So läßt sich die Piano-Klaviatur mit weißen und schwarzen Digi-Tastern aufbauen. Elektrisch sind diese 16 Tasten in einer 4 mal 4 Matrix angeordnet; mechanisch in zwei Reihen. Auch die Tasten-Werte der einzelnen Tasten sind eingezeichnet.

– Ist eine Taste gedrückt, dann gilt:
PA3 . . . PAØ ist 111Ø oder
11Ø1 oder
1Ø11 oder
Ø111 oder das niederwertige Nibble enthält mehrere Nullen, falls mehrere Tasten in der Matrixzeile gleichzeitig gedrückt sind.

Da beim Lesen der Matrix-Spalten die gesamte Breite von PAD (LDA-PAD) gelesen wird, also auch das nichtinteressierende hochwertige Daten-Nibble in den Accu kommt, muß man es maskieren.

LDA-PAD hochwertiges Nibble von PAD steht im Accu und ist gleich dem hochwertigen Nibble der Speicherzelle ROW!

AND # ØF Maskierung auf ØØØØXXXX.

Durch den anschließenden Vergleich (CMP #ØF) und den bedingten Sprung (BEQ) stellt der Prozessor fest, ob in der momentanen Matrix-Reihe eine Taste gedrückt ist.

Ist schließlich eine Reihe gefunden, in der eine Taste gedrückt ist, rettet der Computer die Reihen-Nummer in TEMPX und das Bitmuster der Spalte in KEY.

Jetzt ist der Prozessor beim Label KEYB angekommen und das X-Register arbeitet als Spaltenzähler (zuvor war das X-Register Reihenzähler). Den Inhalt von ROW (= Spalteninformation) schiebt der Prozessor solange nach rechts, bis die C-Flag = Ø ist. Nach vier Schiebeoperationen muß die C-Flag Null sein, anderenfalls ist das Programm falschgelaufen und der Computer versucht aufs neue die Berechnung des Tastenwertes (Verzweigung zum Label KEYVAL).

Ist aber die C-Flag zurückgesetzt, fragt das Programm nach der Matrix-Reihe, in welcher die Taste gedrückt war. Diese Information steht in der Speicherzelle TEMPX. Durch einfachen Vergleich (CMP) läßt sich jetzt feststellen, in welcher Reihe sich die gedrückte Taste befindet. Als Beispiel nehmen wir ROW2 (ROWC). Die Wertigkeit der Tasten steigt von Reihe zu Reihe um vier. Die Tasten in ROW2 haben gegenüber den Tasten

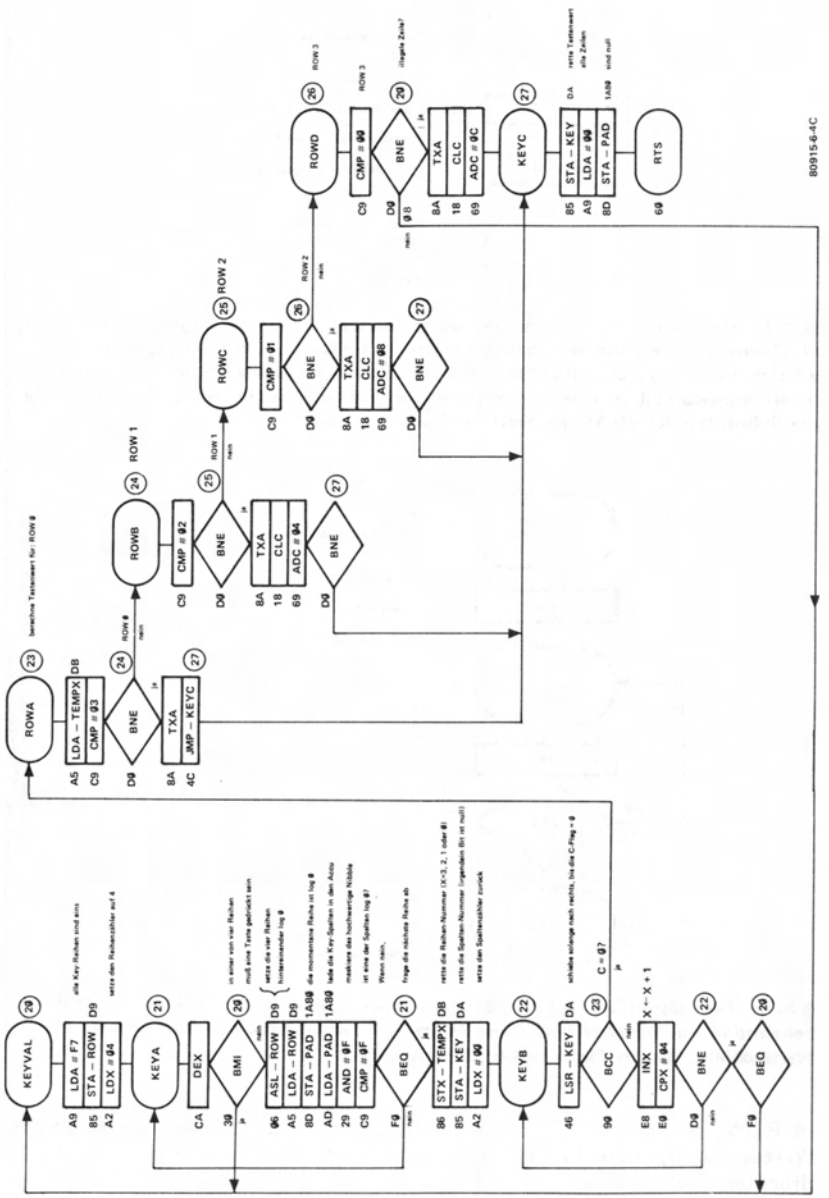


Bild 4c. Die Subroutine KEYVAL: Sie berechnet den Tastenwert einer gedrückten Taste der Keyboardmatrix von Bild 4a. Der berechnete Tastenwert ist nach dem Verlassen der Subroutine in der Speicherzelle KEY abgelegt.

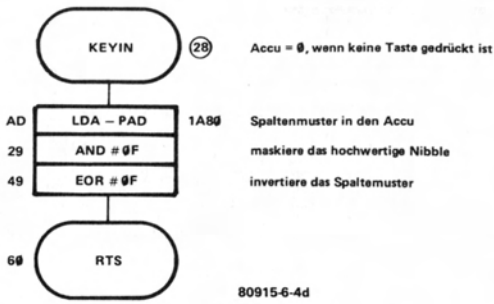


Bild 4d: Die Subroutine KEYIN: Mit dieser Subroutine läßt sich allgemein feststellen, ob überhaupt eine Taste der Keyboardmatrix gedrückt ist. Kehrt der Prozessor aus KEYIN mit einem Accuinhalt ØØ zurück, ist keine Taste gedrückt. Bei einem Accuinhalt ungleich null, ist eine Klaviaturtaste gedrückt und der Prozessor kann mit Hilfe der Subroutine KEYVAL den Wert der Taste berechnen.

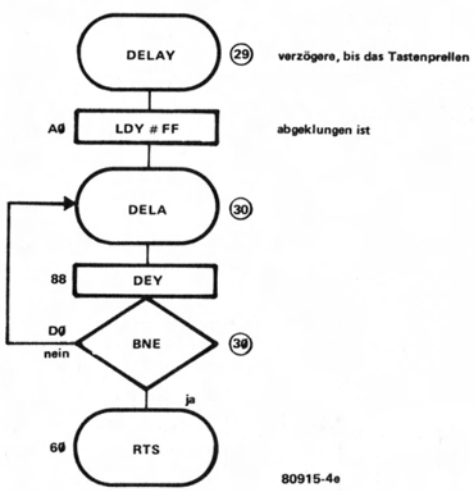


Bild 4e: Die Subroutine DELAY: Bei dem Drücken einer Klaviatur-Taste entsteht bekanntlich ein Tastenprellen. Diese Subroutine zwingt den Prozessor in eine Warteschleife, bis das Tastenprellen abgeklungen ist.

in ROWØ einen um acht höheren Wert. Durch einfaches Addieren dieses Wertes zur Spalten-Nummer erhält man den tatsächlichen Wert der gedrückten Taste.
Ist der Tastenwert berechnet, speichert der Computer ihn beim Label KEYC in der Speicherzelle KEY ab. Der Tastenwert ist somit gerettet.
Anschließend werden alle Zeilen der Keyboard-Matrix auf log. Ø gesetzt. Das ist wichtig für die Subroutine KEYIN (Bild 4d). Mit diesem Unterprogramm läßt sich schnell feststellen, ob irgendeine beliebige Klaviatur-

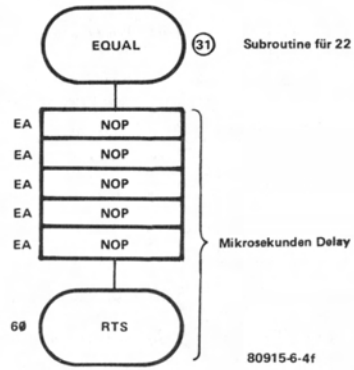


Bild 4f. Die Subroutine EQUAL: Diese Subroutine verzögert inklusive der JSR-Instruktion 22 μs und wird beim PLAY-Programm benötigt.

Taste gedrückt ist: Ist in irgendeiner Matrix-Reihe eine Taste gedrückt, muß eine der Spalten (PA3 . . . PAØ) log. Ø sein. Da KEYIN das gelesene Spaltenmuster invertiert (EOR # ØF), läßt sich feststellen:
Accu = ØØ, wenn keine Taste gedrückt ist,
Accu $\neq$ ØØ, wenn eine Taste gedrückt ist.
Die folgende Subroutine in Bild 4e heißt DELAY. Sie wird benötigt, um eine gedrückte Taste zu entprellen. Das erforderliche Delay erzeugt die Verzögerungsschleife DELA.
Bemerkung: In Kapitel 7 dieses Buches besprechen wir das Keyboard und Display auf der Basisplatine des Junior-Computers. Diese Keyboard-Routine hat viel Ähnlichkeit mit der Subroutine KEYVAL. Beide Programme weisen jedoch, was die Programmierung der Ports anbelangt, einen großen Unterschied auf:
Bei KEYVAL sind die Port-Leitungen miteinander "verkämmt". Das heißt, ein Teil der Port A-Leitungen arbeitet als Eingang und ein anderer als Ausgang (Matrix-Struktur).
Bei der Besprechung der Monitor-Routinen SCANDS, SCAND und GETKEY im folgenden Kapitel sind zwei Ports miteinander "verkämmt": PORT A und PORT B. Dabei ist bei der Keyboard-Scannung PORT A als Eingang und PORT B als Ausgang programmiert.
Bild 4g zeigt das Flußdiagramm von PLAY. Dieses Programm ordnet jeder gedrückten Klaviatur-Taste eine bestimmte Frequenz zu und macht die gespielte Melodie im Lautsprecher hörbar. Am Anfang dieses Programms wird der PIA definiert:
– PA7 . . . PA4 ist Ausgang
– PA3 . . . PAØ ist Eingang
– PBØ ist Ausgang.
Dann kommen wir beim Label PA an. Dort wartet der Junior-Computer auf eine gedrückte Klaviatur-Taste. Ob eine Taste gedrückt ist oder nicht, kann der Computer durch die Subroutine KEYIN (Bild 4d) erkennen.
Erkennt der Computer eine gedrückte Taste, dann gelangt er zur Subroutine DELAY (Bild 4e). Dieses Unterprogramm erledigt die Entprellung der

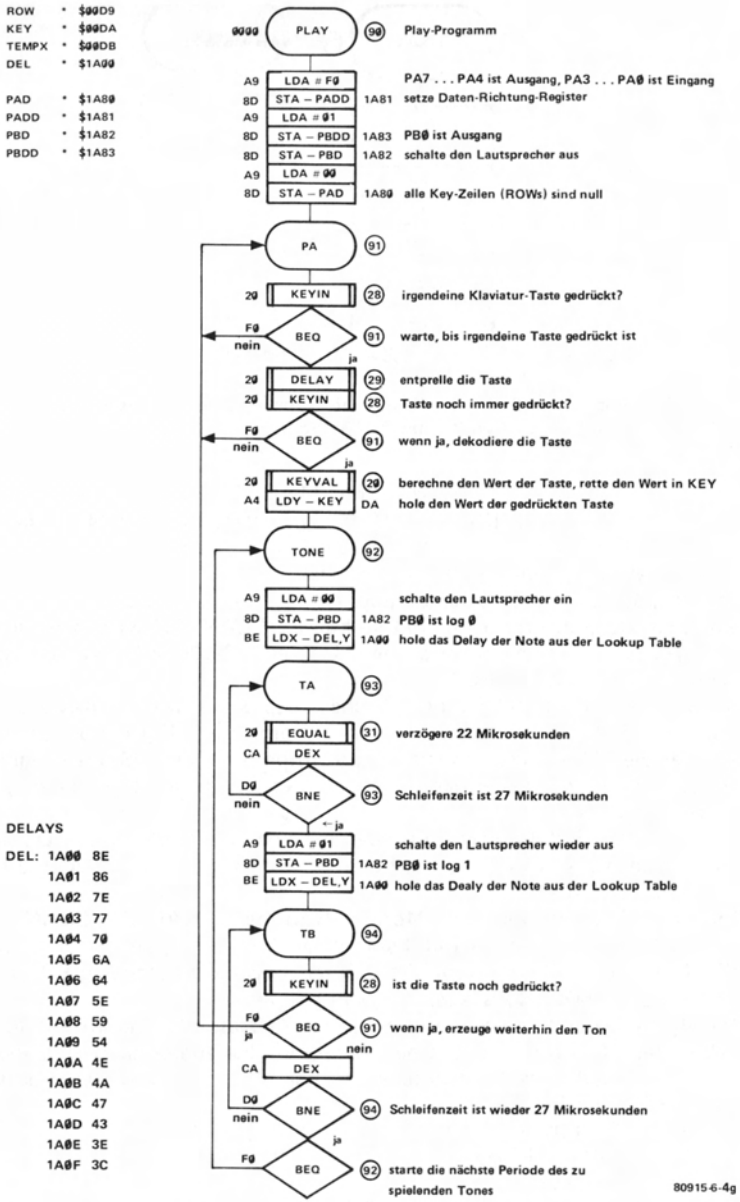


Bild 4g. Das Flußdiagramm des PLAY-Programms. Dieses Programm setzt den Wert einer gedrückten Klaviaturtaste in eine hörbare Frequenz um. Bei der Tastenwert-Frequenzumsetzung bedient sich der Prozessor der Lookup Table DEL. In dieser Tabelle stehen die halben Periodendauern des zu erzeugenden Tones.

Klaviatur-Tastatur. Nachdem das Tastenprellen abgeklungen ist, durchläuft der Computer nochmals die Subroutine KEYIN und prüft, ob die Taste immer noch gedrückt ist. Erst dann berechnet er in der Subroutine KEYVAL (Bild 4c) den Wert der gedrückten Taste. Dieser Tastenwert muß jetzt noch in eine Frequenz umgewandelt werden. Das geschieht ab dem Label TONE.
Bevor wir die Ton-Erzeugung im Detail besprechen, betrachten wir die TONE-Routine global:

- Label TONE: Der Lautsprecher wird für eine bestimmte Zeit eingeschaltet. Zeitbestimmend ist die Programmschleife TA . . . BNE . . . TA. Die Zeitdauer der Programmschleife hängt vom Inhalt des X-Registers ab. Bevor der Computer in diese Programmschleife kommt, lädt er das X-Register mit einer Zahl (Delay), die der Frequenz der gedrückten Taste entspricht. Diese Frequenz ist in der Lookup Table DEL abgelegt. Den Aufbau der Lookup Table besprechen wir später.
- Label TB: der Lautsprecher wird für eine bestimmte Zeit ausgeschaltet. Durch das wechselweise Ein- und Ausschalten entsteht an PBØ eine Rechteckspannung. Zeitbestimmend ist die Programmschleife TB . . . . BNE . . . TB. Auch hier hängt die Zeitdauer der Programmschleife vom Inhalt des X-Registers ab. Wie aus dem Flußdiagramm ersichtlich ist, springt der Mikroprozessor abermals in die Subroutine KEYIN, um zu prüfen, ob die Klaviatur-Taste noch immer gedrückt ist. Solange die Klaviatur-Taste gedrückt ist, entsteht an PBØ eine Rechteckspannung. Erst wenn die Taste in Ruhestellung ist, verzweigt das Programm zum Label PA und wartet auf einen neuen Tastendruck.

Lookup Table und Delays

Bevor sich die Delays in der Lookup Table berechnen lassen, müssen wir uns die Programmschleifen TA . . . BNE . . . TA und TB . . . BNE . . . TB ansehen. Am Beginn von TB springt die CPU in die Subroutine KEYIN, führt bei gedrückter Klaviatur-Taste den folgenden BEQ-Befehl aus (ohne zu verzweigen), dekrementiert das X-Register und verzweigt für eine bestimmte Zeit zum Label TB. Alle Instruktionen benötigen eine bestimmte Zeit, bis sie ausgeführt sind. Der folgende Zeitplan gibt darüber Auskunft:

JSR-KEYIN	= 6 μs	
LDA-PAD	= 4 μs	
AND # ØF	= 2 μs	
EOR # ØF	= 2 μs	
RTS	= 6 μs	
BEQ	= 2 μs	kein Branch, solange die Taste gedrückt ist
DEX	= 2 μs	
BNE-TB	= 3 μs	Branch wird ausgeführt
zusammen:	= 27 μs	

Die Schleifenzeit dauert also 27 μs mal Inhalt des X-Registers. Bild 5 zeigt diesen Zusammenhang in einer Grafik. Auch die Schleife TA . . . BNE . . . TA muß 27 μs betragen. Da in dieser Programmschleife nicht zur Subroutine KEYIN gesprungen wird, ist eine Korrektur der Schleifenzeit nötig.

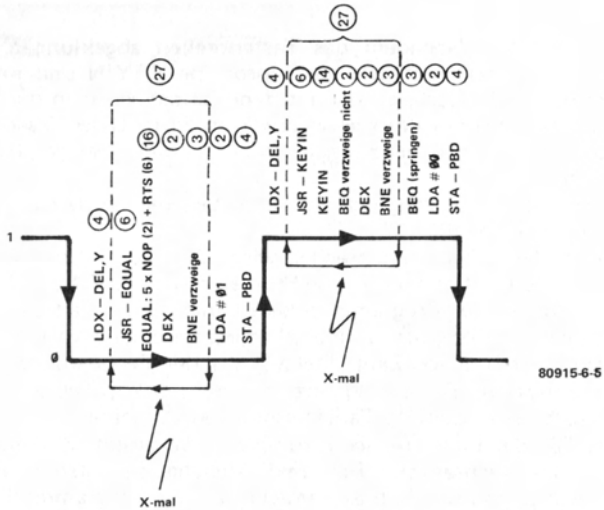


Bild 5. Mit diesem Programm erzeugt der Junior-Computer an der Port-Leitung PBØ eine Rechteckspannung, die über den angeschlossenen Lautsprecher hörbar ist.

Deshalb ist die Subroutine EQUAL vorgesehen; sie gleicht die fehlenden 22 μs aus.
Somit können wir die Lookup Table berechnen. In Bild 4a sind die Tasten mit einigen Zahlen versehen die einer Erklärung bedürfen. Nehmen wir zum Beispiel den Ton "a", der eine Frequenz von 440 Hz hat. Zwischen Zeit und Frequenz besteht der Zusammenhang:

$$T = 1/f.$$

Eine Frequenz von 440 Hz hat also eine Periodendauer von 1/440 Hz = 2273 μs. Bei einer Piano-Klaviatur steigt oder fällt die Frequenz von Taste zu Taste um den Faktor $\sqrt[12]{2}$ = 1,Ø59463. Somit können wir alle Tasten Frequenzen bzw. Periodendauern zuordnen, wie das Bild 4a zeigt. Ferner ist noch die Zeit der Programmschleifen TA . . . BNE . . . TA und TB . . . BNE . . . TB bekannt: 27 μs. Somit lassen sich für jede Taste die exakten Delays angeben. Nehmen wir wieder den Ton "a". Seine Periodendauer ist 2273 μs. Das Delay in der Lookup Table muß deshalb sein:
Delay = Periodendauer : 27 = 2273 : 27 = 84_{10} = $54. Für jede Taste läßt sich somit das Delay bestimmten und wir erhalten die Lookup Table LEN. Da der Mikroprozessor für jede gespielte Tonperiode zweimal das Delay aus der Lookup Table holt, sind die gespielten Töne eine Oktave tiefer als die berechneten!
Nun müssen wir noch das Programm PLAY (Bild 4g) mit seinen Unterprogrammen KEYVAL, KEYIN, DELAY und EQUAL (Bild 4c,d,e,f,) in den Junior-Computer eingeben. Dazu verwenden wir wieder den Editor und Assembler. Das ausführliche Tastenprogramm zeigt Bild 6. Nachdem das Programm assembliert und die Lookup Table in Seite 1 A des Junior-Computers eingetippt ist, kann man die Klaviatur und den Verstärker für den Lautsprecher mit dem PIA verbinden.
Und nun viel Spaß beim Klavierspielen!

Bild 6. Das Tastenprogramm von PLAY (Bild 4g). Das PLAY-Programm ist in Page Ø abgelegt. Nachdem alle Labels und Instruktionen eingetippt sind, wird der Assembler mit der ST-Taste gestartet. Nach erfolgter Assemblierung kann auf der Klaviatur eine Melodie gespielt werden. Die Startadresse von PLAY ist ØØØØ.

Tasten:		Display:	Bemerkung:
RST		XXXX XX	
AD	Ø Ø E 2	ØØE2 XX	
DA	Ø Ø	ØØE2 ØØ	BEGAD = ØØØØ
+	Ø Ø	ØØE3 ØØ	
+	E Ø	ØØE4 EØ	ENDAD = ØØEØ
+	Ø Ø	ØØE5 ØØ	
AD	1 A 7 A	1A7A XX	NMI-Vektor = 1F51
DA	5 1	1A7A 51	(Startadresse Assembler)
+	1 F	1A7B 1F	
AD	1 C B 5	1CB5 2Ø	
GO		77	Startadresse Editor
INSERT	F F 9 Ø Ø Ø	FF 9Ø ØØ	Label 9Ø: PLAY
INPUT	A 9 F Ø	A9 FØ	LDA #FØ
INPUT	8 D 8 1 1 A	8D 81 1A	STA-PADD
INPUT	A 9 Ø 1	A9 Ø1	LDA #Ø1
INPUT	8 D 8 3 1 A	8D 83 1A	STA-PBDD
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	A 9 Ø Ø	A9 ØØ	LDA #ØØ
INPUT	8 D 8 Ø 1 A	8D 8Ø 1A	STA-PAD
INPUT	F F 9 1 Ø Ø	FF 91 ØØ	Label 91: PA
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (Label 28)
INPUT	F Ø 9 1	FØ 91	BEQ zu PA (Label 91)
INPUT	2 Ø 2 9 Ø Ø	2Ø 29 ØØ	JSR-DELAY (Label 29)
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (Label 28)
INPUT	F Ø 9 1	FØ 91	BEQ zu PA (Label 91)
INPUT	2 Ø 2 Ø Ø Ø	2Ø 2Ø ØØ	JSR-KEYVAL (Label 2Ø)
INPUT	A 4 D A	A4 DA	LDYZ-KEY (ØØDA)
INPUT	F F 9 2 Ø Ø	FF 92 ØØ	Label 92: TONE
INPUT	A 9 Ø Ø	A9 ØØ	LDA #ØØ
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E Ø Ø 1 A	BE ØØ 1A	LDX-DEL,Y (DEL = 1AØØ)
INPUT	F F 9 3 Ø Ø	FF 93 ØØ	Label 93: TA
INPUT	2 Ø 3 1 Ø Ø	2Ø 31 ØØ	JSR-EQUAL (Label 31)
INPUT	C A	CA	DEX
INPUT	D Ø 9 3	DØ 93	BNE zu TA (Label 93)
INPUT	A 9 Ø 1	A9 Ø1	LDA #Ø1
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E Ø Ø 1 A	BE ØØ 1A	LDX-DEL,Y
INPUT	F F 9 4 Ø Ø	FF 94 ØØ	Label 94: TB
INPUT	2 Ø 2 8 Ø Ø	2Ø 28 ØØ	JSR-KEYIN (Label 28)
INPUT	F Ø 9 1	FØ 91	BEQ zu PA (Label 91)
INPUT	C A	CA	DEX
INPUT	D Ø 9 4	DØ 94	BNE zu TB (Label 94)
INPUT	F Ø 9 2	FØ 92	BEQ zu TONE (Label 92)

Tasten:		Display:	Bemerkung:
INPUT	F F 2 Ø Ø Ø	FF 2Ø ØØ	Label 2Ø: KEYVAL
INPUT	A 9 F 7	A9 F7	LDA #F7
INPUT	8 5 D 9	85 D9	STAZ-ROW (ØØD9)
INPUT	A 2 Ø 4	A2 Ø4	LDX #Ø4
INPUT	F F 2 1 Ø Ø	FF 21 ØØ	Label 21: KEYA
INPUT	C A	CA	DEX
INPUT	3 Ø 2 Ø	3Ø 2Ø	BMI zu KEYVAL (Label 2Ø)
INPUT	Ø 6 D 9	Ø6 D9	ASLZ-ROW (ØØD9)
INPUT	A 5 D 9	A5 D9	LDAZ-ROW (ØØD9)
INPUT	8 D 8 Ø 1 A	8D 8Ø 1A	STA-PAD
INPUT	A D 8 Ø 1 A	AD 8Ø 1A	LDA-PAD
INPUT	2 9 Ø F	29 ØF	AND #ØF
INPUT	C 9 Ø F	C9 ØF	CMP #ØF
INPUT	F Ø 2 1	FØ 21	BEQ zu KEYA (Label 21)
INPUT	8 6 D B	86 DB	STXZ-TEMPX (ØØDB)
INPUT	8 5 D A	85 DA	STAZ-KEY (ØØDA)
INPUT	A 2 Ø Ø	A2 ØØ	LDX #ØØ
INPUT	F F 2 2 Ø Ø	FF 22 ØØ	Label 22: KEYB
INPUT	4 6 D A	46 DA	LSRZ-KEY (ØØDA)
INPUT	9 Ø 2 3	9Ø 23	BCC zu ROWA (Label 23)
INPUT	E 8	E8	INX
INPUT	E Ø Ø 4	EØ Ø4	CPX #Ø4
INPUT	D Ø 2 2	DØ 22	BNE zu KEYB (Label 22)
INPUT	F Ø 2 Ø	FØ 2Ø	BEQ zu KEYVAL (Label 2Ø)
INPUT	F F 2 3 Ø Ø	FF 23 ØØ	Label 23: ROWA
INPUT	A 5 D B	A5 DB	LDAZ-TEMPX (ØØDB)
INPUT	C 9 Ø 3	C9 Ø3	CMP #Ø3
INPUT	D Ø 2 4	DØ 24	BNE zu ROWB (Label 24)
INPUT	8 A	8A	TXA
INPUT	4 C 2 7 Ø Ø	4C 27 ØØ	JMP-KEYC (Label 27)
INPUT	F F 2 4 Ø Ø	FF 24 ØØ	Label 24: ROWB
INPUT	C 9 Ø 2	C9 Ø2	CMP #Ø2
INPUT	D Ø 2 5	DØ 25	BNE zu ROWC (Label 25)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø 4	69 Ø4	ADC #Ø4
INPUT	D Ø 2 7	DØ 27	BNE zu KEYC (Label 27)
INPUT	F F 2 5 Ø Ø	FF 25 ØØ	Label 25: ROWC
INPUT	C 9 Ø 1	C9 Ø1	CMP #Ø1
INPUT	D Ø 2 6	DØ 26	BNE zu ROWD (Label 26)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø 8	69 Ø8	ADC #Ø8
INPUT	D Ø 2 7	DØ 27	BNE zu KEYC (Label 27)
INPUT	F F 2 6 Ø Ø	FF 26 ØØ	Label 26: ROWD
INPUT	C 9 Ø Ø	C9 ØØ	CMP #ØØ
INPUT	D Ø 2 Ø	DØ 2Ø	BNE zu KEYVAL (Label 2Ø)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC

Tasten:		Display:	Bemerkung:
INPUT	6 9 Ø C	69 ØC	ADC #ØC
INPUT	F F 2 7 Ø Ø	FF 27 ØØ	Label 27: KEYC
INPUT	8 5 D A	85 DA	STAZ-KEY (ØØDA)
INPUT	A 9 Ø Ø	A9 ØØ	LDA #ØØ
INPUT	8 D 8 Ø 1 A	8D 8Ø 1A	STA-PAD
INPUT	6 Ø	6Ø	RTS
INPUT	F F 2 8 Ø Ø	FF 28 ØØ	Label 28: KEYIN
INPUT	A D 8 Ø 1 A	AD 8Ø 1A	LDA-PAD
INPUT	2 9 Ø F	29 ØF	AND #ØF
INPUT	4 9 Ø F	49 ØF	EOR #ØF
INPUT	6 Ø	6Ø	RTS
INPUT	F F 2 9 Ø Ø	FF 29 ØØ	Label 29: DELAY
INPUT	A Ø F F	AØ FF	LDY #FF
INPUT	F F 3 Ø Ø Ø	FF 3Ø ØØ	Label 3Ø: DELA
INPUT	8 8	88	DEY
INPUT	D Ø 3 Ø	DØ 3Ø	BNE zu DELA (Label 3Ø)
INPUT	6 Ø	6Ø	RTS

Tasten:		Display:	Bemerkung:
INPUT	F F 3 1 Ø Ø	FF 31 ØØ	Label 31: EQUAL
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	6 Ø	6Ø	RTS

Testen des eingetippten Programms:

Tasten:		Display:
SEARCH	F F 9 Ø	FF 9Ø ØØ
SKIP		A9 FØ
SKIP		8D 81 1A
SKIP		A9 Ø1
SKIP		8D 83 1A
SKIP		8D 82 1A
SKIP		A9 ØØ
SKIP		8D 8Ø 1A
SKIP		FF 91 ØØ
SKIP		2Ø 28 ØØ
SKIP		FØ 91
SKIP		2Ø 29 ØØ

usw

Tasten:	Display:
ST	XXXXXX

Eingabe der Frequenz-Lookup-Tabel:

Tasten:		Display:
AD	1 A Ø Ø	1AØØ XX
DA	8 E	1AØØ 8E
+	8 6	1AØ1 86
+	7 E	1AØ2 7E
+	7 7	1AØ3 77
+	7 Ø	1AØ4 7Ø
+	6 A	1AØ5 6A
+	6 4	1AØ6 64
+	5 E	1AØ7 5E
+	5 9	1AØ8 59
+	5 4	1AØ9 54
+	4 E	1AØA 4E
+	4 A	1AØB 4A
+	4 7	1AØC 47
+	4 3	1AØD 43
+	3 E	1AØE 3E
+	3 C	1AØF 3C

Programm-Start:

Tasten:		Display:
AD	Ø Ø Ø Ø	ØØØØA9
GO		

Auf der Klaviatur läßt sich jetzt eine Melodie spielen!

Der Intervall-Timer

Der Junior-Computer hat auch einen Intervall-Timer, der ebenfalls in dem IC 6532 untergebracht ist. Dieser Timer läßt sich programmieren und läuft unabhängig vom Mikroprozessor. Das heißt, während der Timer arbeitet, steht die CPU dem Programmierer zur freien Verfügung. Diese Voraussetzung muß erfüllt sein, wenn man mit einem Mikrocomputer effektiv, das heißt schnell arbeiten will.
Das Geheimnis, warum der Junior-Computer nur eine handvoll integrierte Schaltungen benötigt und so billig ist, kann nun gelüftet werden: Der Intervall-Timer und der periphere Interface Adapter sind auf nur einem Chip vereint. Beide sind wichtige Bestandteile des Junior-Computers und steigern dessen Leistungsfähigkeit um ein Vielfaches. Doch bevor wir mit dem Intervall-Timer arbeiten können, muß dieser genau erklärt werden. Deshalb beschreiben wir zuerst das Programmier-Modell des Timers und besprechen anschließend einige Übungsprogramme, bei denen der Intervall-Timer Verwendung findet.

Das Blockschaltbild des Intervall-Timers

Bild 7 zeigt das Blockschaltbild des Intervall-Timers. Aus ihm geht hervor, daß der Timer aus drei Blöcken besteht:

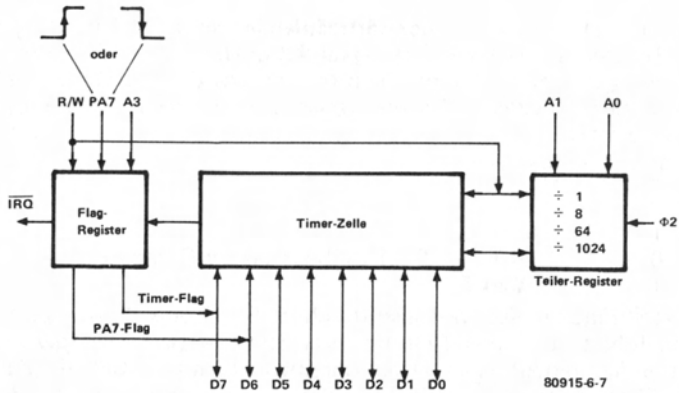


Bild 7. Das Blockschaltbild des Intervall-Timers mit dem Flag-Register. Das Teiler-Register und die Timer-Zelle bilden zusammen den Intervall-Timer. Dabei läßt sich mit dem Teiler-Register ein bestimmter Teilungsfaktor einstellen. Vier Teilungsfaktoren stehen zur Verfügung: ÷1, ÷8, ÷64 und ÷1Ø24. Welcher dieser Teilungsfaktoren aktiv ist, bestimmt das Bitmuster auf den Adreßleitungen AØ und A1. Die Clock-Frequenz Φ2 steuert über das Teiler-Register die Timer-Zelle an. Dabei teilt das Teiler-Register den Phasentakt Φ2 um einen der vier wählbaren Teilungsfaktoren. Der Intervall-Timer ist mit einem setzbaren Rückwärtszähler vergleichbar. Je nach gewähltem Teilungsfaktor wird der Inhalt der Timer-Zelle alle 1 µs, 8 µs, 64 µs oder 1Ø24 µs um eins dekrementiert. Der Start des Intervall-Timers erfolgt durch eine Schreiboperation in die Timer-Zelle. Dann zählt der Intervall-Timer solange rückwärts, bis der Inhalt der Timer-Zelle kleiner als Null geworden ist (Inhalt = FF). Tritt dieser Zustand ein, geht im Flag-Register automatisch die Timer-Flag auf log. 1 (sie ist dann gesetzt). Der Programmierer kann dem Timer erlauben oder verbieten einen IRQ auszulösen. Darf der Timer einen Timer-IRQ auslösen, wird beim Setzen der Timer-Flag die IRQ-Leitung log. Ø.
Im Flag-Register befindet sich noch eine weitere Flag: die PA7-Flag. Sie arbeitet mit dem Flankendetektor zusammen. Die Port-Leitung PA7 ist der Eingang des Flankendetektors. Der Flankendetektor läßt sich so programmieren, daß entweder eine positive oder negative Pulsflanke die PA7-Flag setzt. Der Programmierer kann den Flankendetektor erlauben oder verbieten, einen IRQ auszulösen. Darf der Flankendetektor einen IRQ auslösen, dann wird die IRQ-Leitung beim Setzen der PA7-Flag log. Ø.

- **Timerzelle**: Die Timer-Zelle ist acht Bit breit und mit dem Datenbus verbunden. Die Timer-Zelle ist mit einer herkömmlichen RAM-Zelle vergleichbar: Daten können in sie geschrieben oder aus ihr gelesen werden.
- **Teiler-Register**: Das Teiler-Register ist mit den Adreß-Leitungen AØ und A1 sowie mit dem Taktsignal Φ2 verbunden. Die Timer-Zelle und das Teiler-Register bilden **zusammen** den Intervall-Timer.

Der Intervall-Timer ist mit einem setzbaren TTL- oder CMOS-Rückwärtszähler vergleichbar. Diese bekannten Bausteine haben die Eigenschaft, daß sich das Zähler-Register auf eine bestimmte Binärzahl setzen läßt. Jeder Clockpuls dekrementiert dann den Zähler solange um eins, bis der Zählerinhalt null geworden ist.
Genauso arbeitet der Intervall-Timer des Junior-Computers. Das Taktsignal

Φ2 ist der Clockpuls des **rückwärtszählenden** Intervall-Timers. Mit dem Teiler-Register läßt sich das Taktsignal Φ2 um bestimmte Faktoren teilen. Den Teilungsfaktor bestimmt das Bitmuster auf den Adreßlinien AØ und A1. Mit diesen beiden Adreßbits lassen sich vier verschiedene Teilungsfaktoren einstellen:

A1	AØ	Teilungsfaktor	
Ø	Ø	÷ 1 T	
Ø	1	÷ 8 T	
1	Ø	÷ 64 T	T = 1 μs (beim Junior-Computer)
1	1	÷ 1Ø24 T	

Die Clockfrequenz Φ2 wird also durch 1, 8, 64 oder 1024 geteilt bevor sie den Inhalt der Timer-Zelle um eins dekrementiert. Das folgende Beispiel soll das verdeutlichen. Dazu schreiben wir mit der CPU die Zahl $1E in die Timer-Zelle und wählen mit den Adreßlinien AØ und A1 den Teilungsfaktor ÷ 64 an. Es läßt sich dann folgendes sagen:
$1E = $3Ø$_{1Ø}$ und $3Ø_{1Ø}$ mal $64_{1Ø}$ ist $192Ø_{1Ø}$.
oder mit anderen Worten: Bei einem Teilungsfaktor von 64 und einem Timer-Offset von $1E ist der Inhalt der Timerzelle nach 192Ø μs null geworden. Bevor wir den Intervall-Timer genauer erklären können, muß noch ein weiteres wichtiges Register im IC 6532 besprochen werden, das Flag-Register.

– **Flag-Register**: Das Flag-Register des 6532 hat zwei Flags:
  1. Die Timer-Flag
  2. Die PA7-Flag (wird an anderer Stelle besprochen!)

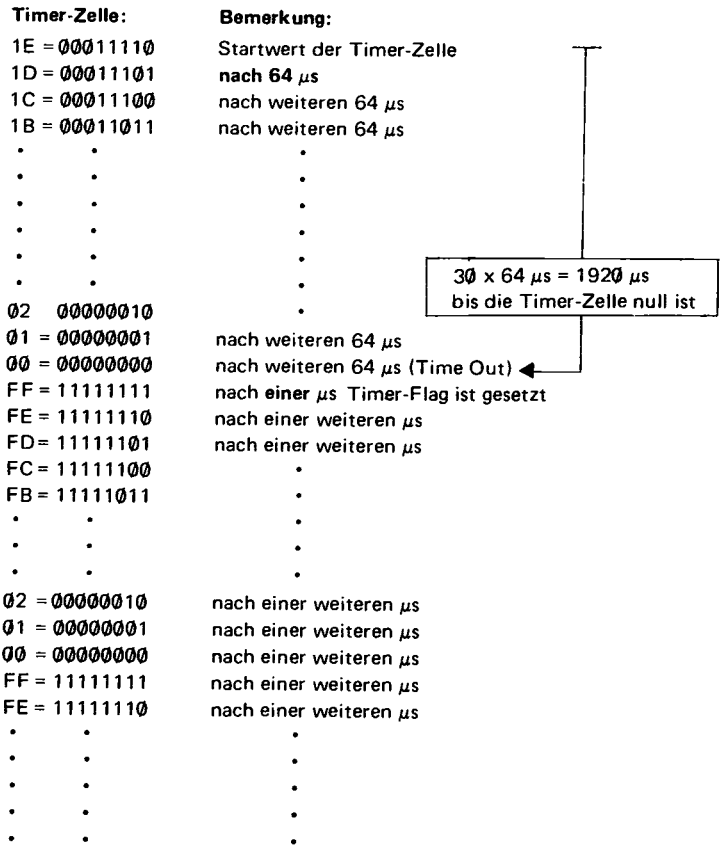
Das Flag-Register ist ebenfalls 8 Bit breit, jedoch nur zwei Bits werden als Flags verwendet: b7 und b6. Die anderen Bits, also b5 . . . bØ, sind immer null, wenn der Prozessor dieses Register liest. Die Timer-Flag setzt der Intervall-Timer, die PA7-Flag eine positive oder negative Pulsflanke an der Port-Leitung PA7.

Der Aufbau des Flag-Registers:

b7	b6	b5	b4	b3	b2	b1	bØ	
X	X	Ø	Ø	Ø	Ø	Ø	Ø	Bits
↑	↑							Flag-Register
└	PA7-Flag							
└	Timer-Flag							

Wann setzt der Intervall-Timer die Timer-Flag?

Bit b7 im Flag-Register ist die Timer-Flag. Diese Flag wird gesetzt (b7 = 1), wenn der Timer abgelaufen ist. Auf das vorige Beispiel bezogen heißt abgelaufen:



Jetzt ist klar, wie das Flag-Register mit dem Intervall-Timer zusammenarbeitet. Nachdem die CPU eine Zahl – das Timer-Offset – in die Timer-Zelle geschrieben hat und bei dieser Schreibeoperation mit den Adreßleitungen A0 sowie A1 der Teilungsfaktor bestimmt ist, beginnt der Intervall-Timer rückwärts zu zählen. Da wir einen Teilungsfaktor von 64 und ein Timer-Offset von 1E = 30_{10} wählten, ist der Inhalt der Timer-Zelle nach 1920 µs null geworden. Der Moment, wo der Inhalt der Timer-Zelle null geworden ist, heißt **Time Out**.

Was man noch wissen muß:

1. Die Timer-Flag im Flag-Register ist nach einer Leseoperation der Timer-Zelle zurückgesetzt, also log. 0 (lesen der Zellen RDTEN oder RDTDIS).
2. Die Timer-Flag im Flag-Register ist nach einer Schreibeoperation in eine Timer-Zelle (Schreiben in eine der CNTA . . . CNTH) zurückgesetzt (log. 0). Nachdem man den Timer gestartet hat (= Schreiben des Timer-Offsets in die Timer-Zelle), ist also die Timer-Flag immer zurückgesetzt.

3. Die Timer-Flag im Flag-Register wird 1 µs **nach dem Time Out** gesetzt. Für unser Beispiel heißt das:
   Timer-Offset = 1E = $3Ø_{1Ø}$
   Teilungsfaktor = 64
   Time Out nach 3Ø x 64 µs = 192Ø µs
   Setzen der Timer-Flag nach 192Ø µs + 1 µs = 1921 µs
4. Nach dem Time Out stellt sich automatisch ein Teilungsfaktor von eins ein. Das ist unabhängig vom zuvor gewählten Teilungsfaktor. War der Teilungsfaktor beispielsweise 64, dann gilt:
   - Vor dem Time Out wird der Inhalt der Timer-Zelle alle 64 µs um eins dekrementiert.
   - Nach dem Time Out wird der Inhalt der Timer-Zelle nach jeder µs um eins dekrementiert (1 µs entspricht 1 MHz = Clock-Frequenz des Junior-Computers).

Das Flag-Register steuert auch noch die IRQ-Leitung, die mit dem IRQ-Eingang der CPU verbunden ist. Ist die I-Flag im Statusregister der CPU zurückgesetzt, kann das IC 6532 einen IRQ auslösen. Die IRQ-Leitung ist dann log. Ø.
Die Steuerung der IRQ-Leitung übernehmen die beiden Flags im Flag-Register des ICs 6532. Ist die Timer-Flag des Intervall-Timers **oder** die PA7-Flag gesetzt, dann **kann** die IRQ-Leitung log. Ø werden und einen Interrupt Request auslösen. Der Programmierer kann jedoch den IRQ-Befehl mit dem Flag-Register im IC 6532 verhindern. Deshalb ist die Adreßlinie A3 mit dem Flag-Register verbunden:
- A3 = Ø : Das IC 6532 **kann keinen** Interrupt auslösen. Die IRQ-Leitung **kann nicht log. Ø** werden.
- A3 = 1 : Das IC 6532 **kann einen** Interrupt auslösen. Die IRQ-Leitung **kann log. Ø** werden.

Zusammenfassung:

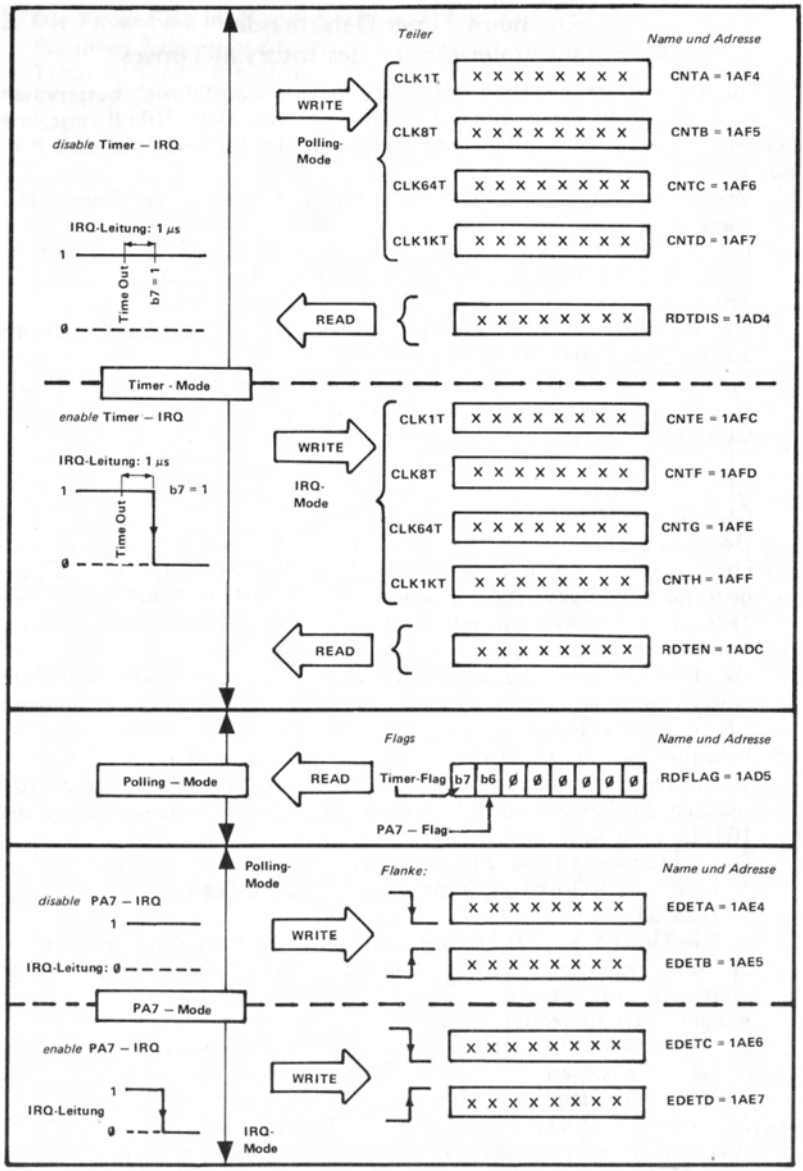
*Das Blockschaltbild des Intervall-Timers ist somit besprochen. Der Intervall-Timer besteht aus einem Teiler-Register und einer Timer-Zelle. Die Timer-Zelle ist mit dem Flag-Register verbunden. Im Flag-Register befinden sich zwei Flags:
1. Die Timer-Flag (b7)
2. Die PA7-Flag (b6).

*Abhängig von der Timer- und PA7-Flag steuert das Flag-Register die IRQ-Leitung. Normal ist diese Leitung log. 1. Ist die IRQ-Leitung aber log. Ø, will das IC 6532 einen Interrupt Request auslösen.
*Der Programmierer kann dem IC 6532 verbieten, einen IRQ auszulösen.
*Nach einer Lese- oder Schreibeoperation aus der/in die Timer-Zelle ist die Timer-Flag im Flag-Register **immer** zurückgesetzt (Timer-Flag = Ø).
*Wenn der Programmierer es wünscht, kann das IC 6532 einen Timer-IRQ auslösen. Ist der Timer abgelaufen, wird die Timer-Flag gesetzt. Ein Setzen der Timer-Flag hat zur Folge, daß die IRQ-Leitung auf log. Ø geht, und die CPU in eine Interrupt-Routine verzweigt.

Die neun Timer-Gebote oder das Programmier-Modell des Intervall-Timers

Nachdem das Blockschaltbild des Intervall-Timers ausführlich besprochen ist, können wir das Programmier-Modell leicht verstehen. Bild 8 zeigt, wie sich der Intervall-Timer, die Timer-Zellen und das Flag-Register dem Programmierer präsentieren.

1. Der Intervall-Timer besteht aus insgesamt acht Timer-Zellen. Die Namen der Timer-Zellen sind:
   CNTA, CNTB, CNTC, CNTD, CNTE, CNTF, CNTG und CNTH.
   Diesen Namen sind gemäß Bild 8 die Adressen 1AF4 . . . 1AF7 und 1AFC . . . 1AFF zugeordnet.
2. Der Prozessor kann sowohl den Inhalt einer Timer-Zelle lesen, als auch in eine Timer-Zelle schreiben.
3. Jeder Timer-Zelle ist ein bestimmter Teilungsfaktor zugeordnet. Vier Teilungsfaktoren stehen zur Auswahl:
   CLK1T = 1
   CLK8T = 8
   CLK64T = 64
   CLK1KT = 1Ø24
   Der Timer-Zelle CNTA ist ein Teilungsfaktor von "1" zu geordnet. Der folgenden Timer-Zelle CNTB ist ein Teilungsfaktor von "8" zugeordnet usw. Bevor also die Clock-Frequenz Φ2 den Inhalt der Timerzelle um eins dekrementiert, wird sie um einen bestimmten Teilungsfaktor herabgesetzt.
4. Der Intervall-Timer ist in zwei Teile zu je vier Zellen aufgeteilt. Den einen Teil bilden die Timer-Zellen CNTA . . . CNTD und den anderen CNTE . . . CNTH.
5. Schreibt der Prozessor in eine der Timer-Zellen CNTA . . . CNTD, startet das den Intervall-Timer. Bei dieser Schreiboperation wird automatisch die Timer-Flag (b7 im Flag-Register) zurückgesetzt. Auch die IRQ-Leitung wird log. 1.
   Das Schreiben in Timer-Zellen CNTA . . . CNTD hat zur Folge:
   – Über die IRQ-Leitung kann das IC 6532 **keinen** Interrupt Request auslösen.
   – Die Timer-Flag (b7) im Flag-Register wird **immer** zurückgesetzt.
   Je nach Teilungsfaktor dauert es kürzer oder länger, bis der Timer abgelaufen ist. Nach dem Time Out (Inhalt der Timer-Zelle ist FF) ereignet sich folgendes:
   – Die IRQ-Leitung kann nicht log. Ø werden und somit **keinen** Interrupt auslösen.
   – Die Timer-Flag (b7 im Flag-Register) ist gesetzt.
6. Schreibt der Prozessor in eine der Timer-Zellen CNTE . . . CNTH, wird der Intervall-Timer gestartet. Gleichzeitig setzt diese Schreiboperation automatisch die Timer-Flag (b7 im Flag-Register) zurück. Auch die IRQ-Leitung wird log. 1 (siehe auch 5).
   Das Schreiben in die Timer-Zellen CNTE . . . CNTH hat zur Folge:
   – Über die IRQ-Leitung kann das IC 6532 einen Interrupt Request auslösen.
   – Die Timer-Flag (b7 im Flag-Register) wird **immer** zurückgesetzt.



80915-6-8

Je nach Teilungsfaktor ist die Zeitdauer bis zum Ablauf des Timers unterschiedlich. Nach dem Time Out (Inhalt der Timer-Zelle ist FF) ereignet sich folgendes:

Bild 8. Das Programmier-Modell des Intervall-Timers und des Flankendetektors. Der Intervall-Timer besteht aus den acht Timer-Zellen CNTA . . . CNTH. Bei der Schreiboperation des Timer-Offsets (ØØ . . . FF) in eine der acht Timerzellen wird der Intervall-Timer gestartet. Nach dem Start zählt der Intervall-Timer in Abhängigkeit des Teilungsfaktors bis zum Time Out rückwärts. Nach dem Time Out wird die Timer-Flag im Flag-Register RDFLAG gesetzt und der Intervall-Timer löst über die IRQ-Leitung einen Timer-IRQ aus, falls der Programmierer das wünscht. Wird der Intervall-Timer durch eine Schreiboperation in eine der Timerzellen CNTA . . . CNTD gestartet, kann er keinen Timer-IRQ auslösen. Die IRQ-Leitung bleibt log. 1, wenn die Timer-Flag gesetzt wird. Bei einer Schreiboperation in die Timerzellen CNTE . . . CNTH wird der Intervall-Timer ebenfalls gestartet und dieser löst nach einem Time Out einen Timer-IRQ aus. Die IRQ-Leitung geht beim Setzen der Timer-Flag auf log. Ø. Eine Schreiboperation in die Timer-Zellen CNTA . . . CNTH legt fest, ob der Intervall-Timer in Polling- oder IRQ-Mode arbeitet. Den Teilungsfaktor bestimmt die Schreiboperation beim Start des Intervall-Timers. Ein Lesen der Timer-Zelle RDTEN bzw. RDTIS vor dem Time Out verändert den Teilungsfaktor nicht, hat jedoch eine Änderung der Polling- oder IRQ-Mode zur Folge. Eine Leseoperation der Timer-Zellen RDTEN oder RDTDIS gibt über den momentanen Inhalt der Timer-Zelle Auskunft. Arbeitet der Intervall-Timer in IRQ-Mode, teilt er dem Mikroprozessor über die IRQ-Leitung mit, daß sich ein Time Out ereignete. Arbeitet aber der Intervall-Timer in Polling-Mode, dann erfährt die CPU durch Lesen des Flag-Registers RDFLAG, ob sich bereits ein Time Out ereignete.
Der Flankendetektor kann wie der Intervall-Timer in Polling- oder IRQ-Mode betrieben werden. Eine Schreiboperation in eines der vier Steuer-Register legt fest, ob der Flankendetektor auf eine positive oder negative Pulsflanke an der PA7-Leitung reagieren soll. Somit kann eine positive oder negative Pulsflanke die PA7-Flag setzen. Ist dem Flankendetektor erlaubt, einen IRQ auszulösen (Schreiboperation in die Steuer-Register EDETC für ↓ oder EDETD für ↑), wird die IRQ-Leitung beim Setzen der PA7-Flag log. Ø. Arbeitet der Flenkendetektor in IRQ-Mode, teilt er dem Mikroprozessor durch die IRQ-Leitung mit, daß an PA7 die gewünschte Pulsflanke eingetroffen ist.

- Die IRQ-Leitung wird log. Ø und löst somit einen Interrupt Request aus (falls die I-Flag im Prozessor Status Register zurückgesetzt ist: CLI).
- Die Timer-Flag (b7 im Flag-Register) wird gesetzt.

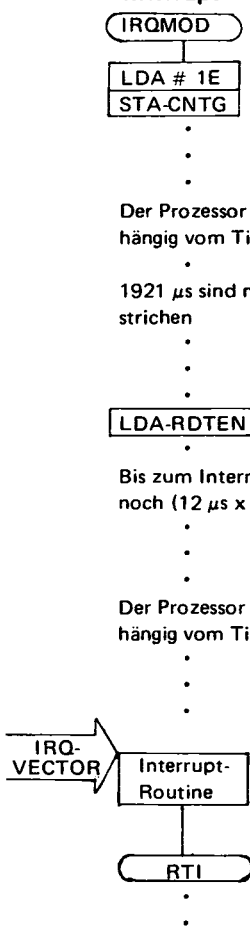
7. Der Prozessor kann auch den momentanen Inhalt der Timer-Zelle lesen. Eine Leseoperation der Timer-Zelle RDTDIS mit der Adresse 1AD4 gibt darüber Auskunft, wieviel Zeit bis zum Time Out noch verbleibt. Ein Lesen der Timer-Zelle RDTDIS verbietet dem Timer, einen Timer-IRQ auszulösen. Eine Leseoperation der Timer-Zelle RDTEN mit der Adresse 1ADC gibt darüber Auskunft, wieviel Zeit bis zum Time Out noch verbleibt. Ein Lesen der Timer-Zelle RDTEN erlaubt dem Timer, einen Timer-IRQ auszulösen.
   Durch eine Leseoperation der Timer-Zelle RDTIS und RDTEN läßt sich der Intervall-Timer von Polling- auf Interrupt-Mode oder umgekehrt umschalten. Die Leseoperation verändert dabei den Inhalt des Timers nicht, sondern gibt über seinen momentanen Zustand Auskunft. Findet die Leseoperation vor dem Time Out statt, dann bleibt der zuvor eingestellte Teilungsfaktor erhalten. Eine Leseoperation aus den Timer-Zellen RDTDIS und RDTEN nach dem Time Out gibt darüber Auskunft, wieviele Mikrosekunden seit dem Time Out ver-

strichen sind. Die folgenden Beispiele sollen diese Zusammenhänge erklären:

Timer-Zelle CNTG:

- Timer-Offset = 1E = 30_{10}
- Teilungsfaktor = 64
- CNTG kann einen IRQ auslösen
- Die Zeit vom Start des Intervall-Timers bis zu Auslösung des IRQ (und bis zum Setzen der Timer-Flag) ist: (30 µs x 64) + 1 µs = 1921 µs.

Das Starten des Intervall-Timers und das Lesen der Timer-Zelle in **Interrupt-Mode** veranschaulicht das folgende Flußdiagramm:



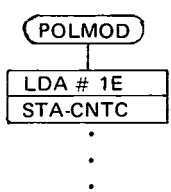
```
        .
        .
        .
        .
        .
        .
LDA-RDTEN   Lesen der Timer-Zelle nach dem Interrupt. Der Inhalt der
        .   Zelle sei A4.
        .   A4 = 10100100   momentaner Inhalt der Zelle
        .   Ā4 = 01011011   komplement = 5B
        .   Ā4 + 1 = 5B + 1 = 5C = 92₁₀ (Zweierkompl.)
Seit dem Interrupt sind 92 µs
verstrichen. Insgesamt sind seit
dem Start des Intervall-Timers
1921 µs = 92 µs = 2013 µs
verstrichen
        .
        .
        .
Der Timer dekrementiert
alle µs um eins bis er
vom Prozessor wieder ge-
startet wird
        .
        .
        .
```

8. **Polling-Mode:** Der Timer kann auch in **Polling Mode** betrieben werden. Das heißt, der Timer wird periodisch oder nichtperiodisch abgefragt, ob sich schon ein Time Out ereignet hat. Dabei verbietet der Programmierer dem Intervall-Timer, nach einem Time Out einen Interrupt Request auszulösen. Ein Schreiben in die Timer-Zellen CNTA . . . CNTD oder ein Lesen aus der Timer-Zelle RDTDIS verbietet dem Intervall-Timer einen Timer-IRQ auszulösen. Das heißt, ist der Timer abgelaufen, wird im Flag-Register die Timer-Flag gesetzt, die IRQ-Leitung bleibt jedoch weiterhin log. 1. Ein Interrupt Request kann somit nicht ausgelöst werden. **In Polling Mode ist das Kriterium für einen Time Out die Timer-Flag im Flag-Register und nicht die IRQ-Leitung.** Das folgende Beispiel zeigt den Programmablauf für Polling Mode. Dazu verwenden wir einen neuen Befehl, der aus Buch 1 noch nicht bekannt ist: BIT. Mit dem BIT-Befehl lassen sich Bits im Arbeitsspeicher, auch Bits im Flag-Register des ICs 6532 prüfen. Die allgemeine Funktionsweise des BIT-Befehls ist:
   - *A ∧ M Der Inhalt des Accus wird dabei nicht zerstört, das Ergebnis der AND-Funktion beeinflußt die Z-Flag.
   - *$M_7 \rightarrow N$ b7 der Speicherzelle wird zur N-Flag im Prozessor Statusregister transportiert (P-Register in der CPU)
   - *$M_6 \rightarrow V$ b6 der Speicherzelle wird zur V-Flag im Prozessor Statusregister transportiert
   - *Der BIT-Befehl hat die Adressierungsarten Zero Page und Absolute.

* Beim Junior-Computer hat die Instruktion BIT-RDFLAG folgenden Effekt:
  - b7 im Flag-Register = Timer-Flag = N-Flag (P-Register in der CPU)
  - b6 im Flag-Register = PA7-Flag = V-Flag (P-Register in der CPU)

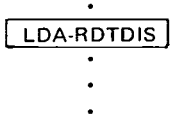
  Schreibt der Prozessor in eine der Timer-Zellen CNTA . . . CNTD, wird der Intervall-Timer gestartet. Bei dieser Schreiboperation wird automatisch die Timer-Flag (b7 im Flag-Register) zurückgesetzt.
* Das Schreiben in Timer-Zellen CNTA . . . CNTD hat zur Folge:
  - Über die IRQ-Leitung kann das IC 6532 keinen Interrupt Request auslösen.
  - Die Timer-Flag (b7 im Flag-Register) wird **immer** zurückgesetzt.
* Je nach Teilungsfaktor dauert es kürzer oder länger, bis der Timer abgelaufen ist. Nach dem Time Out (Inhalt der Timer-Zelle ist FF) ereignet sich folgendes:
  - Die IRQ-Leitung bleibt log. 1 und löst somit keinen Interrupt Request aus.
  - Die Timer-Flag (b7 im Flag-Register) wird gesetzt.
* Liest der Prozessor die Timer-Zellen RDTDIS von einem Time Out, also bevor der Timer die Timer-Flag setzt, gibt ihr Inhalt die noch verbleibende Zeit bis zum Setzen des Timer-Flags an. Nehmen wir dazu nochmals unser Zahlenbeispiel und die Timer-Zelle CNTC:
  - Timer-Offset = 1E = 30_{10}
  - Teilungsfaktor = 64
  - CNTC kann keinen IRQ auslösen
  - Die Zeit vom Start des Intervall-Timers bis zum Setzen der Timer-Flag ist: (30 µs x 64) + 1 µs = 1921 µs.

Das Starten des Intervall-Timers und das Lesen des Flag-Registers RDFLAG veranschaulicht das folgende Flußdiagramm. Dabei wird der Timer in Polling Mode einmal nichtperiodisch und einmal periodisch abgefragt. Das Flag-Register RDFLAG gibt über den momentanen Zustand des Timers Auskunft:



Der Timer wird mit dem Timer-Offset 1E gestartet. Nach (30 µs x 64) + 1 µs = 1921 µs wird die Timer-Flag gesetzt und kein IRQ ausgelöst. CNTC ermöglicht keinen Interrupt-Request.

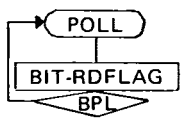
Der Prozessor arbeitet unabhängig vom Timer im Programm weiter. 1921 µs sind noch nicht verstrichen



Die Timer-Zelle wird **nichtperiodisch** gelesen. Die Timer-Flag ist nicht gesetzt, da der Timer noch nicht abgelaufen ist. Der Inhalt der Timer-Zelle sei 0C = 12_{10}

Bis zum Setzen der Timer-Flag verbleiben noch (12 µs x 64) + 1 µs = 769 µs. Der Prozessor arbeitet **unabhängig** vom Timer im Programm weiter.

•
•
•



Das Flag-Register wird **periodisch** gelesen. Der Prozessor arbeitet **abhängig** vom Timer in einer Warteschleife. Ist b7 in RDFLAG null, ist der Timer noch nicht abgelaufen. Ist b7 dagegen eins, dann ist der Timer abgelaufen und der Prozessor verläßt die Warteschleife POLL-BPL-POLL. Das Programm läuft wieder unabhängig vom Timer.

•
•
•

Time Out!!!

•

LDA-RDTDIS

Die Timer-Zelle wird nach dem Time Out gelesen. Dabei wird automatisch die Timer-Flag wieder zurückgesetzt. Der Inhalt der Timer-Zelle ist beispielsweise wieder A4.

•
•
•
•
•

A4 = 10100100
$\overline{A4}$ = 01011011 = 5B (Komplement)
$\overline{A4}$ + 1 = 5B +1 = 5C = 92_{10} (Zweierkomplement)

Seit dem Setzen der Timer-Flag sind 92 µs verstrichen. Seit dem Start des Intervall-Timers sind 1921 µs + 92 µs = 2013 µs verstrichen.

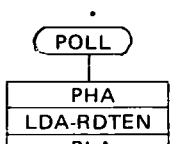
•
•
•

Der Timer dekrementiert alle µs um eins, bis er vom Prozessor wieder gestartet wird.

•
•
•

9. Wie zuvor gezeigt, läßt sich der Timer sowohl in Interrupt- als auch in Polling-Mode betreiben. Manchmal wünscht der Programmierer von einer Betriebsart in die andere umzuschalten. Das heißt, läuft der Timer beispielsweise in Polling-Mode und die Timer-Flag ist noch nicht gesetzt, kann der Programmierer den Intervall-Timer in Interrupt-Mode bringen, ohne diesen neu zu starten:

•
•



Der Intervall-Timer arbeitet in Polling-Mode

rette den Accu auf den Stack
schalte in Interrupt-Mode um
stelle den alten Zustand wieder her

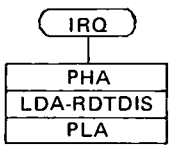
.
.
.

Der Intervall-Timer arbeitet in Interrupt-Mode. **Der Teilungsfaktor ändert sich nicht bei einer Leseoperation der Timer-Zelle.**

.
.
.

Die Umstellung von Interrupt-Mode auf Polling-Mode sieht so aus:

.
.



Der Intervall-Timer arbeitet in Interrupt-Mode

rette den Accu auf den Stack
schalte in Polling-Mode um
stelle den alten Zustand wieder her

.
.

Der Intervall-Timer arbeitet nun in Polling-Mode. **Der Teilungsfaktor ändert sich nicht bei einer Leseoperation der Timerzelle.**

.
.
.

10. **Lesen aus und Schreiben in die Timer-Zellen**
Beim Lesen aus und Schreiben in die Timerzellen sind folgende Besonderheiten zu beachten:

***Lesen der Timer-Zellen RDTDIS und RDTEN vor dem Time Out:**

- Durch eine Leseoperation erfährt der Mikroprozessor den momentanen Inhalt der Timer-Zelle. Für den Programmablauf kann der Computer berechnen, wie lange er noch Zeit hat, bis der Timer abgelaufen ist.
- Eine Leseoperation der Timer-Zelle RDTDIS hat zur Folge, daß der Timer in Polling-Mode gebracht wird und beim Time Out keinen IRQ auslösen kann. Beim Time Out wird nur die Timer-Flag gesetzt.
- Eine Leseoperation der Timer-Zelle RDTEN hat zur Folge, daß der Timer in IRQ-Mode gebracht wird und beim Time Out sich folgendes ereignet:
  a) die Timer-Flag wird gesetzt;
  b) die IRQ-Leitung springt von log. 1 nach log. Ø; dadurch löst der Timer einen IRQ aus.
- Die Teilungsfaktoren 1, 8, 64, 1Ø24 werden durch eine Schreiboperation in die Timer-Zellen CNTA . . . CNTH festgelegt. Eine Leseoperation der Timer-Zellen RDTDIS und RDTEN ändert den Teilungsfaktor nicht!

– Wird nach dem Time Out eine der Timer-Zellen gelesen, setzt diese Leseoperation die Timer-Flag und die Interrupt-Leitung zurück. Hingegen bleibt die Timer-Flag gesetzt, wenn die Leseoperation einer Timer-Zelle mit einem Timer-Interrupt zeitlich zusammenfällt.
– Hat der Timer einen Interrupt ausgelöst, verzweigt der Mikroprozessor in eine Interruptroutine. Dort sollte die Timer-Zelle RDTDIS gelesen werden, um die Interrupt-Leitung und die Timer-Flag zurückzusetzen. Dadurch ist sichergestellt, daß nach der Rückkehr von der Interrupt-Routine derselbe Interrupt nicht nochmals ausgelöst wird.

***Schreiben in die Timer-Zellen CNTA . . . CNTH:**

– Eine Schreiboperation in die Timer-Zellen CNTA . . . CNTD hat zur Folge, daß der Timer in Polling-Mode gestartet wird. Auch der Teilungsfaktor wird beim Timer-Start festgelegt und ändert sich nicht mehr bis zum Time Out.
– Eine Schreiboperation in die Timer-Zellen CNTE . . . CNTH hat zur Folge, daß der Timer in IRQ-Mode gestartet wird. Auch der Teilungsfaktor wird beim Timer-Start festgelegt und ändert sich nicht mehr bis zum Time Out.
– Eine Schreiboperation in eine der Timer-Zellen CNTA . . . CNTG setzt immer die Timer-Flag und die IRQ-Leitung zurück.

Die Beschreibung des Timers ist somit abgeschlossen. Wir haben gelernt, mit acht Timer-Zellen und der Timer-Flag zu arbeiten. Auch die beiden Betriebsarten Polling- und IRQ-Mode sind jetzt ein Begriff. Bevor wir mit dem Intervall-Timer an Hand von zwei Übungsprogrammen praktisch arbeiten, folgt noch die Beschreibung eines weiteren Bausteins im IC 6532: Der Flankendetektor.

Der Flankendetektor im IC 6532

Der Junior-Computer enthält auch einen Flankendetektor. Dieser Flankendetektor ist programmierbar und besteht aus vier Steuer-Registern. Die Namen dieser Register sind:
EDETA, EDETB, EDETC und EDETD. Bild 8 zeigt die Adressen dieser Register.

***Die Hardware des Flankendetektors**

Die Hardware des Flankendetektors läßt sich mit wenigen Worten beschreiben. Der Flankendetektor hat einen Eingang und einen Ausgang. Der Eingang ist die PA7-Leitung des peripheren Interface Adapters und der Ausgang ist die IRQ-Leitung des ICs 6532. Der Eingang des Flankendetektors läßt sich so programmieren, daß die IRQ-Leitung entweder bei einer positiven oder bei einer negativen Pulsflanke auf der PA7-Leitung aktiv wird. Der Programmierer hat jedoch auch die Möglichkeit, dem Flankendetektor zu verbieten, einen Interrupt Request auszulösen. **Arbeitet der Mikroprozessor mit dem Flankendetektor zusammen, ist die PA7-Leitung des PIA als Eingang zu programmieren.**

Bild 10 zeigt zwei häufige Anwendungen für den Flankendetektor. Zum einen führen wir PA7 ein serielles Datensignal zu und zum anderen verwenden wir alle Port A Leitungen, um eln paralleles 7 Bit breites Signal von einem ASCII-Keyboard in den Junior-Computer auszulesen.

– **Serielles Eingangssignal an PA7**
Wie wir aus Buch 1 wissen, verarbeitet der Computer alle Daten parallel. Die Daten haben dabei eine Breite von 8 Bits.

Soll der Junior-Computer beispielsweise mit einem Drucker, einem Videodisplay oder dem Elekterminal zusammenarbeiten, erfolgt der Datenaustausch seriell im ASCII-Code. Wie sich dieser Code aufbaut, besprechen wir ausführlich in Buch 3. Das praktische Arbeiten mit dem ASCII-Code ist an dieser Stelle noch nicht von Bedeutung.

Bild 10a zeigt den zeitlichen Ablauf eines seriellen ASCII-Signales. Es setzt sich aus folgende Bestandteilen zusammen:

1. ein Startbit
2. 8 Datenbits bØ . . . b7
3. ein Paritybit
4. zwei Stopbits.

Jedem Bit im seriellen ASCII-Signal ist eine bestimmte Bitzeit t2 zugeordnet. Der zeitliche Abstand von Bit zu Bit ist also konstant.

Die Übertragung einer seriellen Information beginnt immer mit einem Startbit. Ist die Leitung, auf der die serielle Information übertragen wird, log. 1, heißt das für den Computer, daß keine Daten folgen. Ein Spannungssprung von log. 1 nach log. Ø kennzeichnet also ein Startbit. Dieser Übergang läßt sich mit dem Flankendetektor im IC 6532 leicht aufspüren. Im vorliegenden Beispiel muß der Flankendetektor so programmiert werden, daß er auf eine negative Flanke reagiert (= Beginn eines Startbits).

Hat der Junior-Computer ein Startbit mit dem Flankendetektor erkannt, ist das Einlesen des seriellen ASCII-Signals eine Kleinigkeit für ihn:

1. Das Startbit ist erkannt (negative Flanke an PA7).
2. Der Mikrocomputer verzögert 1,5 Bitzeiten und nimmt dann eine "Probe" von PA7. Das heißt, er sieht nach, ob die Spannung an PA7 dem Pegel, log. Ø oder log. 1 entspricht. Somit kann er dem Datenbit bØ eine "0" oder eine "1" zuordnen. Eine Bitzeit später liegt das Datenbit b1 an der Port-Leitung PA7 an. Auch dann prüft der Junior-Computer wieder, ob es sich um eine "1" oder eine "0" handelt. Dieser Vorgang wiederholt sich solange, bis schließlich das Datenbit b7 in den Computer eingelesen ist. Die Bits werden dabei immer in der Bitmitte auf log. Ø- oder log. 1-Niveau geprüft.
3. Eine Bitzeit nach dem Datenbit b7 liegt das Paritybit an der PA7-Leitung an. Dieses Bit wird zu Testzwecken übertragen und gibt darüber Auskünft, ob das Bitmuster der soeben übertragenen Datenbits eine gerade oder ungerade Anzahl von Einsen oder Nullen hat. Der Computer kann dann leicht feststellen, ob bei der Datenübertragung ein Fehler aufgetreten ist.
4. Die Datenübertragung wird dann von zwei Stopbits abgeschlossen. Dabei ist zu bemerken, daß die beiden Stopbits die umgekehrte Polarität des Startbits haben (Startbit = log. Ø, Stopbit = log. 1).
5. Der Mikrocomputer wartet wieder, bis die Datenleitung log. Ø wird. Der Flankendetektor im IC 6532 kann diesen Spannungssprung aufspüren und es wiederholt sich der oben beschriebene Vorgang.

ADH = 1 A								ADL								
1				A												
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	128 bytes PIA-RAM
X(0)	X(0)	X(0)	X(1)	X(1)	X(0)	1	X(0)	0	X	X	X	X	X	X	X	(1A00 . . . 1A7F)
							1A80	1	X(0)	X(0)	X(0)	X(0)	0	0	0	PAD
							1A81	1	X(0)	X(0)	X(0)	X(0)	0	0	1	PADD
							1A82	1	X(0)	X(0)	X(0)	X(0)	0	1	0	PBD
							1A83	1	X(0)	X(0)	X(0)	X(0)	0	1	1	PBDD
							1AF4	1	X(1)	X(1)	1	0	1	0	0	CLK1T
							1AF5	1	X(1)	X(1)	1	0	1	0	1	CLK8T
						disable Timer-IRQ	1AF6	1	X(1)	X(1)	1	0	1	1	0	CLK64T
							1AF7	1	X(1)	X(1)	1	0	1	1	1	CLK1KT
							1AFC	1	X(1)	X(1)	1	1	1	0	0	CLK1T
							1AFD	1	X(1)	X(1)	1	1	1	0	1	CLK8T
						enable Timer IRQ	1AFE	1	X(1)	X(1)	1	1	1	1	0	CLK64T
							1AFF	1	X(1)	X(1)	1	1	1	1	1	CLK1KT
						disable Timer-IRQ	1AD4	1	X(1)	X(0)	X(1)	0	1	X(0)	0	read Timer
						enable Timer-IRQ	1ADC	1	X(1)	X(0)	X(1)	1	1	X(0)	0	read Timer
							1AD5	1	X(1)	X(0)	X(1)	X(0)	1	X(0)	1	read Flag Register
						disable PA7-IRQ	1AE4	1	X(1)	X(1)	0	X(0)	1	0	0	write neg EDET
						disable PA7-IRQ	1AE5	1	X(1)	X(1)	0	X(0)	1	0	1	write pos EDET
						enable PA7-IRQ	1AE6	1	X(1)	X(1)	0	X(0)	1	1	0	write neg EDET
Bild 9						enable PA7-IRQ	1AE7	1	X(1)	X(1)	0	X(0)	1	1	1	write pos EDET

Bild 9. Diese Tabelle zeigt, welches Bitmuster auf dem Adreßbus liegen muß, damit die einzelnen Register im IC 6532 adressiert werden. Mit der Leitung $\overline{RS}$ = A7 schaltet der Computer vom RAM auf den Timer und Flankendetektor um.

– Paralleles Eingangssignal an PORT A

Der Junior-Computer kann an PORT A auch parallele Eingangssignale einlesen (dasselbe gilt natürlich auch für PORT B). Bild 10b zeigt den Anschluß einer ASCII-Tastatur, die den ASCII-Code einer gedrückten Taste parallel angibt (bØ . . . b6). Dabei ist zu bemerken, daß der ASCII-Code eine Breite von sieben Bit hat (wird ausführlich in Buch 3 besprochen). Tastaturen geben in der Regel den Code parallel aus. Sobald der Tasten-Code auf den Datenleitungen bØ . . . b6 stabil steht, gibt die Tastatur einen Impuls ab. Dieser Impuls wird bei Computertastaturen "Data Strobe" genannt. Auch den Data Strobe kann der Flankendetektor leicht erkennen. Im Beispiel ist das eine positive Pulsflanke. Für den Junior-Computer heißt eine positive Pulsflanke an PA7: Auf den Datenleitungen des Keyboards steht ein Bitmuster, das in den Computer eingelesen werden soll.

Aus diesen beiden Anwendungen ist der Sinn und Zweck des Flankendetektors ersichtlich. Beim Datenaustausch zwischen dem Computer und den angeschlossenen peripheren Geräten entwickelt sich der Flankendetektor zu einer unentbehrlichen Einrichtung. In Buch 3 werden wir noch manche Anwendungsbeispiele für den Flankendetektor zeigen und beschreiben.

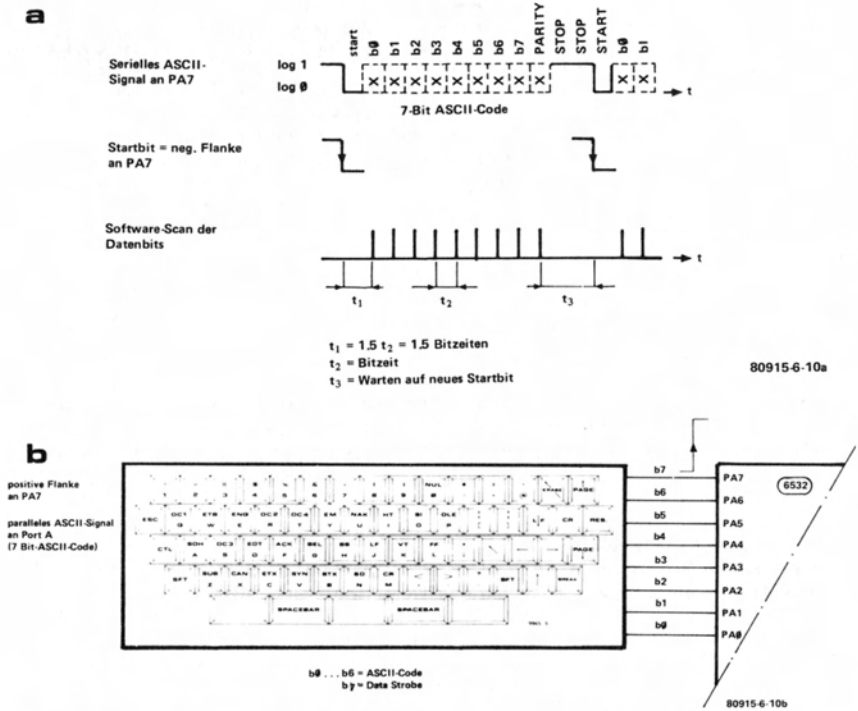


Bild 10. Praktische Anwendungen für den Einsatz des Flankendetektors. In Bild 10a ist die PA7-Leitung als Empfänger für ein serielles Signal programmiert. Der Flankendetektor wartet auf die negative Pulsflanke des Startbits. Trifft eine solche Flanke ein, beginnt der Computer mit dem Einlesen des seriellen Bitmusters. In Bild 10b liest der Junior-Computer ein paralleles Signal über PORT A. Eine positive Pulsflanke an PA7 teilt dem Computer mit, daß das Keyboard einen ASCII-Charakter bereithält.

Das Programmier-Modell des Flankendetektors

Von der Software aus gesehen besteht der Flankendetektor aus vier Steuerregistern. Ihre Namen sind EDETA, EDETB, EDETC und EDETD. Die Adressen dieser Steuer-Register sind aus Bild 9 ersichtlich. Durch das Schreiben eines beliebigen Bytes in eines dieser Register wird dem Flankendetektor mitgeteilt, ob er auf eine positive oder negative Pulsflanke an der Port-Leitung PA7 reagieren soll.

Der Flankendetektor arbeitet auch mit dem Flag-Register im IC 6532 zusammen. Dabei ist b6 im Flag-Register die Flag für den Flankendetektor. Wir nennen diese Flag in Zukunft **PA7-Flag.** Ist beispielsweise der Flankendetektor so programmiert, daß er auf eine negative Flanke an PA7 reagiert, wird beim Eintreffen einer solchen Flanke die PA7-Flag im Flag-Register gesetzt. Dasselbe gilt natürlich auch für eine positive Flanke an PA7.

Wie positive oder negative Flanken die PA7-Flag beeinflussen, können

solche Pulse auch zur Steuerung der Interrupt-Leitung verwendet werden. Erlaubt der Programmierer dem Flankendetektor (durch Schreiben eines beliebigen Bitmusters in eines der Steuer-Register) einen Interrupt Request auszulösen, geht die Interrupt-Leitung des ICs 6532 gleichzeitig mit dem Setzen der PA7-Flag auf null. Zwischen der Timer-Flag und der PA7-Flag läßt sich deshalb folgende Analogie angeben:

- **Timer-Flag:** Die Timer-Flag wird automatisch gesetzt, wenn sich ein Time Out ereignet. Der Programmierer kann dem Timer verbieten, einen IRQ auszulösen. Ist dem Timer erlaubt, einen IRQ auszulösen, wird gleichzeitig beim Setzen der Timer-Flag die IRQ-Leitung log. Ø.
- **PA7-Flag:** wird automatisch gesetzt, wenn an PA7 eine Pulsflanke eintrifft. Ob der Flankendetektor die PA7-Flag bei einer positiven oder negativen Pulsflanke setzt, entscheidet die Schreiboperation in eines der vier Steuer-Register EDETA . . . EDETD. Der Programmierer kann dem Flankendetektor verbieten, einen IRQ auszulösen. Ist dem Flankendetektor erlaubt, einen IRQ auszulösen, wird gleichzeitig beim Setzen der PA7-Flag die IRQ-Leitung log. Ø.
- **Zusammenfassung:** Die Timer- und die PA7-Flag sind im Flag-Register des ICs 6532 untergebracht. **Beide Flags sind voneinander unabhängig.** Beide Flags lassen sich in Polling-Mode abfragen.

Für die Timer-Flag gilt:

BIT-RDFLAG	teste die Timer-Flag
BPL	ist die Timer-Flag schon gesetzt? wenn ja, dann verzweige; wenn nein, dann warte.

Für die PA7-Flag gilt:

BIT-RGFLAG	teste die PA7-Flag
BVC	ist die PA7-Flag schon gesetzt? wenn ja, dann verzweige; wenn nein, dann warte.

Definition der Pulsflanke an PA7

Durch eine **Schreiboperation** in eines der Vier Steuer-Register EDETA . . . EDETD wird dem Flankendetektor mitgeteilt, ob er auf eine positive oder negative Pulsflanke an PA7 reagieren soll. Das Schreiben eines beliebigen Bitmusters in die Steuer-Register EDETA . . . EDETB hat zur Folge, daß beim Setzen der PA7-Flag kein Interrupt Request ausgelöst wird. Soll der Flankendetektor beim Eintreffen einer bestimmten Pulsflanke an PA7 einen Interrupt Request auslösen, ist eine Schreiboperation in eines der Steuer-Register EDETC . . . EDETD nötig. Die folgenden Beispiele verdeutlichen diesen Zusammenhang:

STA-EDETA: Der Flankendetektor reagiert auf eine negative Pulsflanke

an PA7. Trifft dort eine negative Pulsflanke ein, dann wird die PA7-Flag im Flag-Register gesetzt und **kein IRQ ausgelöst.**

STA-EDETB: Der Flankendetektor reagiert auf eine positive Pulsflanke

an PA7. Trifft dort eine positive Pulsflanke ein, dann wird die PA7-Flag im Flag-Register gesetzt und **kein IRQ ausgelöst.**

STA-EDETC: Der Flankendetektor reagiert auf eine negative Pulsflanke an PA7. Trifft dort eine negative Pulsflanke ein, dann wird die PA7-Flag im Flag-Register gesetzt und **ein IRQ ausgelöst.**

STA-EDETD: Der Flankendetektor reagiert auf eine positive Pulsflanke an PA7. Trifft dort eine positive Pulsflanke ein, dann wird die PA7-Flag im Flag-Register gesetzt und **ein IRQ ausgelöst.**

Weitere Besonderheiten des ICs 6532

Wie wir bereits wissen, arbeiten der Timer und der Flankendetektor mit dem Flag-Register RDFLAG zusammen. Dabei ist b7 die Timer-Flag und b6 die PA7-Flag. Erlaubt der Programmierer dem Timer oder dem Flankendetektor einen IRQ auszulösen, geht die IRQ-Leitung beim Setzen der Timer- oder PA7-Flag auf log. Ø. Um denselben Interrupt Request nicht mehrfach auszulösen, muß die IRQ-Leitung wieder zurückgesetzt werden, sobald der Prozessor in der Interruptroutine ist. Beim Zurücksetzen der IRQ-Leitung ist unbedingt folgendes zu beachten:

1) Der Timer löst einen IRQ aus: Hat der Programmierer dem Intervall-Timer erlaubt, einen Interrupt Request auszulösen, wird nach dem Time Out die Timer-Flag gesetzt; die IRQ-Leitung löst einen Interrupt Request aus. Die CPU verzweigt jetzt in eine Interrupt-Routine. Am Anfang dieser Routine muß die IRQ-Leitung zurückgesetzt werden, damit nach der Rückkehr aus der Interrupt-Routine (RTI) derselbe Timer-IRQ nicht nochmals einen Startimpuls erhält.

Das Zurücksetzen der IRQ-Leitung und der Timer-Flag während der Interrupt-Routine geschieht durch eine Schreiboperation in eine der Timer-Zellen CNTA . . . CNTH, oder durch eine Leseoperation aus den Timer-Zellen RDTDIS bzw. RDTEN. Eine Schreiboperation in eine der Timer-Zellen startet den Timer aufs neue und definiert, ob der Timer einen IRQ auslösen darf oder nicht. Eine Leseoperation aus einer Timer-Zelle startet den Intervall-Timer nicht aufs neue, sondern definiert nur den IRQ-Status (IRQ möglich oder nicht).

2) der Flankendetektor löst einen IRQ aus: Hat der Programmierer dem Flankendetektor erlaubt, einen Interrupt Request auszulösen, dann wird beim Eintreffen einer bestimmten Pulsflanke an PA7 die PA7-Flag gesetzt und die IRQ-Leitung löst einen Interrupt Request aus. Die CPU verzweigt nun in eine Interrupt-Routine. Während dieser Routine muß die IRQ-Leitung zurückgesetzt werden, damit nach der Rückkehr aus der Interruptroutine (RTI) derselbe PA7-IRQ nicht nochmals ausgelöst werden kann. **Das Zurücksetzen der IRQ-Leitung und der PA7-Flag während der Interruptroutine geschieht durch eine Leseoperation des Flag-Registers RDFLAG.**

3) Der IRQ-Vektor ist beim Junior-Computer in den RAM-Zellen 1A7E und 1A7F abgelegt. Das heißt, der Computer kann während eines Programms den IRQ-Vektor selbst bestimmen oder verändern. Somit ist es möglich, daß der Timer wie der Flankendetektor ihre eigenen Interrupt-Routinen zugeteilt bekommen.

Der Prozessor kann durch Lesen des Flag-Registers RDFLAG erfahren, welche der beiden Flags einen Interrupt ausgelöst hat. Ist beispielsweise die

Timer-Flag gesetzt, dann setzt der Computer den IRQ-Vektor auf die Timer-Interrupt-Routine oder im anderen Fall auf die PA7-Interrupt-Routine. Es ist somit möglich, beide Interrupt-Routinen ineinander zu verschachteln. Kommt es auf besonders hohe Verarbeitungsgeschwindigkeiten an, verwendet man häufig die Verschachtelung mehrerer Interrupt-Routinen. In einem folgenden Buch werden wir darauf noch eingehen.

4) **Die RES-Leitung:** Mit der RES-Leitung wird der Junior-Computer gestartet. Diese Leitung hat auch Einfluß auf diverse Register des ICs 6532. Wird diese Leitung log. Ø, geschieht folgendes:

- Der Inhalt aller I/O-Register ist null. Somit sind alle Leitungen von Port A und Port B als Eingänge erklärt. Periphere Geräte können jetzt nicht irrtümlicherweise die Ausgangs-Register PAD und PBD zerstören.
- Es kann kein Timer- und PA7-Interrupt Request ausgelöst werden. Somit ist ausgeschlossen, daß der Prozessor in eine Interrupt-Routine verzweigt, für die noch kein IRQ-Vektor gesetzt ist.
- Das RES-Signal programmiert den Flankendetektor auf eine negative Flanke an PA7. Während des Resets kann aber die PA7-Flag irrtümlicherweise durch ein peripheres Gerät gesetzt werden. Bevor man den Flankendetektor verwendet, sollte man deshalb die PA7-Flag durch eine Leseoperation des Flag-Registers RDFLAG zurücksetzen.

Somit haben wir alles über die Register des ICs 6532 erfahren. In den folgenden Musterprogrammen ist ausführlich gezeigt, wie der Timer und der PIA zu programmieren sind. Der Flankendetektor ist dabei bewußt ausgeklammert, da er Thema in einem folgenden Buch ist. Bild 9 zeigt auch noch, welches Bitmuster auf dem Adreßbus stehen muß, wenn der Mikroprozessor die diversen Register im IC 6532 ansprechen will. Dabei ist die Adreßleitung A7 als $\overline{RS}$-Leitung verwendet ($\overline{RS}$ = $\overline{RAM\ Select}$). Ist $\overline{RS}$ = log. Ø, dann sind im IC 6532 die 128 RAM-Zellen angesprochen. Ist aber $\overline{RS}$ = log. 1, arbeitet die CPU mit dem Intervall-Timer, dem PIA oder dem Flankendetektor zusammen. Für die Adressen bedeutet das: 1AØØ . . . 1A7F ist RAM-Speicher und 1A8Ø . . . 1AFF sind die Adressen für die übrigen Register. Es ist dabei zu erkennen, daß nicht alle Adressen von 1A84 . . . 1AFF Verwendung finden. Das hängt mit der nicht vollständigen Adreßkodierung auf dem 6532-Chip zusammen. Bei einem adressierbaren Speicherbereich von 65 KByte ist der Verlust von ein paar Speicherzellen durchaus vertretbar.

Bild 11 zeigt das gesamte Blockschaltbild des ICs 6532. Es ist die Hardware-Konfiguration von PIA, Intervall-Timer, Flankendetektor und RAM-Speicher. Die Software-Konfiguration dieser Elemente, also das Programmier-Modell, zeigt Bild 8.

Praktisches Arbeiten mit dem Intervall-Timer

1) **Der Timer arbeitet in Interrupt-Mode:** Jetzt ist es höchste Zeit, mit dem Intervall-Timer zu arbeiten. Dazu schreiben wir zwei Programme, genannt INPUT und REPEAT. Mit diesen beiden Programmen ist es möglich, in den Junior-Computer mit den Klaviaturtasten von Bild 4a eine Melodie einzugeben und anschließend wieder abzuspielen. Das heißt, der Computer speichert intern die Melodie in einem bestimmten Speicher-

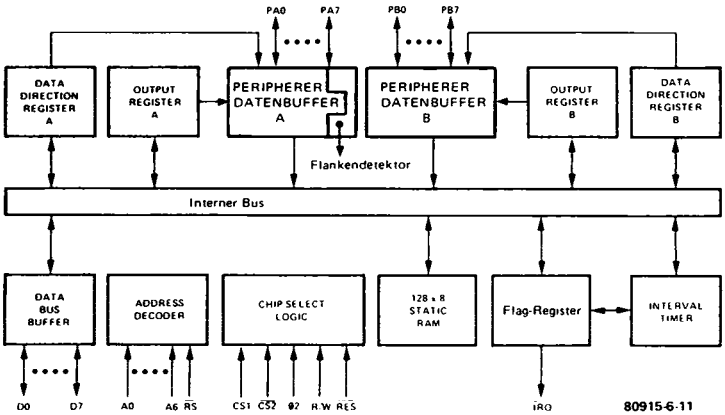


Bild 11. Das Blockschaltbild des ICs 6532. Der PIA, der Intervall-Timer, der Flankendetektor sowie das RAM sind über einen internen Datenbus verbunden. Alle bisher besprochenen Blöcke sind in einem Bild zusammengestellt.

bereich ab. Für die Eingabe der Melodie ist das INPUT-Programm bestimmt.
Ist die Melodie in den Junior-Computer eingegeben, soll er die Melodie selbständig wieder abspielen. Zur Ausgabe der Melodie ist das REPEAT-Programm vorgesehen. Dieser Abschnitt erklärt im erster Linie das praktische Arbeiten mit dem Intervall-Timer. Deshalb greifen wir wieder auf Programme zurück, die bereits aus dem PLAY-Programm (Bild 4) bekannt sind. Es sind dies die verwendeten Subroutinen KEYIN, KEYVAL und EQUAL. Auch das Interface, die Klaviaturtasten und den Verstärker für den Lautsprecher wollen wir unverändert übernehmen.
Das erste Problem ist die Eingabe einer Melodie in den Junior-Computer, die anschließend der Computer selbstständig abspielen, also wiederholen soll. Welche Information müssen wir in den Computer eingeben? Um einen Ton zu spielen, benötigen wir zwei Informationen:

1) Die Tonhöhe oder die Frequenz des Tones. Mit dem PLAY-Programm läßt sich jeder gedrückten Klaviaturtaste eine bestimmte Frequenz zuordnen. Dasselbe Prinzip verwenden wir auch beim INPUT-Programm.
2) Soll der Computer die eingegebene Melodie wiederholen, muß auch eine Aussage darüber bestehen, wie lange die Taste gedrückt war. Dazu ist es erforderlich, die Zeitdauer des Tastendrucks zu bestimmen. Diese Zeitdauer läßt sich auf einfache Weise mit dem Intervall-Timer messen.

Somit ist klar, daß der Computer für jeden gespielten Ton sowohl die Periodendauer des Tones, als auch die Dauer des Tastendrucks im Speicher ablegen muß. Zwei Speicherplätze müssen also für jeden gespielten Ton reserviert werden. Ein Maß für die Periodendauer des Tones ist der Wert der gedrückten Klaviaturtaste. Da 16 Klaviaturtasten vorhanden sind, liegt der Tastenwert zwischen Ø . . . F. Die Dauer des Tastendrucks mißt der Intervall-Timer. Diese Zeit soll die Werte ØØ . . . FF annehmen können.

Somit lassen sich die Forderungen angeben, die die Eingaberoutine INPUT erfüllen muß:

1) Wenn eine Klaviaturtaste gedrückt ist, soll im Lautsprecher ein Ton erklingen, der dieser Taste zugeordnet ist.
2) Gleichzeitig muß der Computer die Länge des Tastendrucks bestimmen.

Da scheinbar zwei Ereignisse parallel laufen, die Erzeugung des Tones und die Messung der Zeitdauer der gedrückten Taste, versagen herkömmliche Programmiertechniken. Deshalb müssen wir auf die Interruptprogrammierung ausweichen.

3) Jeder gedrückten Klaviaturtaste ist ein Wert zugeordnet. Der Computer bestimmt diesen Wert mit der Subroutine KEYVAL.
4) Die Klaviaturtastatur muß sich schnell abfragen lassen. Mit der bekannten Subroutine KEYIN ist das kein Problem. Ist irgendeine gedrückte Taste erkannt, muß ein Entprellprogramm jegliche Fehlermeldung ausschließen. Die bekannte Subroutine DELAY erledigt diese Forderung.
5) Ist die Klaviaturtaste losgelassen, dann soll der Lautsprecher verstummen und der Wert der losgelassenen Taste sowie die Zeitdauer des Tastendrucks im Speicher abgelegt werden. Der Melodienspeicher soll die Adressen Ø1ØØ . . . Ø1D8 in Page 1 haben. Ist dieser Speicherbereich aufgefüllt, soll sich der Junior-Computer automatisch über den Monitor zurückmelden und die Klaviatur ignorieren.
6) Bevor der Junior-Computer eine Melodie im Melodienspeicher ablegt, soll er diesen Speicherbereich auf die Eingabe der Melodie vorbereiten.

Diese Vorbereitung besteht darin, daß der gesamte Melodienspeicher mit 77 aufgefüllt und der Pointer für diesen Speicherbereich gesetzt wird.

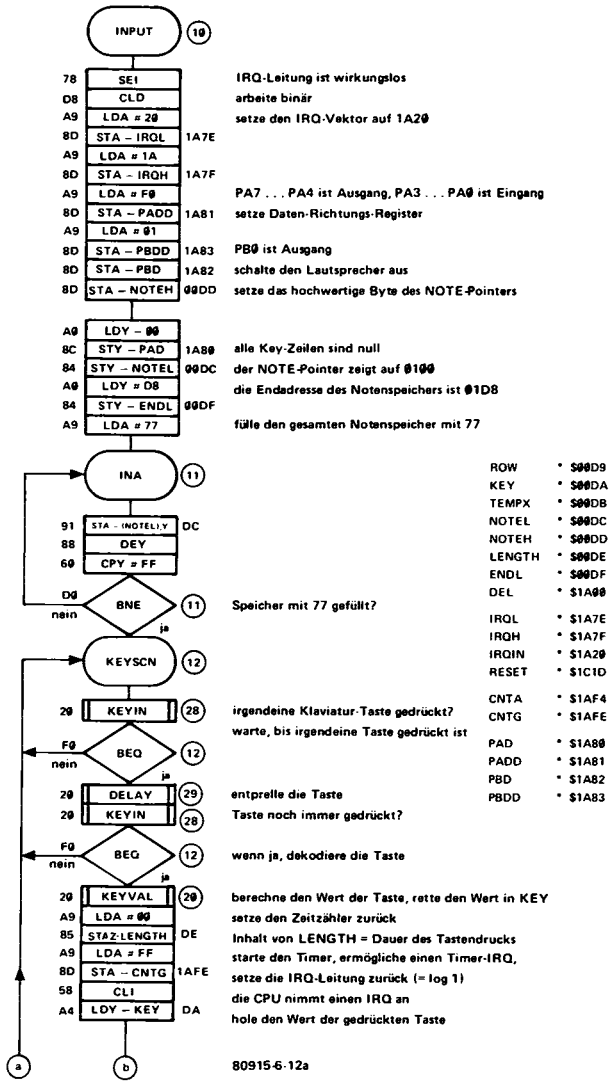
7) Werden beispielsweise die Klaviaturtasten mit den Tastenwerten 9, A, B und C hintereinander gedrückt, soll folgende Information im Melodienspeicher abgelegt werden:

Adresse:	**Daten:**	
Ø1ØØ	Ø9	= Tastenwert
Ø1Ø1	WW	= ØØ . . . FF = Zeitdauer des Tastendrucks
Ø1Ø2	ØA	= Tastenwert
Ø1Ø3	XX	= ØØ . . . FF = Zeitdauer des Tastendrucks
Ø1Ø4	ØB	= Tastenwert
Ø1Ø5	YY	= ØØ . . . FF = Zeitdauer des Tastendrucks
Ø1Ø6	ØC	= Tastenwert
Ø1Ø7	ZZ	= ØØ . . . FF = Zeitdauer des Tastendrucks
Ø1Ø8	77	= Füllcharakter der Vorbereitung
Ø1Ø9	77	
.	.	
.	.	
.	.	
.	.	
Ø1D7	77	
Ø1D8	77	= Ende des Melodienspeichers

Das INPUT-Programm

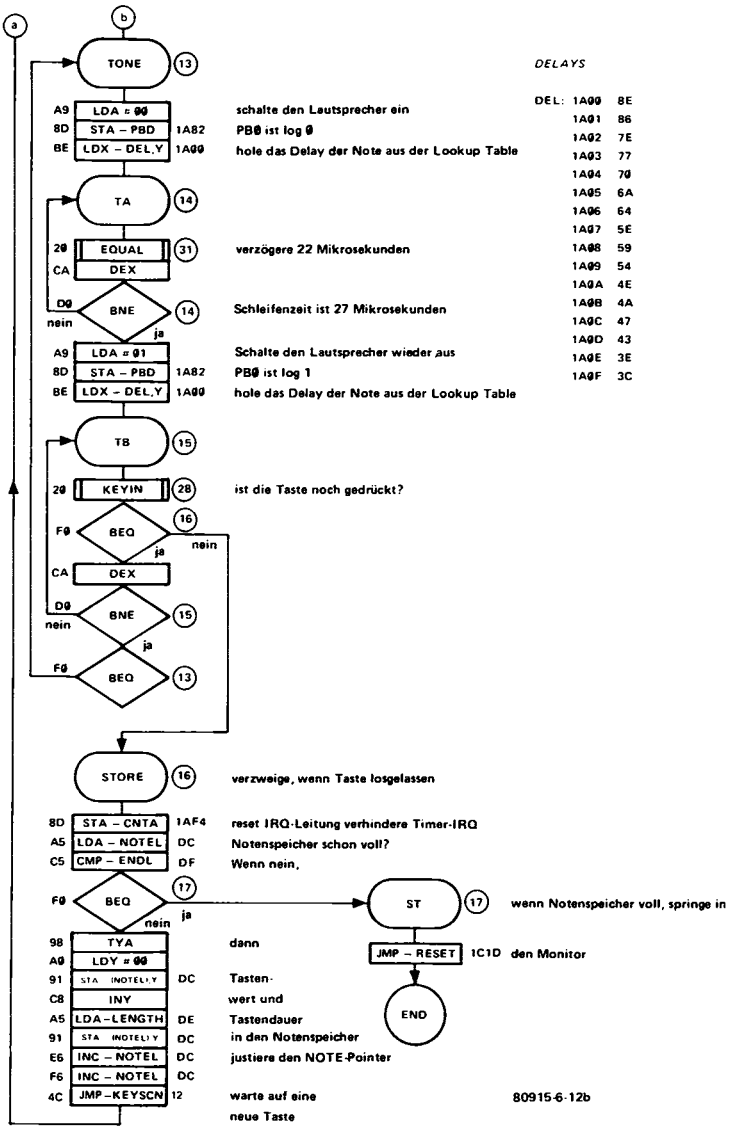
Die Forderungen 1) . . . 7) erfüllt das INPUT-Programm. Das Flußdiagramm zeigt Bild 12. Am Anfang macht das Programm die IRQ-Leitung wirkunglos. Der Befehl ist SEI. Das hat zur Folge, daß nicht irrtümlicher-

12a

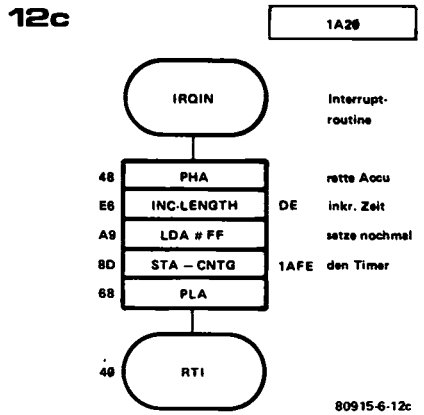


weise ein Interrupt ausgelöst werden kann. Dann setzt der Computer den IRQ-Vektor auf die Adresse 1A20. An dieser Stelle beginnt die Interrupt-Routine (Bild 12c), mit der die Dauer des Tastendruckes einer Klaviaturtaste gemessen wird. Die genaue Besprechung folgt zu einem späteren Zeitpunkt.
Es folgt die Programmierung des PIA. An PORT A ist gemäß Bild 4a die

12b



Matrix der Klaviatur angeschlossen. Deshalb mussen die Leitungen PA7 . . . PA4 zu Ausgängen und die Leitungen PA3 . . . PA0 zu Eingängen erklärt werden. Auch der Verstärker für den Lautsprecher ist wieder an PB0 angeschlossen. Deshalb muß diese Leitung als Ausgang programmiert sein.
Außerdem muß noch der NOTE-Pointer (Bild 12d) für den Melodienspei-



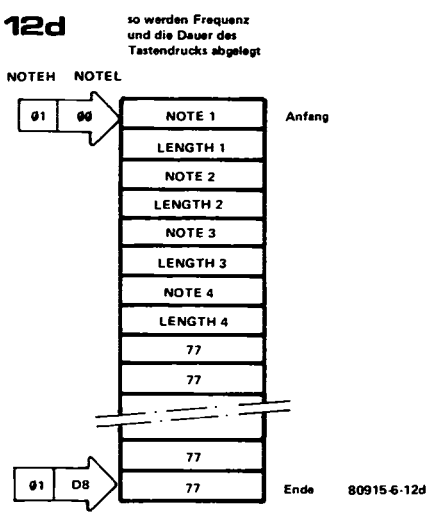


Bild 12a, b, c und d. Mit dem INPUT-Programm und der Klaviatur läßt sich eine Melodie im Speicher des Junior-Computers ablegen. Wird eine Taste gedrückt, ist im Lautsprecher der entsprechende Ton zu hören. Während der Computer den Ton erzeugt, mißt er auch noch die Dauer des Tastendrucks (darf bis zu vier Sekunden betragen). Um diese beiden scheinbar parallelen Ereignisse ausführen zu können, betreiben wir den Timer in IRQ-Mode. In dieser Routine muß wie beim PLAY-Programm die Umsetzung eines Tastenwertes in eine Frequenz vorgenommen werden. Der Junior-Computer verwendet dazu wieder die Lookup Table DEL (Bild 4g), aus der er die halben Periodendauern für die zu erzeugenden Frequenzen liest. Wird die Klaviaturtaste losgelassen, speichert der Computer den Tastenwert und die Dauer des Tastendrucks im Melodienspeicher in Page 1 ab. Das Programm beeinflußt den Adreßpointer NOTE (Bild 12d) so, daß er immer auf die Adresse zeigt, an welcher der nächste Ton (= Tastenwert) abgelegt werden soll.

cher gesetzt werden. Dieser Pointer zeigt auf eine Adresse in Page 1, an welcher der Wert für die nächste gedrückte Klaviaturtaste abgelegt werden soll. Am Anfang zeigt deshalb dieser Pointer auf die Adresse Ø1ØØ. Die Endadresse des Melodienspeichers ist Ø1D8. Zeigt der NOTE-Pointer über diese Adresse hinaus, soll der Junior-Computer in den Monitor zurückspringen. Der Wert $D8 wird deshalb in der Speicherzelle ENDL abgelegt. Der Vergleich des niederwertigen Adreßbytes des NOTE-Pointers mit dem Inhalt von ENDL gibt darüber Aufschluß, ob der Melodienspeicher schon voll ist oder nicht.

Jetzt ist der Computer beim Label INA angenommen. Der Melodienspeicher wird nun vorbereitet, indem der Computer die Adressen Ø1ØØ . . . Ø1D8 mit 77 füllt. Dann kümmert sich der Computer um die Abfrage der Klaviatur. Ist keine Taste gedrückt, verweilt er in der Warteschleife KEYSCN - BEQ - KEYSCN.

Ist aber der Accuinhalt nach der Rückkehr von KEYSCN ungleich null, bedeutet dies, daß irgendeine Klaviaturtaste gedrückt ist. Es erfolgt eine Entprellung (DELAY, Bild 4e). Ist danach die Taste immer noch gedrückt, springt der Computer in die Subroutine KEYVAL (Bild 4c). Dort berechnet er den Wert der gedrückten Taste. Da 16 Tasten zur Verfügung stehen, erstreckt sich der Wertebereich von ØØ . . . ØF. Der berechnete Tastenwert ist nach der Fückkehr von KEYVAL in der Speicherzelle KEY abgelegt. Die Flußdiagramme aller verwendeten Subroutinen zeigen die Bilder 4c, d, e, f. Die ausführliche Beschreibung ist bereits beim PLAY-Programm erfolgt.

Und jetzt wird's interessant! Nachdem der Computer den Wert der gedrückten Klaviaturtaste kennt, kann er mit dem Lautsprecher einen Ton erzeugen. Für die Erzeugung des Tones verwendet er wieder dieselbe Lookup Table DEL wie beim PLAY-Programm (Bild 4g). Die Tonerzeugung beginnt beim Label TONE. Das Programm ist unverändert von PLAY übernommen. Während der Computer im Lautsprecher den Ton für die gedrückte Klaviaturtaste erzeugt, muß er auch noch die Dauer des Tastendruckes messen. Dazu bedienen wir uns der Interrupt-Programmierung. Deshalb sind zwischen der Subroutine KEYVAL und dem Label TONE noch einige Instruktionen eingefügt. Sie haben folgende Bedeutung:

1) Der Inhalt der Speicherzelle LENGTH ist ein Maß für die Dauer des Tastendrucks. Bevor der Junior-Computer den Ton für eine gedrückte Taste erzeugt, muß der Inhalt von LENGTH zurückgesetzt werden (LDA #ØØ, STAZ-LENGTH). Dann startet die CPU den Intervall-Timer, indem sie das Timer-Offset FF in die Timer-Zelle CNTG schreibt. Holen wir uns nochmals die Eigenschaften der Timer-Zelle CNTG an Hand von Bild 8 ins Gedächtnis zurück!

CNTG:
- Der Teilungsfaktor ist 64.
- Der Intervall-Timer darf nach einem Time Out einen Interrupt Request auslösen.
- Ist der Intervall-Timer gestartet, löst er nach (FF μs x 64) + 1 μs = (255 μs x 64) + 1 μs = 16321 μs einen Interrupt Request aus.
- Beim Start des Intervall-Timers wird die IRQ-Leitung zurückgesetzt.

Ist der Timer gestartet und die IRQ-Leitung somit zurückgesetzt, bedient die CPU einen Timer Interrupt Request. Der folgende Befehl CLI bedeutet:

bediene einen Interrupt Request, falls der Timer einen IRQ auslöst.
Erst jetzt beginnt der Computer mit der Erzeugung des Tones. Dazu holt er aus der Speicherzelle KEY den Wert der gedrückten Taste und anschließend aus der Lookup Table DEL (Bild 4g) die Periodendauer des zu spielenden Tones. Für eine halbe Periodendauer ist der Lautsprecher eingeschaltet und für eine weitere halbe Periodendauer ausgeschaltet (Label TA: der Lautsprecher ist eingeschaltet; Label TB; der Lautsprecher ist ausgeschaltet). Während der Tonerzeugung fragt der Computer in der Subroutine KEYIN (Bild 4d) die Klaviatur ab. Er erhält somit darüber Auskunft, ob die Taste noch gedrückt ist oder nicht. Solange die Taste gedrückt ist, erzeugt der Junior-Computer im Lautsprecher einen Ton. Dabei verstreicht von Periodendauer zu Periodendauer eine bestimmte Zeit. Auch der Intervall-Timer läuft noch unabhängig von der CPU. Nach 16321 μs löst der Timer einen Interrupt Request aus. Die IRQ-Leitung wird log. Ø und der Computer verzweigt in die Interrupt-Routine IRQIN (Bild 12c). Dort rettet er zuerst den Inhalt des Accus auf den Stack (PHA) und inkrementiert anschließend den Inhalt der Speicherzelle LENGTH. Dann startet die CPU abermals den Intervall-Timer (LDA #FF, STA-CNTG) und setzt dabei die IRQ-Leitung wieder zurück. Der Timer läuft von jetzt an wieder unabhängig von der CPU. Vor der Rückkehr aus der Interrupt-Routine wird der ursprüngliche Inhalt des Accus wieder hergestellt (PLA).
Nachdem der Computer von der Interrupt-Routine zurückgekehrt ist, kümmert er sich wieder um die Erzeugung des Tones. Nach 16320 μs ist der Timer wieder abgelaufen. Der Prozessor verzweigt dann nach der Auslösung des Timer IRQs wieder in die Interruptroutine und inkrementiert den Inhalt der Speicherzelle LENGTH abermals um eins. Die maximale Zeitdauer eines Tastendrucks, die der Computer messen kann, ist somit:

255 x 64 x 255 μs = 4161600 μs = 4,1616 s.

Timer-Offset | Teilungsfaktor | maximaler Inhalt von LENGTH

(Dabei sind die Zeiten vernachlässigt, die der Computer benötigt, um einen Interrupt Request zu bedienen und in die er in der Interruptroutine verweilt). Der Computer kann also, während er einen Ton erzeugt, die Dauer des Tastendrucks messen. Die maximale Dauer des Tastendrucks kann bis zu vier Sekunden betragen. Wird diese Zeit überschritten, dann mißt der Junior-Computer eine falsche Zeit. Eine Fehlermeldung wird dabei nicht ausgegeben (um das Programm klein und übersichtlich zu halten).
Nach jedem Timer-Interrupt verläßt der Computer die Tonerzeugungsroutine, die beim Label TONE beginnt, um nach der Ausführung der Interruptroutine dorthin wieder zurückzukehren. Danach durchläuft die CPU periodisch die Subroutine KEYIN (Bild 4d). Ist die Klaviaturtaste schließlich losgelassen, kehrt der Prozessor aus KEYIN mit einem Wert im Accu zurück, der gleich null ist. Die Erzeugung des Tones wird somit abgebrochen und in den Speicherzellen KEY und LENGTH steht alles wissenswerte über die soeben losgelassene Taste:
– in KEY steht der Tastenwert;
– in LENGTH steht eine Zahl, die der Dauer des Tastendrucks entspricht.
Doch der Timer könnte in der Zwischenzeit wieder einen Interrupt

Request auslösen, was zu einem falschen Meßergebnis in der Speicherzelle LENGTH führt. Damit der Intervall-Timer nach dem Verlassen der Tonerzeugungsschleife nicht außer Kontrolle gerät, verbieten wir ihm einen neuen IRQ auszulösen. Das geschieht durch eine Schreiboperation in die Timer-Zelle CNTA (STA-CNTA). Wir hätten jedoch auch jede andere Timer-Zelle zwischen CNTA . . . CNTD wählen können. Eine Schreiboperation in diesen Zellen verbietet dem Intervall-Timer einen Interrupt Request auszulösen.
Anschließend vergleicht der Computer das niederwertige Adreßbyte des NOTE-Pointers mit dem Inhalt der Speicherzelle ENDL. Er erhält somit die Auskunft, ob der Melodienspeicher bereits aufgefüllt ist oder nicht.
Ist der Melodienspeicher noch nicht aufgefüllt, transportiert der Computer den Inhalt der Speicherzellen KEY und LENGTH in den Melodienspeicher. Die Zahlen 77 im Melodienspeicher werden jetzt von den Daten der soeben losgelassenen Klaviaturtaste überschrieben. Die CPU gibt dem NOTE-Pointer (2 mal INCZ-NOTEL) eine neue Adresse; bei ihr wird der Wert der nächsten gedrückten Klaviaturtaste abgelegt. Der Junior-Computer wartet auf die Betätigung einer neuen Klaviaturtaste.
Das Tastenprogramm der INPUT-Routine zeigt Bild 13. Für dieses Programm wählen wir den Speicherbereich Ø2ØØ . . . Ø3FF. Nach der Eingabe des INPUT-Programms soll der Assembler mit der ST-Taste gestartet werden. Deshalb müssen wir, bevor der Editor gestartet wird, in den Speicherzellen 1A7A und 1A7B den richtigen NMI-Vektor ablegen. Der NMI-Vektor zeigt auf die Startadresse des Assemblers. Nachdem die INPUT-Routine assembliert ist, muß in den Junior-Computer noch die Lookup-Table LEN und die Interrupt-Routine eingegeben werden. Auch das geht aus Bild 13 hervor. Ist schließlich alles geladen, kann man das INPUT-Programm starten. Die Klaviatur muß natürlich angeschlossen sein. Wir können nun wieder wie beim PLAY-Programm eine Melodie spielen. Allerdings speichert der Junior-Computer diese Melodie im Melodienspeicher ab. Das INPUT-Programm kann auf drei Arten verlassen werden:
- durch Drücken der RST-Taste;
- durch Auffüllen des gesamten Melodienspeichers mit Daten; der Computer verzweigt dann automatisch ins Monitorprogramm;
- durch Drücken der ST-Taste. Allerdings muß dann der NMI-Vektor auf die Startadresse der folgenden REPEAT-Routine zeigen (die Startadresse von REPEAT ist ØØØØ). Somit ist es möglich, eine Melodie in den Computer einzugeben und anschließend beliebig oft durch Drücken der ST-Taste zu wiederholen.

Achtung: Den Junior-Computer nach dem Laden der INPUT-Routine nicht ausschalten, da noch die REPEAT-Routine geladen werden muß. INPUT und REPEAT sind zwei voneinander unabhängige Programme. Beide arbeiten jedoch Hand in Hand: Das INPUT-Programm legt eine Melodie im Speicher ab und das REPEAT-Programm wiederholt die zuvor eingegebene Melodie.

Bild 13. Das Tastenprogramm der INPUT-Routine. Mit dem Editor fällt es nicht schwer, dieses relativ lange Programm in den Junior-Computer einzugeben. Die Interrupt-Routine und die Lookup Table DEL werden getrennt von INPUT eingegeben, da beide in einem anderen Speicherbereich sind. Ist die INPUT-Routine

assembliert, sollte man den Computer nicht mehr ausschalten, da noch die REPEAT-Routine zu laden ist. REPEAT ist die Umkehrung von INPUT: Die zuvor eingegebene Melodie wiederholt der Junior-Computer selbständig.

Tasten:		Display:	Bemerkung:
RST		XXXX XX	
AD	0 0 E 2	00E2 XX	
DA	0 0	00E2 00	} BEGAD zeigt 0200
+	0 2	00E3 02	}
+	F F	00E4 FF	} ENDAD zeigt 03FF
+	0 3	00E5 03	}
AD	1 A 7 A	1A7A XX	
DA	5 1	1A7A 51	} NMI-Vektor = 1F51
+	1 F	1A7B 1F	} (Start Assembler mit ST)
AD	1 C B 5	1CB5 20	Start Editor
GO		77	Editor startklar
INSERT	F F 1 0 0 0	FF 10 00	Label 10: INPUT
INPUT	7 8	78	SEI
INPUT	D 8	D8	CLD
INPUT	A 9 2 0	A9 20	LDA #20
INPUT	8 D 7 E 1 A	8D 7E 1A	STA-IRQL
INPUT	A 9 1 A	A9 1A	LDA #1A
INPUT	8 D 7 F 1 A	8D 7F 1A	STA-IRQH
INPUT	A 9 F 0	A9 F0	LDA #F0
INPUT	8 D 8 1 1 A	8D 81 1A	STA-PADD
INPUT	A 9 0 1	A9 01	LDA #01
INPUT	8 D 8 3 1 A	8D 83 1A	STA-PBDD
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	8 5 D D	85 DD	STAZ-NOTEH
INPUT	A 0 0 0	A0 00	LDY #00
INPUT	8 C 8 0 1 A	8C 80 1A	STY-PAD
INPUT	8 4 D C	84 DC	STYZ-NOTEL
INPUT	A 0 D 8	A0 D8	LDY #D8
INPUT	8 4 D F	84 DF	STYZ-ENDL
INPUT	A 9 7 7	A9 77	LDA #77
INPUT	F F 1 1 0 0	FF 11 00	Label 11: INA
INPUT	9 1 D C	91 DC	STA-(NOTEL),Y
INPUT	8 8	88	DEY
INPUT	C 0 F F	C0 FF	CPY #FF
INPUT	D 0 1 1	D0 11	BNE zu INA (Label 11)
INPUT	F F 1 2 0 0	FF 12 00	Label 12: KEYSCN
INPUT	2 0 2 8 0 0	20 28 00	JSR-KEYIN (Label 28)
INPUT	F 0 1 2	F0 12	BEQ zu KEYSCN (Label 12)
INPUT	2 0 2 9 0 0	20 29 00	JSR-DELAY (Label 29)
INPUT	2 0 2 8 0 0	20 28 00	JSR-KEYIN (Label 28)
INPUT	F 0 1 2	F0 12	BEQ zu KEYSCN (Label 12)
INPUT	2 0 2 0 0 0	20 20 00	JSR-KEYVAL (Label 20)
INPUT	A 9 0 0	A9 00	LDA #00
INPUT	8 5 D E	85 DE	STAZ-LENGTH
INPUT	A 9 F F	A9 FF	LDA #FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG

Tasten:		Display:	Bemerkung:
INPUT	5 8	58	CLI
INPUT	A 4 D A	A4 DA	LDYZ-KEY
INPUT	F F 1 3 0 0	FF 13 00	Label 13: TONE
INPUT	A 9 0 0	A9 00	LDA #00
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E 0 0 1 A	BE 00 1A	LDX-DEL,Y
INPUT	F F 1 4 0 0	FF 14 00	Label 14: TA
INPUT	2 0 3 1 0 0	20 31 00	JSR-EQUAL (Label 31)
INPUT	C A	CA	DEX
INPUT	D 0 1 4	D0 14	BNE zu TA (Label 14)
INPUT	A 9 0 1	A9 01	LDA #01
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E 0 0 1 A	BE 00 1A	LDX-DEL,Y
INPUT	F F 1 5 0 0	FF 15 00	Label 15: TB
INPUT	2 0 2 8 0 0	20 28 00	JSR-KEYIN (Label 28)
INPUT	F 0 1 6	F0 16	BEQ zu STORE (Label 16)
INPUT	C A	CA	DEX
INPUT	D 0 1 5	D0 15	BNE zu TB (Label 15)
INPUT	F 0 1 3	F0 13	BEQ zu TONE (Label 13)
INPUT	F F 1 6 0 0	FF 16 00	Label 16: STORE
INPUT	8 D F 4 1 A	8D F4 1A	STA-CNTA
INPUT	A 5 D C	A5 DC	LDAZ-NOTEL
INPUT	C 5 D F	C5 DF	CMPZ-ENDL
INPUT	F 0 1 7	F0 17	BEQ zu ST (Label 17)
INPUT	9 8	98	TYA
INPUT	A 0 0 0	A0 00	LDY #00
INPUT	9 1 D C	91 DC	STA-(NOTEL),Y
INPUT	C 8	C8	INY
INPUT	A 5 D E	A5 DE	LDA-LENGTH
INPUT	9 1 D C	91 DC	STA-(NOTEL),Y
INPUT	E 6 D C	E6 DC	INC-NOTEL
INPUT	E 6 D C	E6 DC	INC-NOTEL
INPUT	4 C 1 2 0 0	4C 12 00	JMP-KEYSCN (Label 12)
INPUT	F F 1 7 0 0	FF 17 00	Label 17 (ST)
INPUT	4 C 1 D 1 C	4C 1D 1C	JMP-RESET; letzte Instruktion von INPUT
INPUT	F F 2 0 0 0	FF 20 00	Label 20: KEYVAL
INPUT	A 9 F 7	A9 F7	LDA #F7
INPUT	8 5 D 9	85 D9	STAZ-ROW (00D9)
INPUT	A 2 0 4	A2 04	LDX #04
INPUT	F F 2 1 0 0	FF 21 00	Label 21: KEYA
INPUT	C A	CA	DEX
INPUT	3 0 2 0	30 20	BMI zu KEYVAL (Label 20)
INPUT	0 6 D 9	06 D9	ASLZ-ROW (00D9)
INPUT	A 5 D 9	A5 D9	LDAZ-ROW (00D9)
INPUT	8 D 8 0 1 A	8D 80 1A	STA-PAD
INPUT	A D 8 0 1 A	AD 80 1A	LDA-PAD
INPUT	2 9 0 F	29 0F	AND #0F
INPUT	C 9 0 F	C9 0F	CMP #0F
INPUT	F 0 2 1	F0 21	BEQ zu KEYA (Label 21)

Tasten:		Display:	Bemerkung:
INPUT	8 6 D B	86 DB	STXZ-TEMPX (ØØDB)
INPUT	8 5 D A	85 DA	STAZ-KEY (ØØDA)
INPUT	A 2 Ø Ø	A2 ØØ	LDX #ØØ
INPUT	F F 2 2 Ø Ø	FF 22 ØØ	Label 22: KEYB
INPUT	4 6 D A	46 DA	LSRZ-KEY (ØØDA)
INPUT	9 Ø 2 3	9Ø 23	BCC zu ROWA (Label 23)
INPUT	E 8	E8	INX
INPUT	E Ø Ø 4	EØ Ø4	CPX #Ø4
INPUT	D Ø 2 2	DØ 22	BNE zu KEYB (Label 22)
INPUT	F Ø 2 Ø	FØ 2Ø	BEQ zu KEYVAL (Label 2Ø)
INPUT	F F 2 3 Ø Ø	FF 23 ØØ	Label 23: ROWA
INPUT	A 5 D B	A5 DB	LDAZ-TEMPX (ØØDB)
INPUT	C 9 Ø 3	C9 Ø3	CMP #Ø3
INPUT	D Ø 2 4	DØ 24	BNE zu ROWB (Label 24)
INPUT	8 A	8A	TXA
INPUT	4 C 2 7 Ø Ø	4C 27 ØØ	JMP-KEYC (Label 27)
INPUT	F F 2 4 Ø Ø	FF 24 ØØ	Label 24: ROWB
INPUT	C 9 Ø 2	C9 Ø2	CMP #Ø2
INPUT	D Ø 2 5	DØ 25	BNE zu ROWC (Label 25)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø 4	69 Ø4	ADC #Ø4
INPUT	D Ø 2 7	DØ 27	BNE zu KEYC (Label 27)
INPUT	F F 2 5 Ø Ø	FF 25 ØØ	Label 25: ROWC
INPUT	C 9 Ø 1	C9 Ø1	CMP #Ø1
INPUT	D Ø 2 6	DØ 26	BNE zu ROWD (Label 26)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø 8	69 Ø8	ADC #Ø8
INPUT	D Ø 2 7	DØ 27	BNE zu KEYC (Label 27)
INPUT	F F 2 6 Ø Ø	FF 26 ØØ	Label 26: ROWD
INPUT	C 9 Ø Ø	C9 ØØ	CMP #ØØ
INPUT	D Ø 2 Ø	DØ 2Ø	BNE zu KEYVAL (Label 2Ø)
INPUT	8 A	8A	TXA
INPUT	1 8	18	CLC
INPUT	6 9 Ø C	69 ØC	ADC #ØC
INPUT	F F 2 7 Ø Ø	FF 27 ØØ	Label 27: KEYC
INPUT	8 5 D A	85 DA	STAZ-KEY (ØØDA)
INPUT	A 9 Ø Ø	A9 ØØ	LDA #ØØ
INPUT	8 D 8 Ø 1 A	8D 8Ø 1A	STA-PAD
INPUT	6 Ø	6Ø	RTS
INPUT	F F 2 8 Ø Ø	FF 28 ØØ	Label 28: KEYIN
INPUT	A D 8 Ø 1 A	AD 8Ø 1A	LDA-PAD
INPUT	2 9 Ø F	29 ØF	AND #ØF
INPUT	4 9 Ø F	49 ØF	EOR #ØF
INPUT	6 Ø	6Ø	RTS
INPUT	F F 2 9 Ø Ø	FF 29 ØØ	Label 29: DELAY
INPUT	A Ø F F	AØ FF	LDY #FF
INPUT	F F 3 Ø Ø Ø	FF 3Ø ØØ	Label 3Ø: DELA

Tasten:		Display:	Bemerkung:
INPUT	8 8	88	DEY
INPUT	D Ø 3 Ø	D0 30	BNE zu DELA (Label 30)
INPUT	6 Ø	60	RTS
INPUT	F F 3 1 Ø Ø	FF 31 00	Label 31: EQUAL
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	E A	EA	NOP
INPUT	6 Ø	60	RTS

Das INPUT-Programm ist mit allen Subroutinen in den Junior-Computer eingegeben und wir überprüfen es nochmals auf eventuelle Tippfehler:

Tasten:		Display:	Bemerkung:
SEARCH	F F 1 Ø	FF 10 00	INPUT beginnt beim Label 10
SKIP		78	
SKIP		D8	
SKIP		A9 20	
SKIP		8D 7E 1A	
.		.	
.		.	
.		.	
.		.	

Ist alles richtig in der File abgelegt? Wenn ja, dann können wir INPUT assemblieren: Dazu drücken wir auf die ST-Taste.
Dann müssen wir noch die Interrupt-Routine IRQUIN (Bild 12c) und die Lookup Table (Bild 6) in den Junior-Computer eingeben. Bei der Eingabe der Interrupt-Routine verwenden wir wieder den Editor. Die Interrupt-Routine müß nicht assembliert werden!

Tasten:		Display:	Bemerkung:
AD	Ø Ø E 2	00E2 XX	
DA	2 Ø	00E2 20	BEGAD 1A20 — 128 bytes PIA-RAM
+	1 A	00E3 1A	
+	7 9	00E4 79	ENDAD 1A79 — 128 bytes PIA-RAM
+	1 A	00E5 1A	
AD	1 C B 5	1CB5 20	Start Editor
GO		77	Editor startklar
INSERT	4 8	48	PHA
INPUT	E 6 D E	E6 DE	INCZ-LENGTH
INPUT	A 9 F F	A9 FF	LDA # FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG
INPUT	6 8	68	PLA
INPUT	4 Ø	40	RTI
RST			Rückkehr zum Monitor

Tasten:		Display:	Bemerkung:
AD	1 A 0 0	1A00 XX	LOOKUP TABLE LEN
DA	8 E	1A00 8E	LEN * $1A00
+	8 6	1A01 86	
+	7 E	1A02 7E	
+	7 7	1A03 77	
+	7 0	1A04 70	
+	6 A	1A05 6A	
+	6 4	1A06 64	
+	5 E	1A07 5E	
+	5 9	1A08 59	
+	5 4	1A09 54	
+	4 E	1A0A 4E	
+	4 A	1A0B 4A	
+	4 7	1A0C 47	
+	4 3	1A0D 43	
+	3 E	1A0E 3E	
+	3 C	1A0F 3C	
AD	0 2 0 0	0200	
GO			Programmstart

2) Der Intervall-Timer arbeitet abwechselnd in Interrupt- und Polling-Mode

Das REPEAT-Programm hat die Fähigkeit, die mit der INPUT-Routine eingegebene Melodie wieder auszugeben. Das heißt: Der Junior-Computer wiederholt selbständig die im Melodienspeicher abgelegte Melodie. Beim Abspielen der Melodie im Melodienspeicher sollen folgende Bedingungen erfüllt sein:

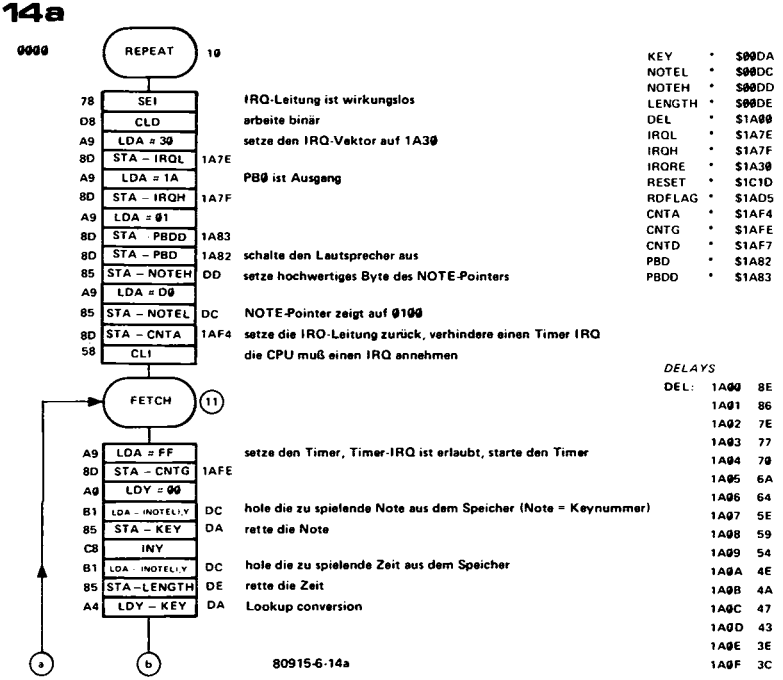
1) Der Computer setzt die im Melodienspeicher abgelegten Tastenwerte in einen Ton um. Bei der Tastenwert-Tonumsetzung findet wieder die Lookup Table LEN Verwendung.
2) Für jeden einzelnen Ton ist im Melodienspeicher die Dauer des Tastendrucks abgelegt. Die Interrupt-Routine setzt beim Wiederholen der eingegebenen Melodie die Tastendruckdauer in eine Tonlänge um (Interrupt-Mode).
3) Damit die Töne beim Wiederholen der Melodie nicht ineinander "verschwimmen", ist zwischen den einzelnen Tönen eine kleine Pause erforderlich. Der Junior-Computer erzeugt mit dem Timer diese Pause selbstständig. Dabei ist der Intervall-Timer in Polling-Mode zu betreiben.
4) Hat der Junior-Computer die gesamte Melodie im Melodienspeicher gespielt, meldet er sich über den Monitor zurück.

All diese Forderungen erfüllt die REPEAT-Routine. Ihr Flußdiagramm ist in Bild 14a und 14b dargestellt. Bei der Besprechung dieser Routine können wir uns kurz fassen, da zwischen REPEAT und INPUT viele Analogen bestehen.

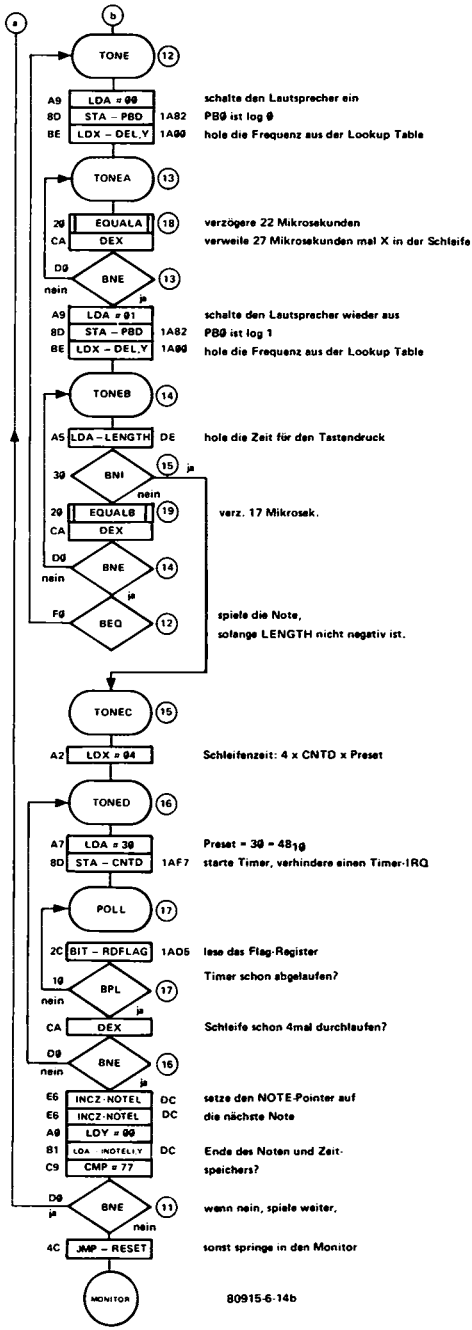
Am Anfang der REPEAT-Routine wird dem Prozessor wieder verboten, einen Interrupt Request anzunehmen. Dann setzt die CPU den IRQ-Vektor

für die REPEAT-Routine. Er zeigt auf die Adresse 1A3Ø (bei der INPUT-Routine zeigte der IRQ-Vektor auf die Adresse 1A2Ø; somit ist es möglich INPUT und REPEAT unabhängig voneinander zu betreiben).
Da an PBØ der Verstärker für den Lautsprecher angeschlossen ist, muß diese Port-Leitung als Ausgang programmiert sein. Auch der NOTE-Pointer ist neu einzurichten. Er zeigt zu Beginn auf die Adresse ØlØØ, wo der erste Tastenwert der zu spielenden Melodie abgelegt ist. Da die Timer-Flag und die IRQ-Leitung noch nicht definiert sind, ist eine Schreiboperation in eine der Timer-Zellen nötig. Die Instruktion STA-CNTA verhindert fürs erste einen Timer-IRQ und setzt die IRQ-Leitung zurück. Der folgende CLI-Befehl gibt die IRQ-Leitung frei. Das heißt: Die CPU muß von jetzt an einen Timer-Interrupt annehmen.
Nun ist der Mikroprozessor beim Label FETCH angekommen. Der Timer wird gestartet. Das Timer-Offset ist FF und der Teilungsfaktor ist wieder 64 wie bei der INPUT-Routine. Eine Schreiboperation in die Timer-Zelle CNTG teilt dem Intervall-Timer mit, daß er einen Interrupt Request auslösen darf. Die Instruktionen LDA # FF, STA-CNTG starten den Timer. Dieser läuft jetzt unabhängig von der CPU bis zum Time Out. Der Computer hat dabei genügend Zeit (FF μs x 64) + 1 μs, den Tastenwert und die Dauer des Tastendruckes aus den Melodienspeicher zu holen. Danach wird der Tastenwert mit Hilfe der Lookup Table LEN in eine Frequenz umgewandelt. Das geschieht ab dem Label TONE: Bei TONEA schaltet der Junior-Computer den Lautsprecher für eine halbe Periodendauer ein und

14a



14b


a
b
TONE
12
A9 LDA # 00
8D STA – PBD 1A82
BE LDX – DEL,Y 1A00
schalte den Lautsprecher ein
PB0 ist log 0
hole die Frequenz aus der Lookup Table
TONEA
13
20 EQUALA
CA DEX
18
verzögere 22 Mikrosekunden
verweile 27 Mikrosekunden mal X in der Schleife
D0 BNE
nein
ja
13
A9 LDA # 01
8D STA – PBD 1A82
BE LDX – DEL,Y 1A00
schalte den Lautsprecher wieder aus
PB0 ist log 1
hole die Frequenz aus der Lookup Table
TONEB
14
A5 LDA – LENGTH DE
hole die Zeit für den Tastendruck
30 BNI
nein
15
ja
20 EQUALB
CA DEX
19
verz. 17 Mikrosek.
D0 BNE
nein
ja
14
F0 BEQ
12
spiele die Note,
solange LENGTH nicht negativ ist.
TONEC
15
A2 LDX # 04
Schleifenzeit: 4 x CNTD x Preset
TONED
16
A7 LDA # 30
8D STA – CNTD 1AF7
Preset = $30 = 48_{10}$
starte Timer, verhindere einen Timer-IRQ
POLL
17
2C BIT – RDFLAG 1AD5
lese das Flag-Register
10 BPL
nein
ja
17
Timer schon abgelaufen?
CA DEX
Schleife schon 4mal durchlaufen?
D0 BNE
nein
ja
16
E6 INCZ-NOTEL DC
E6 INCZ-NOTEL DC
A0 LDY # 00
B1 LDA (NOTEL),Y DC
C9 CMP # 77
setze den NOTE-Pointer auf
die nächste Note
Ende des Noten und Zeit-
speichers?
D0 BNE
ja
nein
11
wenn nein, spiele weiter,
4C JMP – RESET
sonst springe in den Monitor
MONITOR
80915-6-14b

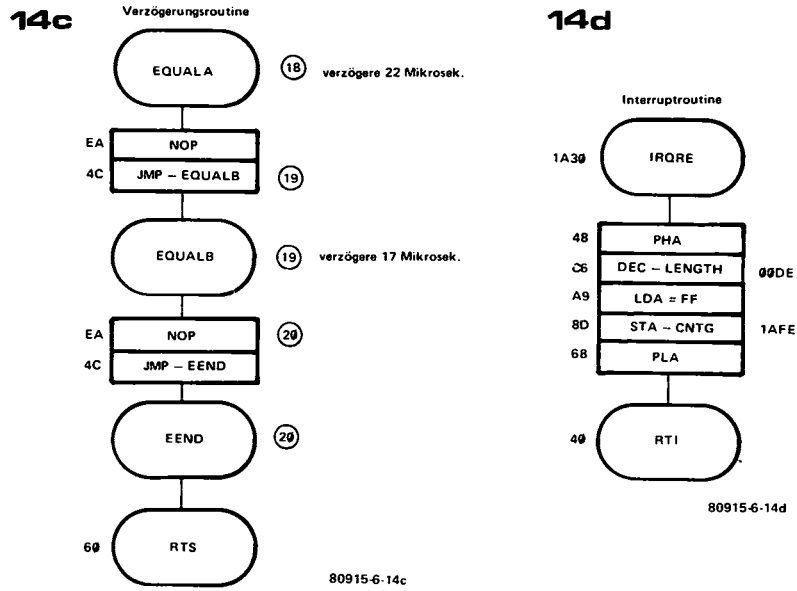


Bild 14. Mit der REPEAT-Routine ist es möglich, den Junior-Computer eine eingegebene Melodie wiederholen zu lassen. Damit beim Abspielen der Melodie die einzelnen Töne nicht ineinanddr "verschwimmen", erzeugt der Computer selbstständig von Ton zu Ton eine kleine Pause. Er verwendet dazu den Intervall-Timer. Während des REPEAT-Programms arbeitet der Intervall-Timer abwechselnd in Polling- und IRQ-Mode. REPEAT hat die Startadresse 0000.

bei TONEB für eine halbe Periodendauer wieder aus. Dabei ist zu beachten, daß die Schleifenzeiten bei der Tonerzeugung wieder wie bei der INPUT-Routine genau 27 μs betragen. Das Programm REPEAT benötigt bei der Tonerzeugung im Gegensatz zur INPUT-Routine zwei Ausgleichs-Subroutinen (Bild 14c): EQUALA (verzögert 22 μs) und EQUALB (verzögert 17 μs).
Der Junior-Computer erzeugt nun für mehrere Periodendauern einen Ton. Irgendwann ist der Timer abgelaufen und löst einen Timer-IRQ aus. Der Computer verzweigt dann in die Interrupt-Routine IRQRE, die bei der Adresse 1A30 beginnt (der IRQ-Vektor wurde am Anfang von REPEAT auf 1A30 gesetzt. Im Interruptprogramm IRQRE (Bild 14d) geschieht genau das Gegenteil dessen, was der Prozessor in IRQIN ausführt:

IRQRE: Der Prozessor dekrementiert den Inhalt der Speicherzelle LENGTH.

IRQIN: Der Prozessor inkrementiert den Inhalt der Speicherzelle LENGTH.

Beim REPEAT-Programm erzeugt also der Prozessor so lange einen Ton (ab Label TONE), bis der Inhalt der Speicherzelle LENGTH infolge mehrerer Timer-Interrupts negativ geworden ist.

Tritt diese Situation ein, verzweigt das Programm zum Label TONEC. Bevor der nächste Ton aus dem Melodienspeicher geholt und gespielt wird, muß der Computer eine kleine Pause machen, damit die einzelnen Töne nicht ineinander "verschwimmen". Wie anfangs gefordert, soll bei der Erzeugung dieser Pause der Timer in Polling-Mode arbeiten. Dazu verbieten wir bei TONED dem Timer, einen Interrupt Request auszulösen. Die Instruktionen LDA #3Ø und STA-CNTD bringen den Timer in Polling-Mode. Die Schreiboperation STA-CNTD hat folgenden Effekt:

- Der Intervall-Timer wird gestartet.
- Das Timer-Offset ist $3Ø = 48_{10}$.
- Die Timer-Flag im Flag-Register wird zurückgesetzt.
- Der Teilungsfaktor ist 1Ø24.
- Der Timer kann keinen Timer-IRQ nach einem Time Out auslösen.

Nachdem der Intervall-Timer gestartet ist, verweilt der Computer in der Warteschleife POLL - BPL - POLL solange, bis nach einem Time-Out (48 μs x 1Ø24) + 1 μs die Timer-Flag gesetzt wird. Dieser Vorgang wiederholt sich viermal, da nach jedem Time Out das X-Register um eins dekrementiert wird. Das heißt, die Gesamtzeit der Warteschleife verlängert sich um den Faktor vier. Ist die so erzeugte Pausenzeit zwischen zwei Tönen abgelaufen, stellt der Computer den NOTE-Pointer auf den nächsten zu spielenden Ton im Melodienspeicher (2 mal INCZ-NOTEL). Bevor er mit dem Spielen dieses Tones beginnt, wird geprüft, ob der NOTE-Pointer schon am Ende der Melodie angekommen ist. Das Ende der zu spielenden Melodie kündigt der EOF-Charakter 77 an (siehe auch Kapitel 5). Sind schließlich alle im Melodienspeicher abgelegten Töne gespielt, meldet sich der Junior-Computer über den Monitor zurück.

Das Tastenprogramm der REPEAT-Routine zeigt Bild 15. Wegen der ausführlichen Kommentare dürfte es nicht schwer fallen, das Programm in Page Ø zu assemblieren. Am Schluß ist noch für alle unmusikalischen Leser die Eingabe einer bekannten Melodie angegeben. Und nun viel Spaß beim Eingeben und Abspielen von Melodien.

Damit ist Kapitel 6 abgeschlossen. Es hat gezeigt, wie sich der Intervall-Timer, der Flankendetektor und der periphere Interface Adapter programmieren lassen. Diese etwas trockene und theoretische Materie ist durch ein paar Spielprogramme aufgelockert. In Zukunft dient dieses Kapitel häufig als Nachschlagwerk, wenn an den Junior-Computer periphere Geräte, wie beispielsweise ein Drucker oder das ELEKTERMINAL angeschlossen werden. In Buch 3 ist ein Cassetteninterface mit Software veröffentlicht. Auch dabei ist das IC 6532 zur Steuerung zweier Rekorder und zur Erzeugung des FSK-Signals verwendet. Der Junior-Computer mausert sich damit zu einem echten Minicomputer. Es ist dann nur noch ein kleiner Sprung zum Personal Computer, der höherer Programmiersprachen, wie zum Beispiel Basic versteht. Doch bevor es soweit ist, erklären wir den Monitor, das Editor-Programm und den Assembler, die im EPROM des Junior-Computers untergebracht sind. **Übrigens:** Für alle Programme in diesem Kapitel ist im Anhang dieses Buches das Source-Listing mit Kommentaren abgedruckt!

Bild 15. Das Tastenprogramm der REPEAT-Routine.

Tasten:		Display:	Bemerkung:
AD	0 0 E 2	00E2 XX	
DA	0 0	00E2 00	BEGAD 0000
+	0 0	00E3 00	
+	E 0	00E4 E0	ENDAD 00E0
+	0 0	00E5 00	
AD	1 A 7 A	1A7A XX	NMI-Vektor = 1F51
DA	5 1	1A7A 51	(Start Assembler mit ST)
+	1 F	1A7B 1F	
AD	1 C B 5	1CB5 20	Start Editor
GO		77	Editor startklar
INSERT	F F 1 0 0 0	FF 10 00	Lagel 10: REPEAT
INPUT	7 8	78	SEI
INPUT	D 8	D8	CLD
INPUT	A 9 3 0	A9 30	LDA #30
INPUT	8 D 7 E 1 A	8D 7E 1A	STA-IRQL
INPUT	A 9 1 A	A9 1A	LDA #1A
INPUT	8 D 7 F 1 A	8D 7F 1A	STA-IRQH
INPUT	A 9 0 1	A9 01	LDA #01
INPUT	8 D 8 3 1 A	8D 83 1A	STA-PBDD
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	8 5 D D	85 DD	STAZ-NOTEH
INPUT	A 9 0 0	A9 00	LDA #00
INPUT	8 5 D C	85 DC	STAZ-NOTEL
INPUT	8 D F 4 1 A	8D F4 1A	STA-CNTA
INPUT	5 8	58	CLI
INPUT	F F 1 1 0 0	FF 11 00	Label 11: FETCH
INPUT	A 9 F F	A9 FF	LDA #FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG
INPUT	A 0 0 0	A0 00	LDY #00
INPUT	B 1 D C	B1 DC	LDA-(NOTEL),Y
INPUT	8 5 D A	85 DA	STAZ-KEY
INPUT	C 8	C8	INY
INPUT	B 1 D C	B1 DC	LDA-(NOTEL),Y
INPUT	8 5 D E	85 DE	STA-LENGTH
INPUT	A 4 D A	A4 DA	LDYZ-KEY
INPUT	F F 1 2 0 0	FF 12 00	Label 12: TONE
INPUT	A 9 0 0	A9 00	LDA #00
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E 0 0 1 A	BE 00 1A	LDX-DEL,Y
INPUT	F F 1 3 0 0	FF 13 00	Label 13: TONEA
INPUT	2 0 1 8 0 0	20 18 00	JSR-EQUALA (Label 18)
INPUT	C A	CA	DEX
INPUT	D 0 1 3	D0 13	BNE zu TONEA (Label 13)
INPUT	A 9 0 1	A9 01	LDA #01
INPUT	8 D 8 2 1 A	8D 82 1A	STA-PBD
INPUT	B E 0 0 1 A	BE 00 1A	LDX-DEL,Y
INPUT	F F 1 4 0 0	FF 14 00	Label 14: TONEB

Tasten:		Display:	Bemerkung:
INPUT	A 5 D E	A5 DE	LDAZ-LENGTH
INPUT	3 Ø 1 5	3Ø 15	BMI zu Label TONEC (Label 15)
INPUT	2 Ø 1 9 Ø Ø	2Ø 19 ØØ	JSR-EQUALB (Label 19)
INPUT	C A	CA	DEX
INPUT	D Ø 1 4	DØ 14	BNE zu TONEB (LABEL 14)
INPUT	F Ø 1 2	FØ 12	BEQ zu TONE (Label 12)
INPUT	F F 1 5 Ø Ø	FF 15 ØØ	Label 15: TONEC
INPUT	A 2 Ø 4	A2 Ø4	LDX #Ø4
INPUT	F F 1 6 Ø Ø	FF 16 ØØ	Label 16: TONED
INPUT	A 9 3 Ø	A9 3Ø	LDA #3Ø
INPUT	8 D F 7 1 A	8D F7 1A	STA-CNTD
INPUT	F F 1 7 Ø Ø	FF 17 ØØ	Label 17: POLL
INPUT	2 C D 5 1 A	2C D5 1A	BIT-RDFLAG
INPUT	1 Ø 1 7	1Ø 17	BPL zu POLL (Label 17)
INPUT	C A	CA	DEX
INPUT	D Ø 1 6	DØ 16	BNE zu TONED (Label 16)
INPUT	E 6 D C	E6 DC	INCZ-NOTEL
INPUT	E 6 D C	E6 DC	INCZ-NOTEL
INPUT	A Ø Ø Ø	AØ ØØ	LDY #ØØ
INPUT	B 1 D C	B1 DC	LDA-(NOTEL),Y
INPUT	C 9 7 7	C9 77	CMP #77
INPUT	D Ø 1 1	DØ 11	BNE zu FETCH (Label 11)
INPUT	4 C 1 D 1 C	4C 1D 1C	JMP-RESET (Monitor; RESET = 1C1D)
INPUT	F F 1 8 Ø Ø	FF 18 ØØ	Label 18: Subroutine EQUALA
INPUT	E A	EA	NOP
INPUT	4 C 1 9 Ø Ø	4C 19 ØØ	JMP-EQUALB (Label 19)
INPUT	F F 1 9 Ø Ø	FF 19 ØØ	Label 19: EQUALB
INPUT	E A	EA	NOP
INPUT	4 C 2 Ø Ø Ø	4C 2Ø ØØ	JMP-EEND (Label 2Ø)
INPUT	F F 2 Ø Ø Ø	FF 2Ø ØØ	Label 2Ø: EEND
INPUT	6 Ø	6Ø	RTS

Testen des REPEAT-Programmes

Tasten:	Display:
SEARCH F F 1 Ø	FF 1Ø
SKIP	78
SKIP	D8
SKIP	A9 3Ø
SKIP	8D 7E 1A
.	.
.	.
.	.
.	.

Alles richtig eingetippt? Wenn ja, kann REPEAT assembliert werden:
ST Start des Assemblers
Jetzt geben wir die Interrupt-Routine IRQRE ein. Dieses Programm muß nicht assembliert werden!

Tasten:		Display:	Bemerkung:
AD	Ø Ø E 2	ØØE2 XX	
DA	3 Ø	ØØE2 3Ø	
+	1 A	ØØE3 1A	BEGAD 1AØ
+	7 9	ØØE4 79	
+	1 A	ØØE5 1A	ENDAD 1A79
AD	1 C B 5	1CB5 2Ø	Start Editor
GO		77	Editor startklar
INSERT	4 8	48	PHA
INPUT	C 6 D E	C6 DE	DEC-LENGTH
INPUT	A 9 F F	A9 FF	LDA # FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG
INPUT	6 8	68	PLA
INPUT	4 Ø	4Ø	RTI
RST			Rückkehr zum Monitor

Tasten:		Display:	Bemerkung:
			Die Lookup Table LEN muß nicht mehr geladen werden, da diese noch von der INPUT-Routine vorhanden ist. Ist zuvor mit der INPUT-Routine eine Melodie eingegeben worden, so spielt diese der Junior-Computer selbstständig nach dem Starten der REPEAT-Routine:
AD	Ø 2 Ø Ø	Ø2ØØ A9	Startadresse der INPUT-Routine
GO		dunkel	Wir spielen mit der Klaviatur: "Alle meine Entchen . . . . ." Da jeder Klaviatur Taste ein Wert zugeordnet ist, läßt sich sogar von unmusikalen Lesern diese einfache Melodie spielen:

Drücke Klaviatur-Taste:	Display:	Bemerkung:
Ø 2 4 5 7 7 9 9 9 7 5 5 5 5 4 4 2 2 2	dunkel	Die Melodie ist in den Junior-Computer eingegeben. Der Computer hat in Page 1 alle Tastenwerte und die Dauer des Tastendrucks abgelegt. Für jede Note sind also zwei speicherstellen belegt.
2 Ø		

Tasten:		Display:	Bemerkung:
AD	Ø Ø Ø Ø	Ø000 78	Startadresse der REPEAT-Routine
GO		dunkel,	solange der Computer die eingegebene Melodie spielt. Ist die gesamte Melodie gespielt, dann zeigt das Display:
		Ø000 78	Mit der GO-Taste läßt sich das Programm nochmals starten.

7

Das Monitorprogramm

Elementare Software für das Tastenfeld und das Display

Will der Programmierer mit einem Computer arbeiten, so muß der Computer die Fähigkeit haben, Daten vom Programmierer entgegenzunehmen und bearbeitete Daten wieder auszugeben. Zwischen dem Computer und dem Programmierer findet also eine Rückkopplung statt. Damit zwischen Mensch und Maschine überhaupt ein Dialog stattfinden kann, muß im Speicher des Computers ein Programm vorhanden sein. Das Programm, das diesen Dialog ermöglicht, heißt Monitorprogramm oder kurz Monitor. Der Monitor des Junior-Computers hat folgende Aufgaben:

* Adreß- und Datenanzeige auf dem 6-stelligen Display;
* Abfrage der Daten- und Befehlstasten;
* Steuerung des Displays im Multiplexbetrieb;
* Ausführung eines Tastenbefehls, den der Programmierer durch Drücken der Tasten AD, DA, +, PC und GO in den Computer eingibt.

Viele Subroutinen unterstützen das Monitorprogramm des Junior-Computers. Das hat den großen Vorteil, daß der Programmierer diese Subroutinen in eigene Programme einbauen kann und nicht erst selbst entwickeln muß. Ferner sind die Interrupt-Vektoren für IRQ und NMI in RAM-Zellen abgelegt. Somit lassen sich diese Vektoren während des Programms ständig ändern. Das ist wichtig, um mit einem und demselben Interrupt-Vektor mehrerer Interruptprogramme aufrufen zu können. Auf diese Zusammenhänge geht das Kapitel bei der Besprechung des Monitors näher ein.

Altbekanntes vom Monitor

Das Monitorprogramm des Junior-Computers ist in einem Teil des EPROMs abgelegt. In der Fachsprache bezeichnet man ein solches Programm als resident. Das heißt, es ist immer vorhanden, auch nach dem Ausschalten der Netzspannung. Arbeitet der Computer im Monitor, benötigt er auch einige Speicherzellen in Page Zero. In diesen Speicherzellen werden Daten abgelegt, auf die der Computer während des Monitorprogramms öfters zurückgreifen muß, z.B. um bestimmte Aufgaben auszuführen. Erinnern wir uns dazu an Buch 1! Dort beendet eine BRK-Instruktion viele Programme. Alle CPU-Register legt dabei das Monitorprogramm in Page Zero ab. Der Programmierer kann so nachsehen, was der Inhalt dieser Register ist, bevor der Junior-Computer ins Monitorprogramm zurückspringt und sich wieder um das Display und Tastenfeld kümmert.
Das Monitorprogramm des Junior-Computers ist eine Warteschleife. Befindet sich der Computer in dieser Programmschleife, fragt er periodisch das Tastenfeld ab und betreibt das 6-stellige Display im Multiplexbetrieb. Dabei transportiert er den Inhalt der drei Displaybuffer POINTH, POINTL und INH zu den Displays. Drückt der Programmierer eine der fünf Befehlstasten AD, DA, +, PC oder GO, dann führt der Computer die zu diesen Tasten gehörenden Handlungen aus. War beispielsweise die AD-Taste gedrückt, interpretiert der Computer die folgende Dateneingabe mit den Daten-Tasten Ø . . . F als Adresse. War die Daten-Taste gedrückt, dann wurden die eingegebenen Daten an der angezeigten Adresse abgelegt. Auch die restlichen Befehlstasten sind uns noch aus Buch 1 vertraut.
Es gibt mehrere Möglichkeiten, in den Monitor des Junior-Computers zu springen, oder aus diesem wieder herauszukommen:

1) Springen in den Monitor:

* Drücken der RST-Taste im Tastenfeld: Der Junior-Computer springt in den Monitor und deklariert die beiden Ports des peripheren Interfaceadapters. Danach verweilt der Computer in einer Warteschleife. In dieser Schleife steuert er das Display und das Tastenfeld. Der Junior-Computer ist jetzt über die Datentasten Ø . . . F sowie die Befehlstasten AD, DA, +, PC und GO ansprechbar.
* Auslösen eines NMI: Über den NMI-Vektor kann ebenfalls ein Sprung ins Monitorprogramm erfolgen. Zeigt der NMI-Vektor auf die Adresse 1CØØ, rettet der Computer alle internen CPU-Register. Der Junior-Computer verweilt danach wieder in einer Warteschleife. In dieser Schleife steuert er das Display sowie das Tastenfeld und ist dann über die Datentasten Ø . . . F sowie die Befehlstasten AD, DA, +, PC und GO ansprechbar. Der PIA wird nicht neu programmiert, wenn der NMI-Vektor auf die Adresse 1CØØ zeigt.
* Auslösen eines IRQ: Auch über den IRQ-Vektor kann ins Monitorprogramm gesprungen werden. Das ist jedoch nur möglich, wenn die I-Flag im P-Register der CPU zurückgesetzt ist (CLI). Zeigt der IRQ-Vektor auf die Adresse 1CØØ, gilt dasselbe wie für den NMI-Vektor.
* Sprung in den Monitor durch das BRK-Kommando: In Buch 1 haben wir viele Programme durch das BRK-Kommando abgeschlossen. Wie bekannt, arbeitet das BRK-Kommando mit dem IRQ-Vektor zusammen. Soll mit dem BRK-Kommando ein Sprung in den Monitor erfolgen,

muß der IRQ-Vektor auf die Adresse 1CØØ zeigen. Der Junior-Computer rettet dann alle internen CPU-Register und kommt in eine Warte-Schleife, wie bei RST, NMI und IRQ.

* Sprung in den Monitor durch einen JSR-Befehl: Der Programmierer kann die Subroutinen des Monitors in sein Programm einbauen. Der Rücksprung von der Monitor-Subroutine ins Programm des Programmierers erfolgt über einen RTS-Befehl am Ende der Subroutine.
* Sprung in den Monitor durch einen JMP-Befehl: Der Programmierer kann am Schluß seines Programms in den Monitor zurückspringen. Hat der Junior-Computer ein externes Programm abgearbeitet, leuchtet das Display wieder auf und der Programmierer kann somit erkennen, daß der Computer das eingetippte Programm abgearbeitet hat. Wird mit einem JMP-Befehl in den Monitor gesprungen, dann sollte der Sprung immer zur Adresse 1C1D (JMP-1C1D oder 4C 1D 1C) erfolgen. Ab dieser Adresse läuft eine Resetsequenz ab und der PIA wird wieder für die Tastenfeldabfrage und die Steuerung des Displays programmiert. Auch hier verweilt der Junior-Computer in einer Warteschleife wie zuvor bei RST, NMI, IRQ und BRK.

2) Springen aus dem Monitor:

* Drücken der GO-Taste: Die gebräuchlichste Art, das Monitorprogramm zu verlassen, ist das Drücken der GO-Taste. Nach dem Verlassen des Monitorprogramms fängt die CPU an der auf dem Display angezeigten Adresse zu arbeiten an. Das Display erlischt dann solange, bis wieder ein Rücksprung ins Monitorprogramm erfolgt.
* Auslösen eines NMI: Über den NMI-Vektor kann die CPU das Monitorprogramm verlassen. Steht am Ende der Interruptroutine ein RTI-Befehl, erfolgt wieder ein Rücksprung in den Monitor. Erfolgt aber nach dem Auslösen des NMI kein Rücksprung in den Monitor, dann sollte unmittelbar nach dem Verlassen des Monitorprogramms die I/O neu deklariert werden: Die 6-stellige Anzeige kann dann nicht willkürlich aufleuchten! Soll die Anzeige ganz ausgeschaltet sein, muß nach dem Verlassen des Monitors durch einen NMI oder IRQ die folgende Programmsequenz folgen:

```
(PHA)     rette den Accu
LDA # Ø6
STA-PBD   alle Display-Kathoden sind ausgeschaltet
LDA # ØØ
STA-PAD   kein Display-Segment zieht Strom
(PLA)     stelle den alten Accuzustand wieder her
```

  Dabei ist zu beachten, daß man den Accu nur dann retten muß, wenn sein Inhalt nach dem Verlassen des Monitors benötigt wird. Deshalb sind die PHA- und die PLA-Befehle eingeklammert.
* Auslösen einer IRQ: Über den IRQ-Vektor kann die CPU ebenfalls das Monitorprogramm verlassen. Voraussetzung ist allerdings, daß die I-Flag im P-Register der CPU zurückgesetzt sein muß, damit der Prozessor den IRQ bedienen kann. Ansonsten gilt dasselbe, was zuvor beim NMI beschrieben wurde.

Zur Erinnerung: Der NMI- und der IRQ-Vektor sind in Seite 1A in RAM-

Zellen abgelegt. Folgende Adressen sind diesen beiden Vektoren zugeordnet:
NMIL * $1A7A
NMIH * $1A7B
IRQL * $1A7E
IRQH * $1A7F
Der Inhalt dieser Speicherzellen entscheidet, wohin die CPU nach einem NMI, einem IRQ oder BRK verzweigt. Dabei bilden zwei Speicherzellen den Interrupt-Vektor, also
NMIH, NMIL = Non Maskable Interrupt-Vektor und
IRQH, IRQL = Interrupt ReQuest-Vektor. Diese beiden Vektoren kann der Programmierer sowohl von Hand setzen oder auch durch das Programm beeinflussen.
Die bisher gemachten Aussagen sind eine kurze Zusammenfassung von Kapitel 3 aus Buch 1. Diese Zusammenfassung ist aber nötig, um das große Flußdiagramm des Monitorprogramms zu verstehen.

Das globale Flußdiagramm des Monitors

Bild 1 zeigt das globale Flußdiagramm des Monitors. Dort sind die gebräuchlichsten "Ein- und Ausgänge" zusammengestellt. Das Monitorprogramm hat drei Eingänge und zwei Ausgänge. Das Verlassen des Monitors via NMI oder IRQ ist dabei **nicht** berücksichtigt!

Die Ein- und Ausgänge des Monitors

Eingang ① führt zum Label RESET. Die CPU kommt über diesen Eingang in das Monitorprogramm, wenn die RST-Taste betätigt wurde. Dem Label RESET ist die Adresse 1C1D zugeordnet. Ab diesem Label programmiert der Prozessor den PIA und inititialisiert den Stack-Pointer. Auch das Prozessor-Status-Register wird in einem definierten Zustand gebracht. Wie das im einzelnen geschieht, besprechen wir später. Es ist jetzt nur von Wichtigkeit zu wissen, daß zwischen dem Label RESET und START der Junior-Computer darauf vorbereitet wird, das Display zu steuern und das Tastenfeld abzufragen. Ferner arbeitet von nun an der Computer als binär denkende Maschine (CLD) und die I-Flag im P-Register der CPU ist gesetzt. Der Junior-Computer kann also keinen IRQ bedienen.

Zusammenfassung: Es gibt mehrere Möglichkeiten, um über den Eingang ① in den Monitor des Junior-Computers zu kommen:

- Drücken der RST-Taste;
- durch einen Sprungbefehl zum Label RESET: JMP-1C1D
- durch einen NMI: der NMI-Vektor zeigt auf die Adresse 1C1D, das heißt: NMIH, NMIL = 1C1D. In den Speicherzellen 1A7A und 1A7B müssen folgende Bytes abgelegt sein:
  NMIL = $1A7A enthält 1D und
  NMIH = $1A7B enthält 1C
  } Label RESET
- durch einen IRQ (falls die I-Flag null ist): der IRQ-Vektor zeigt auf die Adresse 1C1D, das heißt:
  IRQH, IRQL = 1C1D. In den Speicherzellen 1A7E und 1A7F müssen folgende Bytes abgelegt sein:
  IRQL = $1A7E enthält 1D und
  IRQH = $1A7F enthält 1C
  } Label RESET

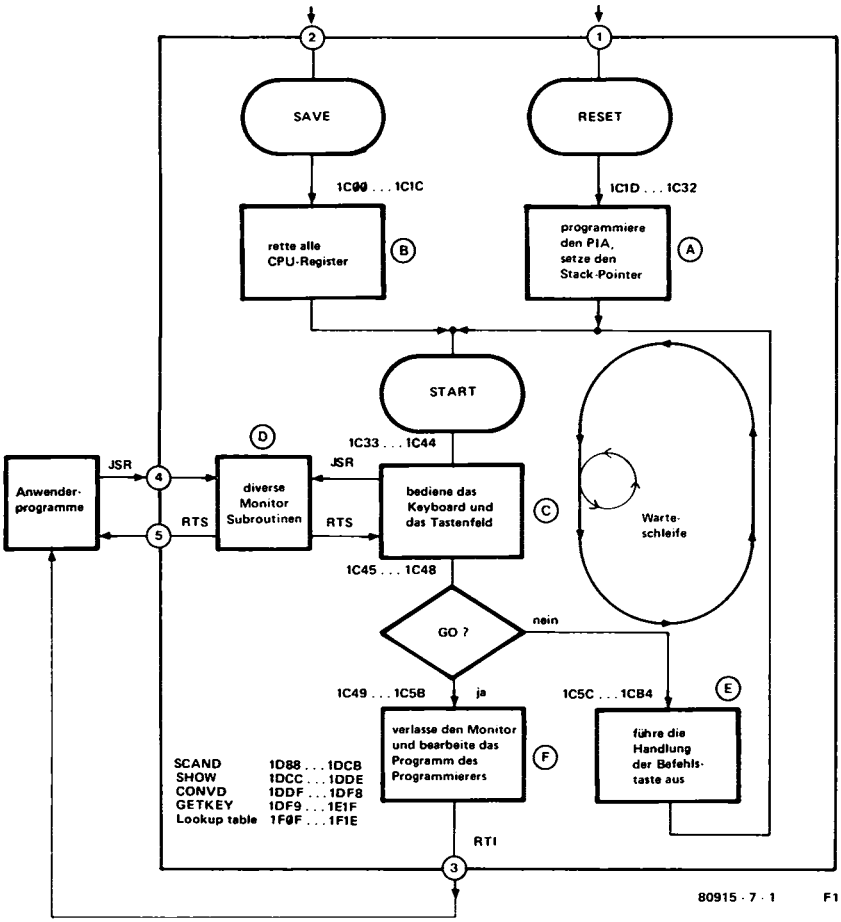


Bild 1. Das globale Flußdiagramm des Monitors. Das Monitor-Programm hat drei Eingänge und zwei Ausgänge. Diverse Adressen in diesem globalen Flußdiagramm kann man auch im Source Listing im Anhang dieses Buches nachschlagen.

– durch das BRK-Kommando: das BRK-Kommando arbeitet über den IRQ-Vektor. Soll mit diesem Kommando beim Eingang ① in den Monitor gesprungen werden (Bild 1), muß der IRQ-Vektor auf die Adresse 1C1D zeigen.

Eingang ② führt zum Label SAVE. Diesem Label ist die Adresse 1C00 (Label SAVE) zugeordnet. Zwischen den beiden Labels SAVE und START werden alle internen CPU-Register in Page Zero an folgenden Adressen abgelegt:

PCL: an der Adresse 00EF
PCH: an der Adresse 00F0
P-Register: an der Adresse 00F1

User Stack-Pointer: an der Adresse ØØF2
Accumulator: an der Adresse ØØF3
Y-Register: an der Adresse ØØF4
X-Register: an der Adresse ØØF5
Achtung! Über Eingang ② sollte nur mit den Interrupt-Vektoren in den Monitor gesprungen werden. Nur dann werden die CPU-Register PCH, PCL und das Status-Register korrekt gerettet. Auch der Stack-Pointer hat dann nach abermaligen Verlassen des Monitors mit der GO-Taste die richtige Position. Eine genaue Begründung für diesen Sachverhalt folgt anschließend. **Wählt der Programmierer eine andere Art und Weise für den PIA, ist die I/0 für die Steuerung des Displays und die Abfrage des Tastenfeldes beim Rücksprung in den Monitor über Eingang ② nicht definiert!**

Zusammenfassung: Es gibt mehrere Möglichkeiten, um über Eingang ② in den Monitor des Junior-Computers zu kommen:
- Drücken der ST-Taste: dadurch wird ein NMI ausgelöst.
  Der NMI-Vektor ist dabei auf die Adresse 1CØØ auszurichten, das heißt: NMIH, NMIL = 1CØØ. In den Speicherzellen 1A7A und 1A7B müssen folgende Bytes abgelegt sein:
  NMIL = $1A7A enthält ØØ und
  NMIH = $1A7B enthält 1C.
- Durch einen NMI, ausgelöst in Step by Step Mode: Bereits in Buch 1 sind Programme in Step by Step Mode ausgeführt worden. Damit ist es möglich, ein Programm schrittweise zu durchlaufen. Oder genauer gesagt: Die CPU führt nach dem Drücken der GO-Taste den auf dem Display angezeigten Befehl aus. Anschließend zeigt das Display den nächsten Befehl und rettet danach alle CPU-Register in Page Zero. Damit der Junior-Computer in Step by Step Mode arbeiten kann, muß man zuerst den NMI-Vektor auf die Adresse 1CØØ setzen (Label SAVE). Dieses schrittweise Durchlaufen eines Programms mit dem NMI-Vektor erklären wir anschließend noch genauer.
- Durch einen NMI, ausgelöst von irgendeinem peripheren Gerät, wie zum Beispiel einem Drucker etc. Auch hier kann der NMI-Vektor auf die Adresse 1CØØ zeigen.
- Durch einen IRQ (falls die I-Flag null ist): der IRQ-Vektor zeigt auf die Adresse 1CØØ, das heißt:
  IRQH, IRQL = 1CØØ. In den Speicherzellen 1A7E und 1A7F müssen folgende Bytes abgelegt sein:
  IRQL = $1A7E enthält ØØ und
  IRQH = $1A7F enthält 1C.
- Durch das BRK-Kommando sollte man nicht mehrmals hintereinander über den Eingang ② in den Monitor springen. Der Einfachheit halber sind wir trotzdem in Buch 1 über BRK und Eingang ② in den Monitor verzweigt. Dabei kann sich folgender Fehler einstellen:
  Der Stack läuft über.
  Auf dem Stack wird der falsche Programm Counter abgelegt (PC + 2)

* Eingang ④ führt zu diversen Subroutinen im Monitorprogramm. Diese Subroutinen kann der Programmierer in eigene Programme einbauen und mit dem JSR-Befehl aufrufen. Der RTS-Befehl am Schluß dieser Subrou-

tinen führt den Prozessor wieder zurück zum Programm des Programmierers (Ausgang 5).

Somit sind alle Aufgaben bekannt, welche die CPU zwischen den Labels RESET und START sowie SAVE und START ausführt. Ab dem Label START befindet sich der Junior-Computer in einer Warteschleife, die durch das Drücken der GO-Taste verlassen werden kann. In dieser Warteschleife führt der Computer folgende Aufgaben aus:

1. Steuerung des Displays und Abfrage des Tastenfeldes. Dabei werden diverse Monitor-Subroutinen aufgerufen.
2. Ist eine Taste gedrückt, fragt der Computer, ob es sich um die GO-Taste handelt. Wenn es **nicht** die GO-Taste ist, verweilt der Junior-Computer im Monitor. War eine Befehlstaste gedrückt, führt er die Handlung des Befehls aus. War es eine Datentaste, werden diese dem Adreß- oder Datendisplay zugeordnet. Anschließend erfolgt am Label Start der Vorgang erneut.
3. Ist in der Warteschleife irgendwann die GO-Taste gedrückt, verläßt der Computer das Monitor-Programm. Über den RTI-Befehl verzweigt er dann zum Programm des Programmierers. Diesen Vorgang beschreiben wir noch ausführlich beim detaillierten Flußdiagramm des Monitors. Normalerweise wird der Monitor immer über den Ausgang ③ verlassen.

Kurzes Intermezzo: Was ist Step by Step?

Bei der Besprechung des globalen Monitor-Flußdiagramms dürfen wir die Step by Step Mode nicht übergehen. In dieser Mode arbeiten Hardware und Sofware des Junior-Computers Hand in Hand. Um das zu zeigen, betrachten wir nochmals das Schaltbild des Junior-Computers (Buch 1, Kapitel 1, Bild 4). Pin 7 der CPU (IC1, SYNC-Ausgang) ist mit dem NAND-Gatter N5 verbunden. Der andere Gattereingang bekommt das $\overline{CS}$-Signal K7 zugeführt. Der Gatterausgang ist mit der NMI-Leitung des Prozessors verbunden. Wenn der Schalter S24 geschlossen ist, arbeitet der Computer in Step by Step Mode. Die Step by Step Mode läßt sich folgendermaßen beschreiben:

a. Das Display zeigt eine Adresse, an der ein Befehl abgelegt ist. Der Schalter S24 ist geschlossen.
b. Der Programmierer möchte den momentan angezeigten Maschinenbefehl ausführen lassen. Dazu drückt er die GO-Taste. Der Prozessor verläßt den Monitor über Ausgang ③ und fängt an der angezeigten Adresse zu arbeiten an, in dem er den Maschinenbefehl aus dem Speicher holt.
c. Während der Prozessor den Maschinenbefehl (OP-Code) aus dem Speicher holt, springt der SYNC-Ausgang der CPU (Pin 7) auf log. 1. Das EPROM IC2 ist während dieser Zeit nicht adressiert und die $\overline{CS}$-Leitung K7 ist somit log. 1. Deshalb ruft der SYNC-Ausgang der CPU am Ausgang des NAND-Gatters N5 eine negative Flanke hervor.
d. Diese negative Flanke verursacht einen NMI. Der Prozessor führt den auf dem Display angezeigten Befehl aus und bedient dann den NMI. Über den NMI-Vektor gelangt die CPU zum Label SAVE mit der Adresse 1CØØ (Eingang ②).
e. Der Prozessor befindet sich jetzt wieder im Monitorprogramm. Somit ist das EPROM IC2 adressiert und die Leitung K7 ist log. Ø. Es kann kein NMI ausgelöst werden, da der Ausgang des NAND-Gatters N5

immer log. 1 bleibt. Ab dem Label SAVE rettet die CPU alle internen Register in Page Zero und der Programm Counter PC zeigt auf den OP-Code der folgende Instruktion. Ab dem Label SAVE gelangt der neue Inhalt des Programm Counters zu den Displaybuffern POINTH und POINTL. Ab dem Label START zeigt dann der Computer den folgenden Befehl mit der dazugehörenden Adresse auf dem Display an. Erneutes Drücken der GO-Taste löst den soeben beschriebenen Vorgang nochmals aus.

Detaillierter Ablauf der Interrupts und des BRK-Kommandos

Bis jetzt ist nur klar, wo die Interrupt-Vektoren für NMI und IRQ abgelegt sein müssen, damit die CPU einen Interrupt bedienen kann. Dabei ist noch nicht deutlich, wie die CPU aus den Speicherzellen 1A7A, 1A7B und 1A7E, 1A7F den NMI- und den IRQ-Vektor holt. **Bild 2** gibt über diese Vorgänge ausführlich Auskunft. Auf der linken Seite finden wir einen Speicherraum, der aus mehreren Teilen besteht: Der obere Teil besteht aus RAM-Zellen, in denen das Programm des Programmierers abgelegt ist. Das sind bekanntlich die Seiten Ø, 1, 2, 3 und das 1/8 K RAM in Seite 1A. Den unteren Teil nimmt dagegen einen Teil des Monitorprogramms im EPROM ein. Dieses Monitorprogramm ist identisch mit **Bild 1.**
Um das Interruptkonzept des Junior-Computers zu veranschaulichen, sind nur die wichtigsten Speicherzellen herausgenommen. Auf der rechten Seite von **Bild 2** sehen wir diverse Möglichkeiten, um in den Monitor zu springen oder herauszukommen. Unter Monitor verstehen wir dabei den Teil im EPROM, mit dem bereits in Buch 1 gearbeitet wird. Außerdem die sechs Speicherzellen für den NMI-, Reset-, und IRQ-Vektor mit den Adressen 1FFA . . . 1FFF. In den folgenden Punkten ist alles wissenswerte über diese Vektoren erklärt:

1. Drücken der RST-Taste:
Die CPU holt aus den EPROM-Zellen 1FFC und 1FFD den Reset-Vektor. Bei ① verzweigt sie anschließend zum Label RESET in **Bild 1.** Es ist jedoch auch möglich, vom Programm des Programmierers über eine JMP-Instruktion zu diesem Label zu springen.

2. Auslösen eines NMI:
Ein NMI kann beim Junior-Computer auf verschiedene Arten ausgelöst werden:
* Durch Drücken der ST-Taste.
* Durch ein extern angeschlossenes Gerät über die NMI-Leitung auf dem Controlbus. Diese Leitung ist auch auf den Expansion-Konnektor herausgeführt.
* Durch eine negative Flanke am Ausgang von N5, wenn der Junior-Computer in Step by Step Mode arbeitet.

Sobald eine negative Flanke am NMI-Pin der CPU ankommt, führt der Computer eine NMI-Sequenz aus. **Bild 2** veranschaulicht die einzelnen Programmschritte:
– Nach dem Eintreffen der negativen Pulsflanke am NMI-Pin führt die CPU die momentan behandelte Instruktion aus und kümmert sich anschließend um den Interrupt. Vor dem Holen des NMI-Vektors

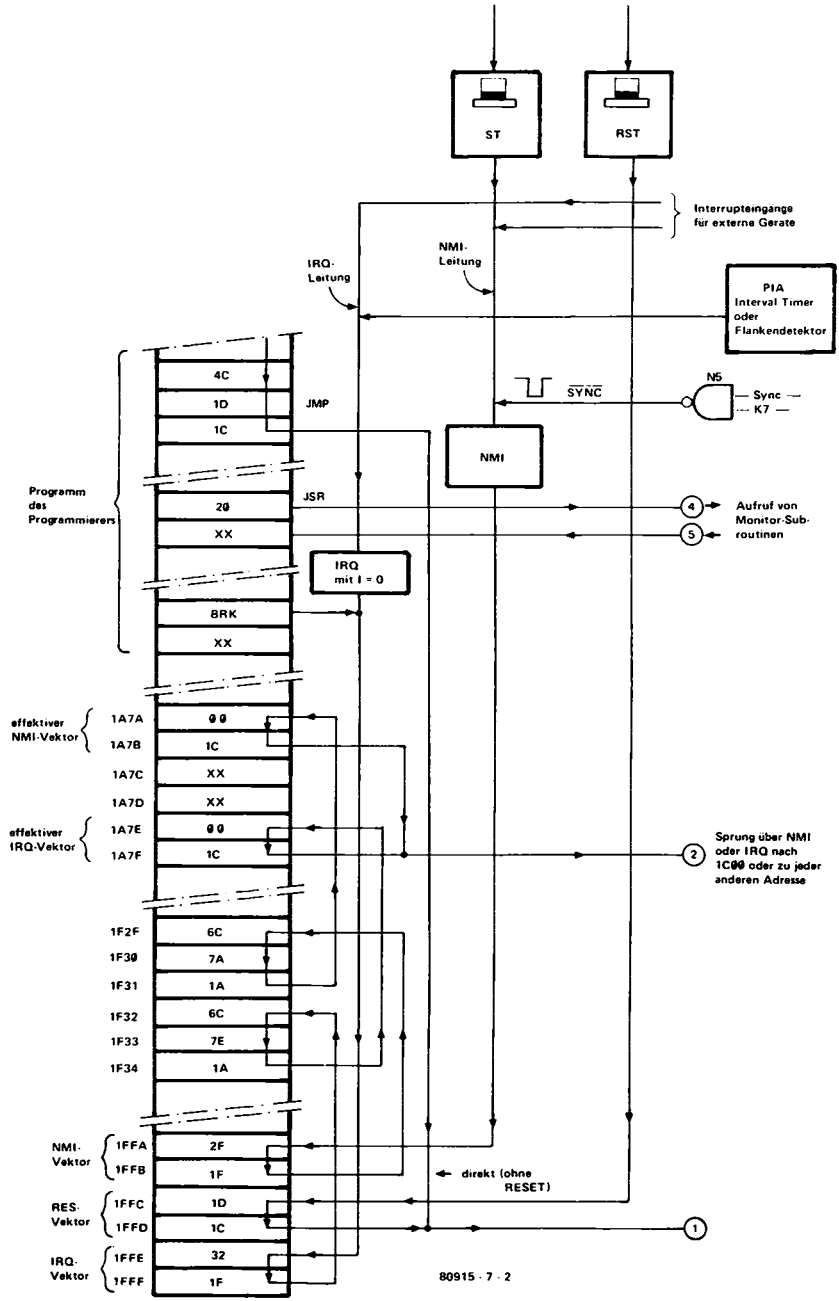


Bild 2. Wie kommt man in und aus dem Monitor? Alle Möglichkeiten sind hier in einer Grafik zusammengestellt. Da die Interrupt-Vektoren auf indirekte Programmsprünge zeigen, kann diese der Programmierer von Hand oder per Programm modifizieren.

rettet die CPU der Reihenfolge nach PCH, PCL und das P-Register auf den Stack. Die I-Flag ist gesetzt.

– Aus den Speicherzellen 1FFA und 1FFB am Ende des EPROMs holt der Prozessor den NMI-Vektor. Dieser Vektor zeigt beim Junior-Computer auf die Adresse 1F2F. An dieser Adresse steht im Monitorprogramm ein indirekter Sprungbefehl: JMP – (IND) = JMP – (1A7A) = 6C 7A 1A. Wie aus **Buch 1** bekannt ist, führt ein indirekter Sprung die CPU zu zwei Speicherzellen, in denen die effektive Sprungadresse abgelegt ist. Diese beiden Speicherzellen liegen beim Junior-Computer in Seite 1A und haben die Adressen 1A7A und 1A7B. Der Inhalt dieser Speicherzellen ist also die Adresse, an der der Computer nach der Auslösung eines NMI zu arbeiten anfängt.

In **Bild 2** zeigt der effektive NMI-Vektor auf die Adresse 1CØØ. Ist der Junior-Computer bei ② angekommen, gelangt er zum Label SAVE in **Bild 1**. Da die Speicherzellen mit den Adressen 1A7A und 1A7B RAM-Zellen sind, kann der Interrupt-Vektor auf beliebige Adressen gesetzt werden.

3. Auslösen eines IRQ:

Damit die CPU einen IRQ auslösen kann, muß die I-Flag im Prozessor-Status-Register zurückgesetzt sein (CLI). Ein IRQ kann beim Junior-Computer auf verschiedene Arten ausgelöst werden:

* Durch ein extern angeschlossenes Gerät über die IRQ-Leitung auf dem Controlbus. Diese Leitung ist auch auf den Expansion-Konnektor herausgeführt.
* Durch einen Timer-IRQ (siehe Kapitel 6)
* Durch einen PA7-IRQ (siehe Kapitel 6)

Ist die IRQ-Leitung für eine Zeitdauer von mehreren Mikrosekunden log. Ø, führt der Computer immer eine IRQ-Sequenz aus. Der Programmierer muß darauf achten, daß die IRQ-Leitung vor der Rückkehr aus der Interrupt-Routine zurückgesetzt ist (log. 1), damit der Prozessor denselben Interrupt Request nicht nochmals ausführt (siehe Kapitel 6). **Bild 2** veranschaulicht die einzelnen Programmschritte bei einem IRQ:

– Nachdem der IRQ-Pin der CPU log. Ø geworden ist, führt der Prozessor die momentan behandelte Instruktion zu Ende und kümmert sich anschließend um den IRQ. Vor dem Holen des IRQ-Vektors rettet die CPU der Reihenfolge nach PCH, PCL und das P-Register auf den Stack. Die I-Flag ist gesetzt.

– Aus den Speicherzellen 1FFE und 1FFF holt der Prozessor den IRQ-Vektor. Dieser Vektor zeigt beim Junior-Computer auf die Adresse 1F32. An dieser Adresse steht im Monitorprogramm ein indirekter Sprung-Befehl: JMP – (IND) = JMP – (1A7E) = 6C 7E 1A. Wie aus **Buch 1** bekannt ist, führt ein indirekter Sprung die CPU zu zwei Speicherzellen, in denen die effektive Sprungadresse abgelegt ist. Diese beiden Speicherzellen für den effektiven IRQ-Vektor liegen beim Junior-Computer in Seite 1A und haben die Adressen 1A7E und 1A7F. Der Inhalt dieser beiden Speicherzellen ist also die Adresse mit welcher der Computer nach der Auslösung eines IRQ zu arbeiten anfängt.

In **Bild 2** zeigt der IRQ-Vektor ebenfalls auf die Adresse 1CØØ. Ist der Junior-Computer wieder bei ② angekommen, gelangt er zum Label SAVE

in **Bild 1.** Da die Speicherzellen mit den Adressen 1A7E und 1A7F RAM-Zellen sind, kann der IRQ-Vektor auf beliebige Adressen gesetzt werden.

4. Das BRK-Kommando

Obwohl in **Buch 1** bei verschiedenen Programmen das BRK-Kommando Verwendung findet, ist es noch nicht im Detail beschrieben. Deshalb folgt die Beschreibung des Kommandos an dieser Stelle:

- Der Prozessor arbeitet in einem Programm, das der Programmierer in den Computer eingegeben hat. Am Ende dieses Programms trifft die CPU auf ein BRK-Kommando (OP-Code ist ØØ). Das BRK-Kommando arbeitet mit dem IRQ-Vektor zusammen; das heißt, die Programmverzweigung läuft wie bei einer IRQ-Sequenz ab (siehe Punkt 3). Vor dem Holen des IRQ-Vektors rettet die CPU der Reihenfolge nach PCH, PCL+2 und das P-Register auf den Stack. Die B-Flag im P-Register ist gesetzt. Steht beispielsweise das BRK-Kommando an der Adresse Ø22A, dann wird Ø22C als Program Counter auf den Stack gerettet — **also der falsche Program Counter!**
- Aus den Speicherzellen 1FFE und 1FFF holt der Prozessor den IRQ-Vektor. Dieser Vektor zeigt beim Junior-Computer auf die Adresse 1F32. An dieser Adresse steht im Monitorprogramm ein indirekter Sprung-Befehl. Dieser indirekte Sprung führt die CPU zu zwei Speicherzellen in Seite 1A, in denen die effektive Sprungadresse abgelegt ist. Die beiden Speicherzellen, welche die effektive Sprungadresse beinhalten, haben die Adressen 1A7E und 1A7F. In **Bild 2** fängt die CPU nach dem BRK-Kommando an der Adresse 1CØØ zu arbeiten an. Da der IRQ-Vektor in RAM-Zellen abgelegt ist, können mit dem BRK-Kommando beliebige Adressen angesprungen werden.

5) BRK-Flag contra I-Flag und einiges mehr:

Führt der Prozessor einen IRQ aus, setzt er automatisch die I-Flag im P-Register. Führt der Prozessor einen BRK aus, setzt er automatisch die B-Flag im P-Register. Wozu sind diese Flags erforderlich? Die Antwort ist denkbar einfach! Sowohl das BRK-Kommando wie der Interrupt Request arbeiten mit einem Vektor: den IRQ-Vektor. Arbeiten beispielsweise mehrere Drucker an einem Zentral-Computer und fordern diese über die beiden Interrupt-Leitungen eine Bedienung vom Computer an, dann sind mehrere Interrupts ineinander verschachtelt. Ist dem Intervall-Timer noch erlaubt einen IRQ auszulösen, sind schnell fünf oder sechs Interrupt-Routinen ineinander verschachtelt. Der Junior-Computer hat 16 I/O-Leitungen. Theoretisch könnten an diese Leitungen gleichzeitig 16 Drucker oder andere Geräte angeschlossen werden.

Durch das Setzen der I-Flag weiß der Computer, daß ein Interrupt (NMI oder IRQ) über einen der Interrupteingänge der CPU ausgelöst ist. Das gesetzte B-Flag signalisiert dem Computer, daß nicht ein Signal an den Interrupteingängen die Interruptsequenz ausgelöst hat, sondern das BRK-Kommando. Das BRK-Kommando führt aber über den IRQ-Vektor zur Programmverzweigung. Somit ist eindeutig definiert, ob ein Interrupt Request oder ein BRK-Kommando den IRQ-Vektor benutzt hat. Warum das P-Register die B-Flag und die I-Flag enthält, dürfte jetzt klar geworden sein. Übrigens: Die B-Flag bleibt nach der Ausführung des BRK-Kom-

mandos gesetzt. Trifft irgendwann danach an der CPU ein Interrupt-Signal ein, wird beim Setzen der I-Flag die B-Flag zurückgesetzt.

Verzweigen über das BRK-Kommando in eine Interrupt-Routine, die mit RTI abgeschlossen ist

Bei der Fehlersuche wünscht der Programmierer häufig mit dem BRK-Kommando in eine Interrupt-Routine zu verzweigen, die mit dem RTI-Befehl abgeschlossen ist. Das ist mit dem Sprung in eine Subroutine vergleichbar, nur daß die Sprung-Instruktion durch das BRK-Kommando nicht mehr drei, sondern ein Byte beträgt. Zwischen dem BRK-Kommando und dem JSR-Befehl lassen sich dann folgende Analogien feststellen:

```
•
•
•
JSR-XXXX        springe absolut
•               in eine Subroutine
•               mit der Adresse XXXX
•
•
Startadresse der Subroutine
Verweile in der Subroutine
bis zum RTS-Befehl
•
•
•
RTS             kehre ins Hauptprogramm
                zurück
```

Ein Sprung in eine Interrupt-Routine mit dem BRK-Kommando läuft über den IRQ-Vektor. Dabei wird vor der Verzweigung in die Interrupt-Routine das P-Register und der **falsche Program Counter** auf den Stack gerettet. Vor dem RTI-Befehl ist deshalb eine Korrektur des Program Counters nötig:

```
•
•
•
BRK     Springe vom Hauptprogramm in eine Interruptroutine, die an der Adresse
•       beginnt, auf die der IRQ-Vektor zeigt. Rette zuvor der Reihenfolge nach
•       PCH, PCL+2 und das P-Register auf den Stack (automatisch).
•
Startadresse der Interrupt-
Routine. Verweile in der In-
terrupt-Routine bis zum RTI-
Befehl
•
•
•
Die Interrupt-Routine ist ab-
```

gearbeitet, es soll ein Rücksprung
ins Hauptprogramm erfolgen.
Dazu muß PC korrigiert werden:

```
•
•
•
PLA          hole altes P-Register vom Stack
STA-MEM      rette das P-Register
PLA          hole falschen PCL vom Stack
STA-MEM+1    rette den falschen Program Counter
PLA          hole falschen PCH vom Stack
STA-MEM+2    rette den falschen Program Counter
SEC
LDA-MEM+1    hole den falschen PCL
SBC # Ø1     korrigiere PCL
STA-MEM+1    rette den korrigierten PCL
LDA-MEM+2    hole den falschen PCH
SBC # ØØ     korrigiere PCH
PHA          lege den korrigierten PCH auf den Stack
LDA-MEM+1    hole den korrigierten PCL
PHA          lege den korrigierten PCL auf den Stack
LDA-MEM      hole das alte P-Register
PHA          Lege das alte P-Register auf den Stack
             (Achtung! Diese Programmsequenz kann
             nicht im Step by Step Mode durchlaufen
             werden)
RTI          kehre mit dem korrigierten Program Counter
             ins Hauptprogramm zurück
```

Eine interessante Bemerkung:
Beim Austesten eines Basic-Compilers mußte öfters über das BRK-Kommando in eine Interrupt-Routine verzweigt werden, die ein RTI-Befehl abschließt. Diese Interrupt-Routine hat in Abhängigkeit verschiedener Operationen die Flags im P-Register beeinflußt. Nach der Rückkehr von der Interrupt-Routine in das Hauptprogramm war also **nicht** das alte P-Register von Bedeutung, das der Prozessor vor der Verzweigung in die Interrupt-Routine auf den Stack rettete. Benötigt der Programmierer nach der Rückkehr von der Interrupt-Routine den Zustand des P-Registers vor dem RTI-Befehl, dann ist folgende Programmsequenz auszuführen:

```
•
•
•
BRK          Springe vom Hauptprogramm in eine Interrupt-Routine, die an der Adresse
•            beginnt, auf die der IRQ-Vektor zeigt. Rette zuvor der Reihenfolge nach
•            PCH, PCL+2 und das P-Register auf den Stack (automatisch).
•
•
Startadresse der Interrupt-
```

Routine. Verweile in der Interrupt-Routine bis zum RTI-Befehl.

•
•
•

Die Interrupt-Routine ist abgearbeitet und es soll ein Rücksprung ins Hauptprogramm erfolgen. Im Hauptprogramm benötigt der Programmierer das von der Interrupt-Routine beeinflußte P-Register. Auch PC muß korrigiert werden:

•
•
•

PHP	rette das von der Interrupt-Routine beeinflußte P-Register
PLA	in den Speicher
STA-MEM	
PLA	hole altes P-Register vom Stack
PLA	hole falschen PCL vom Stack
STA-MEM+1	rette den falschen Program Counter
PLA	hole falschen PCH vom Stack
STA-MEM+2	rette den falschen Program Counter
SEC	
LDA-MEM+1	hole den falschen PCL
SBC # Ø1	korrigiere PCL
STA-MEM+1	rette den korrigierten PCL
LDA-MEM+2	hole den falschen PCH
SBC # ØØ	korrigiere PCH
PHA	lege den korrigierten PCH auf den Stack
LDA-MEM+1	hole den korrigierten PCL
PHA	lege den korrigierten PCL auf den Stack
LDA-MEM	hole das alte P-Register
PHA	lege das alte P-Register auf den Stack (Achtung! Diese Programmsequenz kann **nicht** in Step by Step Mode durchlaufen werden)
RTI	kehre mit dem korrigierten Program Counter ins Hauptprogramm zurück

Im Hauptprogramm steht dem Programmierer das während der Interrupt-Routine beeinflußte P-Register zur Verfügung.

Bild 2 hat neben den NMI und IRQ-Eingängen noch andere. An der Adresse 1CØØ beginnt der Monitor. Normalerweise wird dieses Programm durch Drücken der GO-Taste verlassen. Der Ausgang ist ③ (vergleiche mit **Bild 1**). Über den Eingang ④ kann der Programmierer bestimmte Monitor-Subroutinen aufrufen. Am Ende dieser Subroutinen (RTS) verläßt der Prozessor wieder den Monitor und springt zurück zum Programm des Programmierers.
Somit sind alle Ein- und Ausgänge des Monitorprogramms besprochen.

Auch ist uns das globale Flußdiagramm (Bild 1) vertraut. Die genaue Beschreibung des BRK-Kommandos und der Interrupts hat einen besonderen Grund: die Programme werden immer größer und benötigen mehr und mehr Zeit. Der Zeitaufwand für das Testen von eingegebenen Programmen und die Verarbeitungszeit dieser Programme läßt sich jetzt auf ein Minimum reduzieren, da das BRK-Kommando und das Interruptkonzept der CPU 6502 bekannt sind. Und jetzt besprechen wir das Monitorprogramm. Instruktion für Instruktion!

Der Monitor

In Bild 1 sind drei wichtige Label des Monitors eingezeichnet: RESET, SAVE und START. Die Programmabläufe hinter diesen Labels wollen wir jetzt Instruktion für Instruktion kennenlernen. Durch das Verfolgen der Flußdiagramme des Monitors hat der Anfänger die Möglichkeit, das Gehirn seines Junior-Computers genau kennen zu lernen. Wir besprechen den Monitor auch noch aus einem anderen Grunde sehr ausführlich: Die Junior-Computer-Bücher sind so geschrieben, daß man im Selbstunterricht das Programmieren in Maschinensprache lernen kann. Der Lernwillige Programmierer kann beim Verfolgen der Flußdiagramme erkennen, wie professionelle Programmierer Programme schreiben.

RESET

Damit sich der Junior-Computer über das Display zurückmelden und das Tastenfeld abfragen kann, muß er nach dem Drücken der RST-Taste in einen definierten Zustand gebracht werden. Wie wir aus Kapitel 6 wissen, muß dazu der periphere Interface Adapter programmiert sein. Auch interne CPU-Register, wie der Stack-Pointer und das Prozessor-Status-Register muß ein kleines Programm in einen definierten Zustand bringen. Nach dem Holen des Reset-Vektors aus den Speicherzellen mit den Adressen 1FFC und 1FFE kommt der Junior-Computer zum Label RESET in **Bild 3**. Dort definiert er zuerst die I/O von Port B. Die Portleitungen PB1 . . . PB4 werden dabei zu Ausgängen programmiert. Wie wir noch aus Buch 1 wissen, sind in Page Zero bestimmte Speicherzellen für das Retten der internen CPU-Register reserviert. Eine dieser Speicherzellen heißt PREG. In diese wird das Prozessor Status Register gerettet. Wird nach einer Restsequenz der Monitor zum ersten Male durch Drücken der GO-Taste verlassen (siehe Bild 1), muß der Inhalt der Speicherzelle PREG einen definierten Zustand haben. Warum das so ist, besprechen wir anschließend. In PREG wird Ø4 geladen. Da in **Bild 3** auch der Aufbau des Prozessor Status Register (P-Register) gezeichnet ist, kann man leicht erkennen, welche Flags gesetzt oder zurückgesetzt sind: Die I-Flag ist gesetzt, alle anderen Flags sind zurüchgesetzt. Da auch die D-Flag zurückgesetzt ist, arbeitet der Junior-Computer von jetzt ab als binär denkende Maschine. Durch das Setzen der I-Flag kann die CPU keinen IRQ bedienen. Ein an den Controlbus angeschlossenes Gerät, zum Beispiel ein Drucker, kann also nicht irrtümlicherweise einen IRQ auslösen. Dann schreibt der Prozessor die Zahl in die Speicherzelle, die ungleich null ist. Ist der Inhalt von MODE ungleich null, arbeitet der Junior-Computer in AD-MODE, ist der

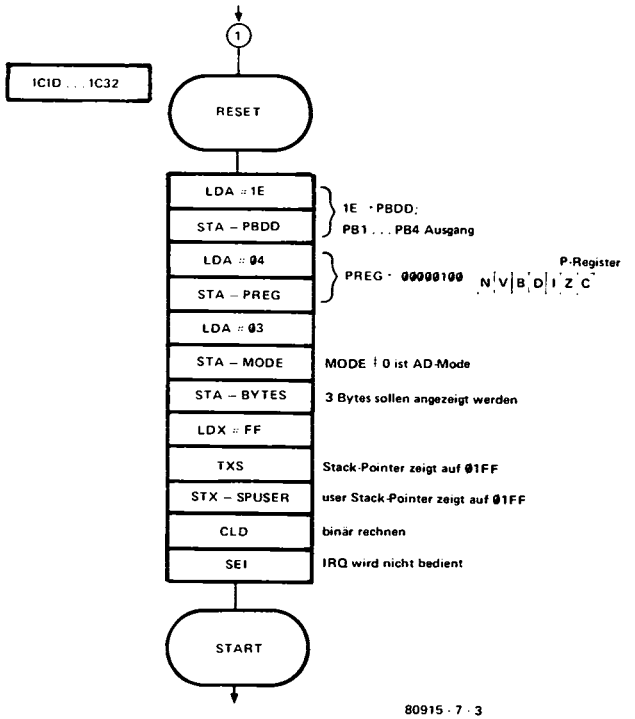


Bild 3. Wird der Monitor mit der RST-Taste gestartet, dann durchläuft die CPU die RESET-Routine. In dieser Routine definiert der Computer die I/O zur Steuerung des Displays und des Tastenfeldes. Auch diverse interne CPU-Register werden in einen definierten Zustand gebracht: Der Stack Pointer und das Prozessor Status Register.

Inhalt von MODE gleich null, arbeitet er in DA-MODE. Ebenfalls schreibt auch noch der Computer in die Speicherzelle BYTES die Zahl Ø3. Das bedeutet, es sollen drei BYTES auf den sechs Displays angezeigt werden: zwei Adreßbytes und ein Datenbyte. Erinnern wir uns an das Eintippen der Programme mit dem Editor (Kapitel 5). Das Display hat dabei eine variable Länge in Abhängigkeit von der angezeigten Instruktion.
Ist der Inhalt von BYTES Ø1, leuchten nur die beiden linken Displays auf. Ist der Inhalt dieser Speicherzelle Ø2, leuchten die beiden linken und die beiden mittleren Displays auf. Dagegen leuchten alle Displays auf, wenn der Inhalt von BYTES Ø3 ist.
Die CPU richtet außerdem noch den Stack-Pointer auf die Adresse Ø1FF aus. Da der Stack-Pointer bei der CPU 6502 immer in Seite 1 ist, muß nur das niederwertige Pointerbyte gesetzt werden, das hochwertige hat immer den Wert Ø1! Die Instruktionen sind: LDX # FF, TXS. Da der Stackpointer auch in Page Zero gerettet wird, muß auch in die Speicherzelle SPUSER das niederwertige Stack-Pointerbyte geladen sein.
Die beiden folgenden Instruktionen sorgen dafür, daß der Junior-Com-

puter im Monitorprogramm binär arbeitet und keinen IRQ akzeptiert. Beim ersten Verlassen des Monitors mit der GO-Taste sind die I-Flag gesetzt und die D-Flag zurückgesetzt, da in diesem Fall der Inhalt der Speicherzelle PREG den tatsächlichen Zustand des P-Registers repräsentiert. Das Wie und Warum erklären wir im folgenden Abschnitt. Erreicht schließlich die CPU den Label START, ist die Resetsequenz beendet.

SAVE

Nehmen wir nochmals **Bild 1** zur Hand. Hinter dem Eingang ② befindet sich der Label SAVE. Die diesem Label folgende Programmsequenz rettet alle internen CPU-Register in Page Zero. Über Eingang ② sind wir in Step by Step Mode in den Monitor gesprungen, also durch einen NMI, ausgelöst durch die negative Flanke am Ausgang des Gatters N5 (Bild 2).

Bild 4a zeigt, wie die einzelnen internen CPU-Register in Page Zero gerettet werden. Den Label SAVE erreichen wir in Step by Step Mode durch einen NMI. Ein NMI legt bekanntlich PCH, PCL und das P-Register in der beschriebenen Reihenfolge auf dem Stack ab. Bild 4b zeigt den Zustand des Stacks, wenn die CPU beim Label SAVE angekommen ist: Vor dem NMI stand der Stack-Pointer auf der Adresse Ø1XX. An dieser Adresse legt die CPU PCH ab und dekrementiert den niederwertigen Teil des Stack-Pointers. Der Stack-Pointer zeigt jetzt auf die Adresse Ø1XX-1. Dort legt der Prozessor PCL ab und dekrementiert den Stack-Pointer abermals um eins. Dieser zeigt somit auf die Adresse Ø1XX-2. An dieser Adresse wird jetzt das P-Register auf dem Stack abgelegt und der Stack-Pointer auf die Adresse Ø1XX-3 gesetzt.

Nun ist der Prozessor beim Label SAVE angekommen. Zuerst wird der Inhalt des Accumulators in die Speicherzelle ACC (ØØF3) gerettet. Die Momentaufnahmen ① . . . ④ in Bild 4a,b veranschaulichen den Programmablauf.

Die folgende PLA-Instruktion holt das P-Register vom Stack und rettet dieses in die Speicherzelle PREG (Moment ②). Beim Label SAVE trifft der Prozessor wieder auf eine PLA-Instruktion. Durch diese holt er vom Stack das niederwertige Program Counter Byte. Das niederwertige Programm-Zählerbyte rettet die CPU in die Speicherzelle PCL in Page Zero und POINTL (Moment ③). Bei der folgenden PLA-Instruktion holt der Prozessor das hochwertige Program Counter Byte vom Stack. Dieses Byte wird ebenfalls in die Speicherzellen PCH und POINTH in Page Zero gerettet (Moment ④). Wir erinnern uns: vor dem Label SAVE zeigte der Stack-Pointer auf die Adresse Ø1XX. Nach dem Interrupt rettete die CPU PCH, PCL und das P-Register auf den Stack. War der Prozessor beim Label SAVE angekommen, dann zeigte der Stack-Pointer auf die Adresse Ø1XX-3. Zwischen dem Label SAVE und SAVEB führte der Prozessor dreimal hintereinander eine PLA-Instruktion aus. In diesem Moment (Moment ④) zeigt der Stack-Pointer wieder auf die Adresse Ø1XX und hat somit seinen ursprünglichen Zustand eingenommen.

Bis jetzt rettete der Prozessor mit Hilfe des Stacks als temporären Zwischenspeicher das P-Register, PCL, PCH und frischte durch das Schrei-

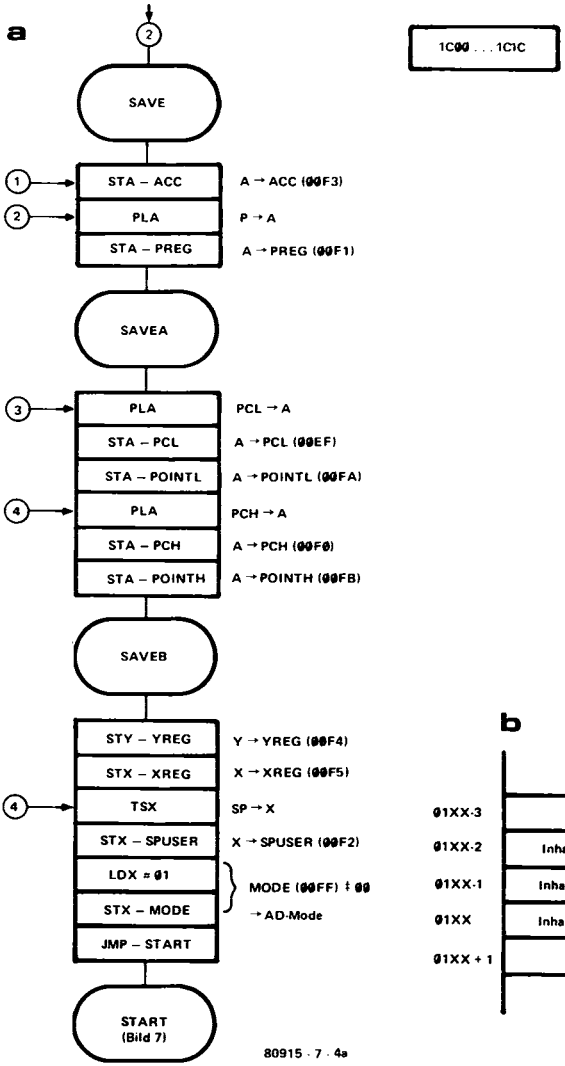


Bild 4a. So sieht das detaillierte Flußdiagramm der SAVE-Routine aus. Durch die SAVE-Routine ist es beim Junior-Computer möglich, ein Programm in Step by Step Mode zu durchlaufen. Alle internen CPU-Register werden dabei in bestimmte Speicherzellen in Page Zero kopiert. Normal wird die SAVE-Routine über einen NMI aufgerufen. Nach der Ausführung einer Instruktion in Step by Step Mode wird hinter dem Label SAVEA der Displaybuffer automatisch aufgefrischt.

Bild 4b. Der Stack Pointer ist in Step by Step Mode ein wichtiger temporärer Zwischenspeicher. Um auf Daten, die auf dem Stack abgelegt sind, Zugriff zu haben, manipuliert das Monitorprogramm den Stack Pointer.

ben des neuen Program Counters in die Speicherzellen POINTH, POINTL den Displaybuffer auf. Jetzt müssen noch die übrigen CPU-Register gerettet werden. Das erledigt der Prozessor ab dem Label SAVEB. Dabei rettet er das Y-Register in die Speicherzelle XREG. Auch der Stack-Pointer ist noch nicht gerettet. Die beiden Instruktionen TSX und STX-SPUSER retten den Stack-Pointer in die Speicherzelle SPUSER. Der Zustand des Stack-Pointers hat sich seit Moment ④ nicht mehr geändert. Das niederwertige Adreßbyte des Stack-Pointers , also XX, können wir in der Speicherzelle SPUSER nachschlagen. Somit sind alle internen CPU-Register gerettet. Desweiteren füllt der Prozessor noch eine nützliche Aufgabe: er schreibt eine Zahl, die ungleich null ist, in die Speicherzelle MODE. Das hat zur Folge, daß der Junior-Computer in Step by Step Mode nach der Ausführung einer Instruktion automatisch in AD-MODE geschaltet wird. Somit kann der Programmierer nicht irrtümlicherweise einen Maschinenbefehl durch falsche Daten überschreiben.
Jetzt ist der Prozessor beim Label START angekommen und kümmert sich um die Steuerung des Displays und die periodische Abfrage des Tastenfeldes. Durch Drücken der GO-Taste kann bekanntlich der Junior-Computer das Monitorprogramm verlassen (siehe Bild 1) und in das Programm des Programmierers verzweigen. Wie dieses Verzweigen Schritt für Schritt vor sich geht, besprechen wir im folgenden Abschnitt.

Das Verlassen des Monitors

Bekanntlich befindet sich der Junior-Computer im Monitorprogramm in einer Warteschleife. In dieser Programmschleife steuert er das Display und fragt periodisch das Tastenfeld ab. Drückt der Programmierer die GO-Taste, verläßt der Computer die Warteschleife und verzweigt in das Programm des Programmierers. Wir erinnern uns an Buch 1: Beim Drücken der GO-Taste fängt der Junior-Computer an der Adresse zu arbeiten an, die momentan auf dem Display angezeigt ist. Sobald der Junior-Computer das Monitorprogramm verlassen hat, kümmert er sich nicht mehr um Tastenfeld und Display, sondern ausschließlich um das Programm des Programmierers.
Bild 5a und Bild 5b geben darüber Auskunft, wie der Junior-Computer durch Drücken der GO-Taste das Monitor-Programm verläßt und an der auf dem Display angezeigten Adresse zu arbeiten beginnt. Auch dabei verwendet der Computer den Stack als temporären Zwischenspeicher. (Momente ① . . . ④).
Zuerst stellt die CPU den ursprünglichen Stack- Pointerzustand wieder her. Die Instruktionen sind: LDX-SPUSER, TXS. Der Stack-Pointer zeigt wieder auf die Adresse Ø1XX (Moment ①). Dann wird der Inhalt der Displaybuffer POINTH und POINTL auf den Stack gebracht. Nachdem POINTH auf dem Stack ist, zeigt der Stack-Pointer auf die Adresse Ø1XX-1 (Moment ②) und nach dem Transport von POINTL auf den Stack zeigt der Stack-Pointer auf Ø1XX-2 (Moment ③). Jetzt ist der Inhalt von POINTH, POINTL als neuer Programm-Zähler auf den Stack gerettet. Als nächstes wechselt der ursprüngliche Zustand des P-Registers auf den Stack. Die Instruktionen sind: LDA-PREG, PHA. Der Stack-Pointer zeigt jetzt auf die Adresse Ø1XX-3 (Moment ④). Nachdem der Programm-Zähler und das P-Register auf dem Stack sind, stellt die CPU den ursprünglichen Zu-

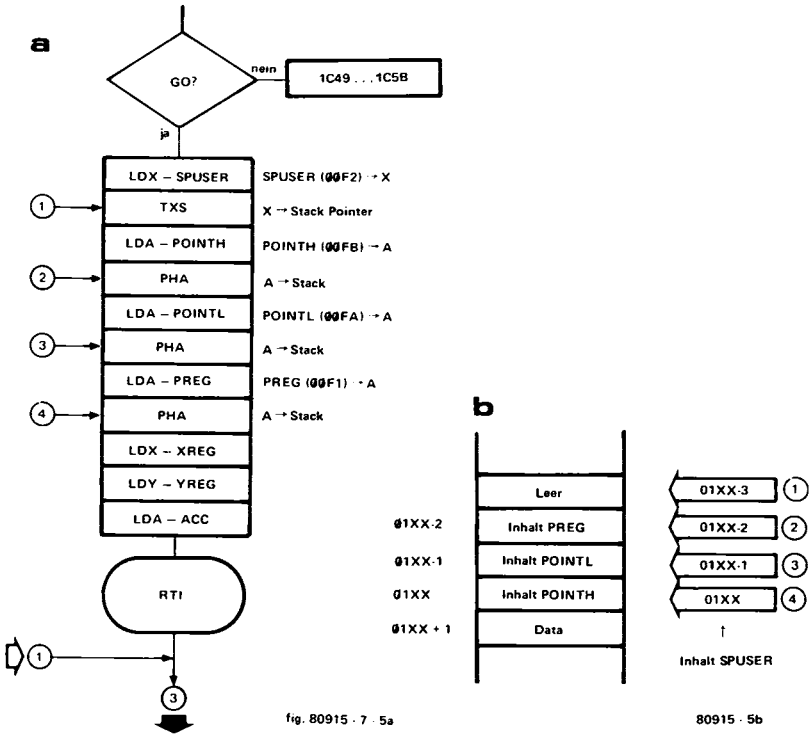


Bild 5a. Dieses detaillierte Flußdiagramm führt der Prozessor aus, wenn der Programmierer die GO-Taste drückt. Das Drücken dieser Taste ist eine der Möglichkeiten, um das Monitorprogramm zu verlassen und in ein Programm des Programmierers zu verzweigen. Dabei stimmen die Momentaufnahmen des Stack Pointers mit denen von Bild 4b überein.

Bild 5b. Zustand des Stacks und des Stack Pointers zu bestimmten Momenten, wenn der Prozessor die Instruktionen in Bild 5a ausführt.

stand des Accumulators sowie der beiden Index-Register X und Y wieder her. Somit haben alle internen CPU-Register einen definierten Zustand und der Junior-Computer kann ins Programm des Programmierers verzweigen. Diese Programmverzweigung geschieht mit dem Befehl RTI. Wir erinnern uns, daß bei einem RTI-Befehl der Reihe nach das P-Register, PCL und PCH vom Stack geholt werden. Dabei ist PCH und PCL der Inhalt der beiden Displaybuffer POINTH und POINTL. Somit beginnt der Prozessor an der auf dem Display angezeigten Adresse zu arbeiten und verläßt die Warteschleife im Monitorprogramm.
Bemerkung: Ist der Junior-Computer in Step by Step Mode, führt er beim Drücken der GO-Taste die Programmsequenz in **Bild 5** aus. Er verzweigt zu der auf dem Display abgebildeten Adresse. An dieser Adresse steht ein Maschinenbefehl. Diesen Maschinenbefehl führt der Computer aus. Beim Laden des OP-Codes wird ein NMI über den SYNC-Ausgang der CPU aus-

gelöst. Der Prozessor führt die momentane Instruktion zu Ende und kümmert sich dann um den NMI. Der NMI-Vektor zeigt in Step by Step Mode auf die Adresse 1CØØ. An dieser Adresse steht der Label SAVE. Die CPU führt jetzt die gezeigte Programmsequenz in **Bild 4** aus. Dabei werden wieder alle CPU-Register gerettet und der Junior-Computer in AD-Mode geschaltet. Nach dieser Programmsequenz ist der Prozessor beim Label START angekommen und kümmert sich in einer Warteschleife um Display und Keyboard. Auf dem Display erscheint der folgende Maschinenbefehl mit seiner Adresse. Der Junior-Computer wartet ab dem Label START auf ein abermaliges Drücken der GO-Taste, um den momentan angezeigten Befehl abzuarbeiten. Der soeben beschriebene Vorgang wiederholt sich bei jedem Drücken der GO-Taste.

So "hört" der Junior-Computer dem Programmierer zu

In diesem Abschnitt besprechen wir, wie der Junior-Computer auf eine gedrückte Taste reagiert. Voraussetzung ist, daß der Computer in der Warteschleife eine gedrückte Taste erkennt. Die Erkennung dieser Taste, das heißt die Berechnung des Tastenwertes, erfolgt in einer Subroutine. Am Ende dieser Subroutine steht der Tastenwert im Accumulator der CPU. Wie wir noch von Buch 1 wissen, sind den Tasten folgende Werte zugeordnet:

Ø:ØØ	5:Ø5	A:ØA	F:ØF	PC:14
1:Ø1	6:Ø6	B:ØB	AD:10	falsche oder illegale
2:Ø2	7:Ø7	C:ØC	DA:11	Taste: 15
3:Ø3	8:Ø8	D:ØD	+:12	
4:Ø4	9:Ø9	E:ØE	GO:13	

Bild 6 zeigt, wie der Computer eine gedrückte Taste identifiziert. Eine CMP-Instruktion gefolgt von einer BNE-Instruktion filtert die gedrückte Taste heraus. Die CPU führt eine Verzweigung aus, wenn die wirklich gedrückte Taste nicht gleich dem Vergleichstastenwert ist. Ist aber die gedrückte Taste gleich dem Vergleichstastenwert, kümmert sich der Prozessor um die gedrückte Taste.

Das Tastenfeld des Junior-Computers läßt sich in zwei verschiedene Tastenfelder aufteilen: in ein Tastenfeld für die Datentasten und in ein solches für die Befehlstasten. Was beim Drücken der Befehlstaste GO geschieht, ist bereits besprochen. Drückt man die AD-Taste, sie ist ebenfalls eine Befehlstaste, wird der Inhalt der Speicherzelle MODE ungleich null gemacht. Der Junior-Computer interpretiert jede gedrückte Datentaste als Adresse. Über die folgende BNE-Instruktion verzweigt das Programm via STEPA zum Label START.

Hat man beispielsweise die AD-Taste gedrückt, macht der Computer den Inhalt der Speicherzelle MODE gleich null. Der Junior-Computer interpretiert jetzt jede gedrückte Datentaste als Daten, die an der angezeigten Adresse abgelegt und anschließend im Display angezeigt werden sollen.

Wie bereits bekannt, erhöht der Computer beim Drücken der +Taste die angezeigte Adresse um eins und zeigt anschließend die Daten auf dem Display, die an dieser Adresse abgelegt sind. War beispielsweise der Inhalt von POINTL vor dem Drücken der +Taste FF, dann ist dieser nach dem Drücken ØØ. Dabei muß der Inhalt des Adreßbuffers POINTH korrigiert,

das heißt um eins inkrementiert werden. Diese Korrektur erfolgt über eine BNE-Instruktion. Danach verzweigt das Programm wieder über STEPA zurück zum Label START.
Beim Drücken der PC-Taste kopiert der Computer den letzten Stand des Programmzählers ins Adreßdisplay. Das läßt sich mit ein paar Instruktionen leicht erledigen. Dazu kopiert man PCH, PCL, PCH (abgelegt in Page Zero) in die Displaybuffer POINTH und POINTL. Anschließend verzweigt das Monitorprogramm wieder zum Label START. Die Funktionsweise der PC-Taste ist uns noch von Buch 1 bekannt: Beim Arbeiten in Step by Step Mode, kontrolliert man häufig den Zustand der internen CPU-Register nach einzelnen Programmschritten. Da diese internen CPU-Register in Page Zero abgelegt sind, mußte das Programm des Programmierers zeitweise verlassen werden. Ein Drücken der PC-Taste stellte dann den ursprünglichen Zustand des Displays wieder her. Ein abermaliges Drücken auf die GO-Taste leitet den nächsten Programmschritt wieder ein.
Die bisherige Beschreibung gilt nur unter der Voraussetzung, daß das Monitorprogramm alle Befehlstasten richtig erkannt hat. Da jede Taste einen festen Tastenwert besitzt, können nur Tastenwerte zwischen Ø . . . 14 vorkommen. Es ist jedoch auch möglich, daß der Junior-Computer infolge einer Netzstörung oder wegen Tastenprellen einen falschen Tastenwert berechnet. Ist dies trotz Software-Entprellung der Fall, ignoriert der Computer beim Label ILLKEY die Taste, indem er wieder über STEPA zum Label START springt.

Modifizieren von Daten

Drückt der Programmierer keine Befehlstaste, sondern eine Datentaste, verzweigt das Programm zum Label DATA. Ab diesem Label verarbeitet der Computer die Datentasten Ø . . . F. Zuerst kommt der Tastenwert in die Speicherzelle KEY und der Accumulator ist für andere Aufgaben frei. Anschließend geht der Inhalt der Speicherzelle MODE ins Y-Register. Das Y-Register kann jetzt null oder ungleich null sein. Ist beispielsweise das Y-Register null, dann ist die gedrückte Datentaste, deren Wert in der Speicherzelle KEY steht, als Dateneingabe in den Junior-Computer zu interpretieren. Ist aber das Y-Register ungleich null, weiß der Computer, daß der Programmierer eine neue Adresse eingeben möchte und ordnet die Datentaste in der Speicherzelle KEY dem Adreßdisplay zu. Eine BNE-Instruktion unterscheidet zwischen Adreß- und Dateneingabe: LDY-MODE, BNE-MODE, BNE-ADDRES.
War zuvor das Y-Register gleich null, handelt es sich um eine Dateneingabe und die Verzweigung zum Label ADDRES findet nicht statt. Der Junior-Computer schiebt von rechts nach links die gedrückte Taste in das Datendisplay. Das sind die beiden rechten Displays auf der Anzeigenplatine, denen der Displaybuffer INH zugeordnet ist. Die Instruktionen für das Linksschieben des Datendisplays laufen wie folgt ab:
Mit dem Adreß-Pointer POINTH, POINTL lädt die CPU die auf dem Display angezeigten Daten in den Accu. Dazu bedient sie sich der indirekten indizierten Adressierung, wobei das Y-Register gleich null ist. Der Tastenwert in der Speicherzelle KEY ist in diesem Moment noch nicht ins Display geschoben. Nehmen wir jetzt an, die soeben geladenen Daten bestehen

aus den Bits p . . . w und stehen wie folgt im Accu:

```
p q r s t u v w . . . Accuinhalt nach dem Laden
q r s t u v w Ø . . . Accuinhalt nach dem ersten ASL
r s t u v w Ø Ø . . . Accuinhalt nach dem zweiten ASL
s t u v w Ø Ø Ø . . . Accuinhalt nach dem dritten ASL
t u v w Ø Ø Ø Ø . . . Accuinhalt nach dem vierten ASL
```

Nach viermaligem Schieben ist das niederwertige Datennibble in das hochwertige gewechselt, wobei das hochwertige Datennibble verloren ging. Das niederwertige Datennibble ist wegen der viermaligen Schiebeoperation gleich null.
Der Inhalt der Bufferzelle KEY ist ØØØØXXXX, wobei die X der Wert der gedrückten Datentaste sind. Durch eine OR-Operation des Accus mit der Speicherzelle KEY erhalten wir folgendes Resultat:

```
t u v w Ø Ø Ø Ø  Accuinhalt vor ORA
Ø Ø Ø Ø X X X X  Inhalt der Speicherzelle KEY
t u v w X X X X  Resultat nach der ORA-Operation
```

Das hochwertige Datennibble bleibt dabei unverändert und das niederwertige Nibble ist gleich dem Wert der gedrückten Tasten. Somit ist die gedrückte Taste von rechts nach links in den Accu geschoben. Das so gewonnene Datenbyte speichert die CPU mit dem Adreß-Pointer POINTH, POINTL an der angezeigten Adresse ab. Dann erfolgt wieder ein Rücksprung über STEPA zum Label START.
Der Label STEPA sieht beim ersten Hinsehen merkwürdig aus. Er ist jedoch notwendig, weil hinter ihm Branch-Instruktionen stehen, die zum Label START führen. Diese Branch-Instruktionen führen über einen Speicherbereich von mehr als -128 Schritte. Deshalb muß über STEPA direkt zum Label START gesprungen werden.

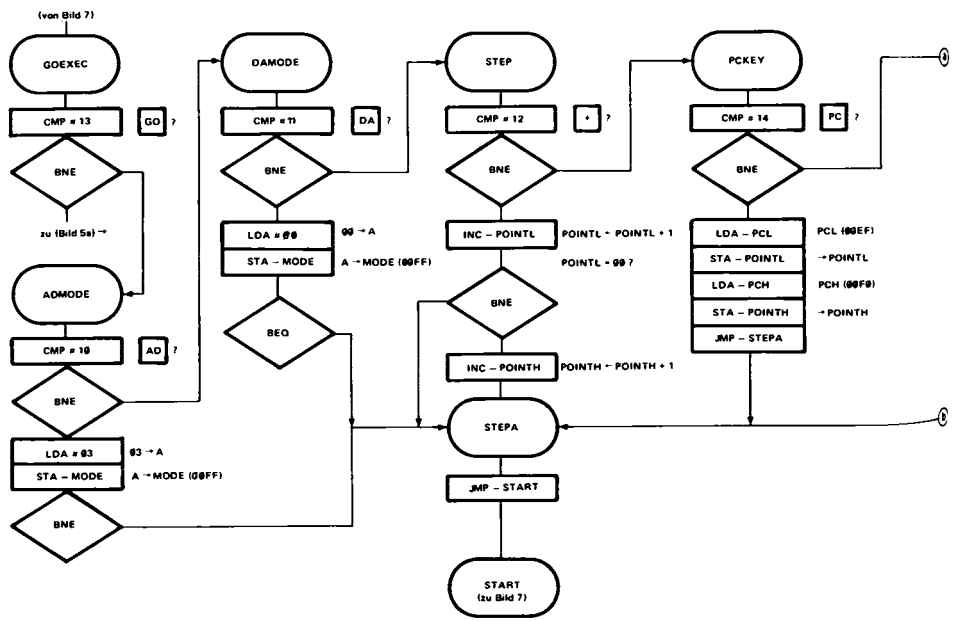
Modifizieren von Adressen

Wie der Programmierer im Speicher des Junior-Computers modifizieren kann, so kann er auch neue Adressen eingeben. Das heißt, der Inhalt der beiden Speicherzellen POINTH und POINTL, die den Adreß-Pointer bilden, kann man nach dem Drücken der AD-Taste verändern. Diese Änderung vollbringt das Monitorprogramm nach dem Label ADDRES. Da am Anfang das X-Register mit Ø4 geladen ist, durchläuft der Prozessor die Programmschleife ADLOOP-BNE-ADLOOP viermal. Viermal wird dabei der Inhalt von POINTL nach links geschoben und der Inhalt von POINTH nach links rotiert.
Zur Erinnerung: Bei ASL wird in bØ eine Null geschoben und die C-Flag ist b7 gleichgesetzt. Dagegen wird bei ROL die C-Flag in bØ geschoben und b7 der C-Flag gleichgesetzt.
Mit folgenden Vereinbarungen lassen sich die Schiebe- und Rotiere- Instruktionen leicht verstehen:
Das hochwertige Nibble von POINTH ist h i j k,
das niederwertige Nibble von POINTH ist l m n o,
das hochwertige Nibble von POINTL ist p q r s und
das niederwertige Nibble von POINTL ist t u v w.
Bei der Ausgangssituation kann die C-Flag gesetzt oder zurückgesetzt

sein. Die Ausgangssituation ist die Adresse, die momentan auf dem Display angezeigt wird. Die modifizierte Adresse ist die Adresse, die auf dem Display sichtbar ist, wenn der Programmierer in AD-Mode eine der Datentasten Ø . . . F drückt. Diese modifizierte Adresse erzeugt der Computer wie folgt:

POINTH	C	POINTL		
h i j k l m n o	x	p q r s t u v w	. . . Beginn	
h i j k l m n o	p	q r s t u v w Ø	. . . nach ASL-POINTL	X = Ø4
i j k l m n o p	h	q r s t u v w Ø	. . . nach ROL-POINTH	
i j k l m n o p	q	r s t u v w Ø Ø	. . . nach ASL-POINTL	X = Ø3
j k l m n o p q	i	r s t u v w Ø Ø	. . . nach ROL-POINTH	
j k l m n o p q	r	s t u v w Ø Ø Ø	. . . nach ASL-POINTL	X = Ø2
k l m n o p q r	j	s t u v w Ø Ø Ø	. . . nach ROL-POINTH	
k l m n o p q r	s	t u v w Ø Ø Ø Ø	. . . nach ASL-POINTL	X = Ø1
l m n o p q r s	k	t u v w Ø Ø Ø Ø	. . . nach ROL-POINTH	

Jetzt ist der Inhalt der beiden Displaybuffer POINTH und POINTL um vier Bits nach links verschoben. Das hochwertige Nibble von POINTL steht nun als niederwertiges Nibble von POINTH zur Verfügung, während das hochwertige Nibble von POINTH verloren ging. Bei der Schiebe- Rotieroperation fungiert die C-Flag als Zwischenstation.



80915 · 7 · 6

Der Inhalt von POINTL kommt jetzt in den Accu und es findet eine OR-Operation mit dem Inhalt der Speicherzelle KEY statt:

```
t u v w 0 0 0 0  Accuinhalt vor ORAZ-KEY
0 0 0 0 X X X X  Inhalt von KEY
t u v w X X X X  Resultat nach ORAZ-KEY
```

Das so gewonnene Resultat speichert der Prozessor in der Speicherzelle POINTL ab und frischt somit den Displaybuffer für die Adreßanzeige auf. Das niederwertige POINTL-Nibble ist damit der Tastenwert der gedrückten Adreßtaste. Über STEPA verzweigt das Programm zurück zum Label START und der Junior-Computer zeigt die modifizierte Adresse mit den dazugehörenden Daten im Display an. Eine numerische Taste stellt also immer ein Datennibble dar, das entweder zum Adreß- oder Datendisplay transportiert wird. Dabei sieht der Junior-Computer in der Speicherzelle MODE nach und stellt so fest, ob er die gedrückte Datentaste als Adresse oder als normale Daten, die der Programmierer in den Computer eingibt, interpretieren soll.
Das Hauptprogramm des Monitors ist somit besprochen. **Bild 1** am Anfang dieses Kapitels ist das Blockdiagramm. während **Bild 6** das detaillierte Flußdiagramm zeigt. Aus dem detaillierten Flußdiagramm geht hervor, wie der Junior-Computer eine gedrückte Datentaste von einer gedrückten Befehlstaste unterscheidet und wie er die einzelnen Befehlstasten in Befehle umsetzt. Damit ist also klar, wie der Junior-Computer intern Daten und Befehle, die der Programmierer eingibt, verarbeitet. Der nächste Abschnitt

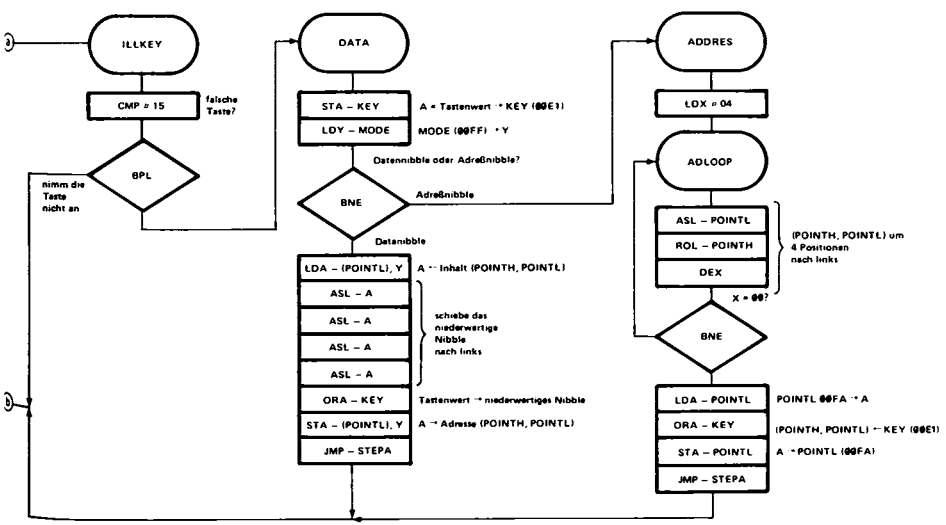


Bild 6. Das Bild zeigt, wie die Kommando-Tasten AD, DA, + und PC herausgefiltert werden. Dabei gehört zu jeder Tastenfunktion eine Tasten-Routine, die der Prozessor nach der Erkennung einer Kommando-Taste ausführt. Hinter dem Label DATA werden die Daten-Tasten ins Daten-Display geschoben, während hinter dem Label ADDRESS die Daten-Tasten ins Adreß-Display kommen.

beschreibt, wie sich der Computer über das Display zurückmeldet. Erst dann kann zwischen Programmierer und Computer ein Dialog stattfinden!

So "antwortet" der Junior-Computer dem Programmierer

Nehmen wir nochmals **Bild 1** zur Hand! Direkt hinter dem Label START bedient der Junior-Computer das Keyboard und das Display. Das heißt, er berechnet den Wert irgendeiner gedrückten Taste und zeigt diese Taste auf dem Display an, falls man eine der Datentasten Ø . . . F betätigt hat. Durch das Anzeigen einer gedrückten Datentaste auf dem Display antwortet der Computer dem Programmierer.

Wie bekannt, erreicht der Computer den Label START auf viele Arten:

* Nach dem Abhandeln einer gedrückten Keyboard-Taste verzweigt das Monitorprogramm zum Label START (siehe **Bild 6**).
* Nach der Rückkehr ins Monitorprogramm über den Label SAVE springt der Computer zum Label START. Von dieser Möglichkeit machten wir Gebrauch, wenn der Junior-Computer in Step by Step Mode war oder am Ende eines Programms ein BRK-Kommando stand.
* Auch nach dem Drücken der RST-Taste verzweigt das Monitorprogramm nach der Programmierung des PIA zum Label START.

Nach dem Label START ist der Junior-Computer immer in einer Warteschleife, die meistens durch das Drücken der GO-Taste verlassen wird. Was sich im Monitorprogramm hinter diesem Label ereignet, wollen wir nun Instruktion für Instruktion besprechen.

Bild 7 zeigt die ersten Schritte hinter dem Label START. Die Subroutine SCAND (wird später genau besprochen) sorgt für das Anzeigen der drei Displaybuffers POINTH, POINTL und INH und detektiert, ob irgendeine Taste gedrückt ist oder nicht. Sobald man während SCAND irgendeine Keyboard-Taste drückt, ist nach der Rückkehr ins Hauptprogramm der Accuinhalt ungleich null. Drückt man dagegen keine Taste, ist nach der Rückkehr aus dieser Subroutine der Accuinhalt gleich null. Somit läßt sich mit einer BNE-Instruktion abfragen, ob eine Taste gedrückt ist. Mit einer BEQ-Instruktion kann man dagegen feststellen, ob keine Taste betätigt ist.

Die Subroutine SCAND ist fast identisch mit der in Buch1 häufig verwendeten Subroutine SCANDS:

SCANDS: Zeigt nur den Inhalt des gesamten Displaybuffers POINTH, POINTL und INH auf dem Display an. Nach der Rückkehr von dieser Subroutine ins Hauptprogramm ist der Accu gleich/ungleich null, wenn eine Keyboard-Taste nicht gedrückt/gedrückt ist.

SCAND: Diese Subroutine hat dieselbe Funktion wie SCANDS. Allerdings werden in den Displaybuffer INH die Daten der Speicherzelle geladen, auf die der Adreßpointer POINTH, POINTL (Y = Ø) zeigt. Der Junior-Computer zeigt somit auf dem Display eine Adresse und die an dieser Adresse abgelegten Daten an.

Soviel fürs erste zur Subroutine SCAND. Nach dieser Subroutine treffen wir auf mehrere BEQ- bzw. BNE-Instruktionen. Welchen Sinn diese Instruktionen haben, zeigt folgendes praktisches Beispiel:

1. Nehmen wir an, der Junior-Computer sei in Step by Step Mode. Auf

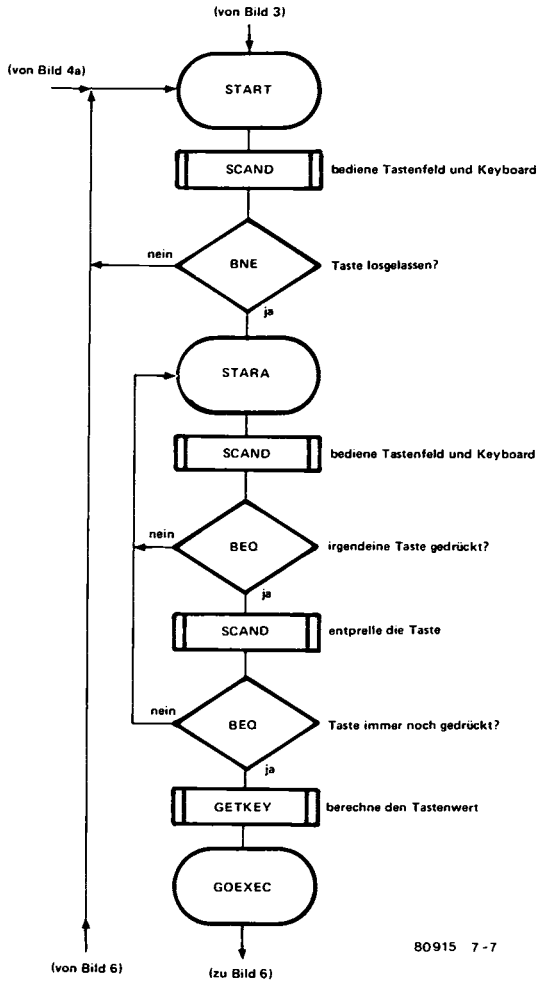


Bild 7. Dieser Teil des Monitor Programms transportiert den Inhalt der Displaybuffer POINTH, POINTL und INH zum Display des Junior-Computers. Dabei werden die anzuzeigenden Daten in den 7-Segment-Code umgesetzt. Anschließend kümmert sich der Prozessor um die Abfrage des Tastenfeldes. Ist eine gedrückte Taste identifiziert, wird der Tastenwert in der Subroutine GETKEY berechnet.

dem Display ist ein Maschinenbefehl an einer bestimmten Adresse sichtbar, den der Computer abarbeiten soll.

2. Der Programmierer drückt die GO-Taste. Der Junior-Computer arbeitet den auf dem Display angezeigten Befehl ab. Während er den Maschinenbefehl in die CPU lädt, wird über den SYNC-Ausgang und Gatter N5 **(Bild 2)** ein NMI ausgelöst. Der Computer arbeitet den soeben angezeigten

Befehl zu Ende und gelangt über den NMI-Vektor zum Label SAVE (siehe Blockdiagramm Bild 1 in diesem Kapitel). Dort angekommen rettet er alle internen CPU-Register in reservierte Speicherzellen in Page Zero und frischt den Displaybuffer POINTH, POINTL für die Adreßanzeige auf dem Display auf. Der Adreß-Pointer POINTH, POINTL zeigt somit auf die Adresse, an welcher der folgende Maschinenbefehl steht. Jetzt ist der Junior-Computer wieder beim Label START angekommen.

3. Was soll der Junior-Computer jetzt tun? Die Antwort darauf ist klar: Er soll den folgenden Maschinenbefehl mit der dazugehörenden Adresse auf dem Display anzeigen. Der Displaybuffer für die Adressen, also POINTH und POINTL, ist zuvor in der SAVE-Routine korrigiert worden. Die Adresse in diesen beiden Speicherzellen gehört zum folgenden Maschinenbefehl. Den Maschinenbefehl, der als nächster abgearbeitet werden soll, lädt der Computer in der Subroutine SCAND (Bild 7) in den Displaybuffer INH. Somit ist das gesamte Display wieder aufgefrischt. Die in den Punkten 1 . . . 3 beschriebenen Operationen bearbeitet der Computer in Bruchteilen von Sekunden! Bekanntlich detektiert die Subroutine SCAND, ob irgendeine Keyboard-Taste gedrückt ist. Das Drücken der GO-Taste hat den Computer veranlaßt, die zuvor beschriebene Programmsequenz auszuführen. Kehrt der Junior-Computer aus der Subroutine SCAND zurück, ist die GO-Taste noch immer gedrückt. Soll der Junior-Computer den nächsten Befehl abarbeiten? Nein, sicher nicht! Der Computer soll den folgenden Befehl samt Adresse auf dem Display anzeigen und warten, bis der Programmierer abermals die GO-Taste drückt. Bevor die GO-Taste abermals betätigt werden kann, muß diese zuerst losgelassen werden! Der Junior-Computer verweilt also nach der Ausführung des Befehls in der Warteschleife START-BNE-START solange, bis der Programmierer die GO-Taste losläßt.
4. Jetzt läßt der Programmierer die GO-Taste los. Der Computer kehrt aus der Subroutine SCAND mit einem Accuinhalt von null zurück (Kennzeichen, daß keine Taste gedrückt ist) und gelangt zum Label STARA. Dort findet wieder ein Sprung in die Subroutine SCAND statt. Adresse und Befehl werden auf dem Display angezeigt. Die GO-Taste, die in Step by Step Mode das Abarbeiten des momentan angezeigten Befehls zur Folge hat, ist noch immer nicht gedrückt. Deshalb ist der Accuinhalt nach der Rückkehr aus der Subroutine SCAND gleich null. Der Junior-Computer verweilt in der Warteschleife STARA-BEQ-STARA.
5. Der Programmierer möchte, daß der Computer wieder einen Maschinenbefehl abarbeitet. Dazu drückt er auf die GO-Taste. Der Junior-Computer kehrt in der Warteschleife STARA-BEQ-STARA aus der Subroutine SCAND mit einem Accuinhalt von ungleich null zurück (Kennzeichen, daß eine Taste gedrückt ist). Der Nachteil mechanischer Tasten ist das Tastenprellen. Deshalb wird die Subroutine SCAND abermals aufgerufen. Kehrt der Computer aus dieser Subroutine zurück, ist genügend Zeit verstrichen und das Tastenprellen abgeklungen. Die zweite BEQ-Instruktion stellt fest, ob die Taste immer noch gedrückt ist. Ist die Taste nicht mehr gedrückt, verzweigt das Programm zum

Label STARA, was ein Kennzeichen dafür ist, daß ein Störimpuls zum Verlassen der Warteschleife geführt hat. Der Computer wartet dann auf einen neuen Tastendruck.

6. Ist die Taste nach dem letzten Aufruf der Subroutine SCAND noch immer gedrückt, verläßt der Computer die Warteschleife und springt zur Subroutine GETKEY. In dieser Subroutine berechnet er den Wert der gedrückten Taste. Nach der Rückkehr aus der Subroutine GETKEY steht der Tastenwert der gedrückten Taste im Accu der CPU und wir sind wieder beim Label GOEXEC angekommen. Ab diesem Label werden die Daten- und Befehlstasten weiterverarbeitet (siehe Bild 6).

Zusammenfassung
Ab dem Label START bedient der Junior-Computer das Tastenfeld und das Display. Dazu finden die Subroutinen SCAND und GETKEY sowie BEQ- und BNE-Instruktionen Verwendung. In der Warteschleife STARA-BEQ-STARA verweilt der Computer solange, bis eine Taste gedrückt ist. Auch die Steuerung des Displays geschieht in dieser Warteschleife. Zwischen dem Label START und dem Label GOEXEC durchläuft der Computer mehrere Programmschleifen, die bestimmte Aufgaben haben:
***Die Schleife START-BNE-START:** In dieser Programmschleife bedient der Computer das Tastenfeld und das Display. Er verweilt in dieser Programmschleife solange, bis eine gedrückte Taste losgelassen ist.
***Die Schleife STARA-BEQ-STARA:** In dieser Programmschleife bedient der Computer das Tastenfeld und das Display. Er verweilt in dieser Programmschleife solange, bis eine Taste gedrückt wird. Bei der Dateneingabe verweilt der Junior-Computer die meiste Zeit in dieser Programmschleife.
***Die Schleife STARA-BEQ-BEQ-STARA:** In dieser Programmschleife werden die Tasten entprellt. Eine Verzweigung von der zweiten BEQ-Instruktion zum Label STARA findet nur dann statt, wenn nach der Subroutine SCAND das Tastenprellen noch nicht abgeklungen ist, oder ein Störimpuls einen unerlaubten Tastendruck auslöste.

Tastenprellen

Gehen wir davon aus, eine gedrückte Taste "zieht" eine der Port-Leitungen PAØ . . . PA6 nach log. Ø, ist im Idealfall der Spannungsverlauf an dieser Port-Leitung ein negativer Rechteckpuls (**Bild 8a**). Läßt man die Taste wieder los, sind alle Port-Leitungen log. 1. In der Praxis produziert eine gedrückte Taste ein Kontaktprellen, wie es **Bild 8b** zeigt. Zu den Zeitpunkten t3 und t5 überschreitet das Kontaktprellen die Triggerschwelle der Portlogik im IC 6532. Der Computer kann zu diesen Zeitpunkten von sich aus nicht unterscheiden, ob eine Taste gedrückt ist oder nicht. Deshalb muß das Monitorprogramm über eine Zeitverzögerung verfügen, damit die Port-Leitung erst dann abgefragt wird, wenn das Kontaktprellen zum Zeitpunkt t6 abgeklungen ist. In der Subroutine SCAND ist der Junior-Computer einige Millisekunden beschäftigt. Das ist genügend Zeit für das Abklingen des Kontaktprellens. Die benötigte Zeitverzögerung erzeugt die Programmsequenz BEQ-SCAND-BEQ. Ist bei der letzten BEQ-Instruktion das Tastenprellen noch immer vorhanden, durchläuft der Prozessor die Subroutine SCAND ein zweites mal. Oder anders gesagt, die Subroutine SCAND

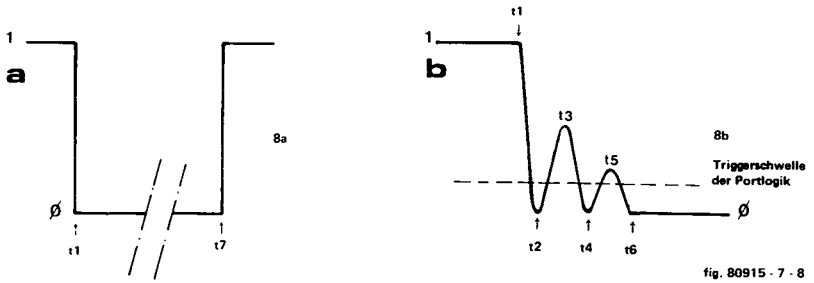


Bild 8. Bild 8a zeigt den idealen Spannungsverlauf an einer Port A-Leitung, wenn man eine der vom Computer abgefragten Tasten betätigt. Drückt der Programmierer eine der Tasten, wird die entsprechende Port-Leitung nach log. Ø gezogen (Zeitpunkt t1). Zum Zeitpunkt t7 läßt der Programmierer die Taste wieder los. Dabei springt die Port-Leitung wieder auf log. 1.
Bild 8b zeigt den praktischen Spannungsverlauf an einer Port A-Leitung, wenn der Programmierer eine Taste drückt. Dabei entsteht ein Tastenprellen. Dieses Tastenprellen wird durch zweimaliges Aufrufen der Subroutine SCAND in Bild 7 eliminiert. Bei der Berechnung des Tastenwertes in der Subroutine GETKEY können somit keine Berechnungsfehler unterlaufen, da sich die Port-Leitung bereits "beruhigt" hat.

wird so oft durchlaufen, bis das Tastenprellen auf der Port-Leitung abgeklungen ist.

Der Programmierer unterhält sich mit dem Junior-Computer

Bekanntlich findet das Gespräch zwischen dem Programmierer und dem Computer über das Tastenfeld und Display statt. Dabei verrichtet die I/0 auf dem Junior-Computer eine harte Arbeit: Die I/0 bedient dabei von Software gesteuerte Hardware (= Display) oder der Computer liest über die I/0 von Hardware (= Keyboard) erzeugte Daten.
Bild 9 zeigt nochmals, was an die I/0 angeschlossen ist (vergleiche Buch 1, Kapitel 1, Bild 4). Dabei arbeitet Port B (PB1 . . . PB4) ausschließlich als Ausgang und ist an den Punkten A . . . D von IC7 angeschlossen. In Abhängigkeit des Bitmusters auf den Eingängen A . . . D ist einer der Ausgänge von IC7 log. Ø. Sechs der zehn Ausgänge sind verbunden mit den gemeinsamen Daten des Displays. Sollen die Segmente eines bestimmten Displays aufleuchten, muß die gemeinsame Kathode des Displays auf Masse-Potential liegen. Port B erledigt diese Aufgabe über den Displayschalter IC7 (BCD-Dezimaldekoder). Die folgende Wahrheitstabelle gibt darüber Auskunft:

Tabelle 1

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PBØ	eingeschaltetes Display	
			D	C	B	A			
X	X	X	Ø	1	Ø	Ø	X	Di1	ADH (buffer POINTH)
X	X	X	Ø	1	Ø	1	X	Di2	
X	X	X	Ø	1	1	Ø	X	Di3	ADL (buffer POINTL)
X	X	X	Ø	1	1	1	X	Di4	
X	X	X	1	Ø	Ø	Ø	X	Di5	data (buffer INH)
X	X	X	1	Ø	Ø	1	X	Di6	

Wenn ein Portbit mit "X" gekennzeichnet ist, kann es log. Ø oder log. 1 sein, da an die betreffende Port-Leitung nichts angeschlossen ist.
Ist ein bestimmtes Display mit dem Displayschalter IC7 eingeschaltet, und sind alle Ausgänge von IC11 log. 1, leuchten alle Segmente dieses Displays auf. Das Display zeigt dann eine "8". Ein Display muß die Zeichen Ø . . .F anzeigen, das sind 16 verschiedene Zeichen. Deshalb muß die Möglichkeit vorhanden sein, verschiedene Segmente ein- und auszuschalten. Diese Aufgabe erledigt Port A, wenn er als Ausgang deklariert ist. Port A arbeitet somit als Segmentschalter.
Soll beispielsweise das Segment "C" ausgeschaltet werden, dann müssen wir dafür sorgen, daß der Inverterausgang, angeschlossen an Pin 12 von IC11 auf Massepotential liegt. In diesen Ausgang fließt dann Strom über den Widerstand R11 und am Segment kann keine Spannung abfallen. Um dieses zu erreichen, muß der entsprechende Invertereingang (Pin 5 von IC11) log. 1 sein. Oder anders gesagt, auf der Ausgangsleitung PA2 muß log. 1 stehen. Somit lassen sich durch ein Bitmuster auf den Ausgangsleitungen von Port A die Segmente eines Displays ein- und ausschalten. Steht auf eine Port A-Leitung log. 1, leuchtet das betreffende Segment nicht, bei log. Ø ist das angesteuerte Segment eingeschaltet. Tabelle 2 zeigt, welches Bitmuster auf den Port A-Leitungen stehen muß, um die entsprechenden hexadezimalen Zahlensymbole auf dem Display aufleuchten zu lassen. Die genaue Bezeichnung der Segmente eines 7-Segment-Displays zeigt **Bild 10**.

Tabelle 2

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ	hexadezimal
	$\overline{g}$	$\overline{f}$	$\overline{e}$	$\overline{d}$	$\overline{c}$	$\overline{b}$	$\overline{a}$	
X	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø
X	1	1	1	1	Ø	Ø	1	1
X	Ø	1	Ø	Ø	1	Ø	Ø	2
X	Ø	1	1	Ø	Ø	Ø	Ø	3
X	Ø	Ø	1	1	Ø	Ø	1	4
X	Ø	Ø	1	Ø	Ø	1	Ø	5
X	Ø	Ø	Ø	Ø	Ø	1	Ø	6
X	1	1	1	1	Ø	Ø	Ø	7
X	Ø	Ø	Ø	Ø	Ø	Ø	Ø	8
X	Ø	Ø	1	Ø	Ø	Ø	Ø	9
X	Ø	Ø	Ø	1	Ø	Ø	Ø	A
X	Ø	Ø	Ø	Ø	Ø	1	1	B
X	1	Ø	Ø	Ø	1	1	Ø	C
X	Ø	1	Ø	Ø	Ø	Ø	1	D
X	Ø	Ø	Ø	Ø	1	1	Ø	E
X	Ø	Ø	Ø	1	1	1	Ø	F

Bei der Steuerung der Segmente ist PA7 als Eingang erklärt. An dieser Leitung ist nichts angeschlossen. Das Bitmister für die einzelnen Segmente in Tabelle 2 ist im EPROM des Junior-Computers in einer Lookup Table abgelegt. Die 16 Hexzahlen dieser Tabelle befinden sich an den Adressen 1FØF . . . 1F1E (siehe Source Listing im Anhang dieses Buches).

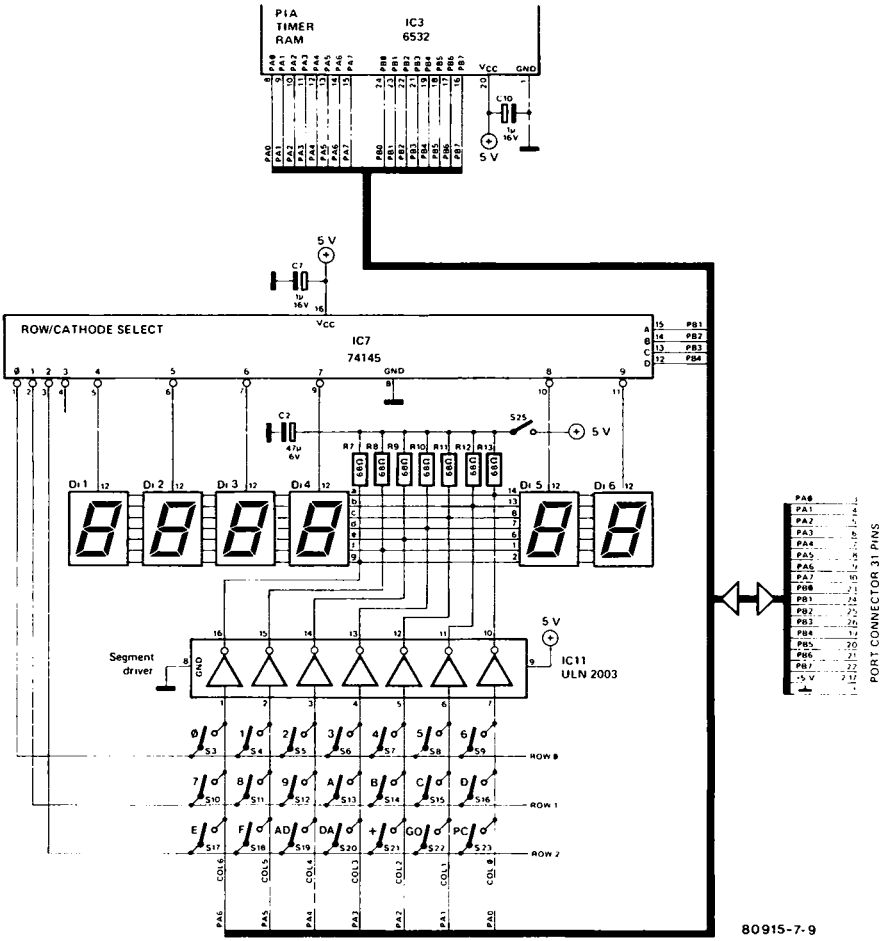


Bild 9. Ausschnitt aus dem Schaltbild des Junior-Computers. So sind das Display, das Tastenfeld und der Port-Konnektor an den PIA (IC3) angeschlossen.

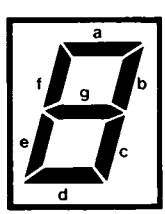


Bild 10. Die Segmente der Displays sind mit den Buchstaben a . . . g gekennzeichnet. Jedem Segment ist dabei eine Leitung von Port A zugeordnet. Wegen dem Segment-Treiber IC11 in Bild 9 sind die Port-Leitungen invertiert auf die Segmente geschaltet. Ist eine Port A-Leitung log. Ø leuchtet das angeschlossene Display-Segment auf, während bei einer log. 1 das angeschlossene Segment erlischt.

Taste gedrückt oder nicht gedrückt?

Aus **Bild 9** ist ersichtlich, daß jede Taste zwei Anschlüsse hat. Ein Kontakt ist mit einem der Ausgänge von IC7 verbunden, der andere mit einer der Port-Leitungen PAØ . . . PA6. Die Tasten sind in einer Matrix angeordnet. Diese Matrix besteht aus drei Reihen und Spalten. Tasten in einer Reihe sind gemeinsam mit einem bestimmten Ausgang von IC7 verbunden. Die Tasten in einer Spalte sind gemeinsam mit einer Port-Leitung verbunden.

Um festzustellen, ob irgendeine Taste gedrückt ist, ist Port A als Eingang programmiert. Durch das Bitmuster auf den Leitungen PB1 . . . PB4 wird bewirkt, daß einer der Ausgänge "Ø" oder "1" oder "2" von IC7 log. Ø ist. Somit verursacht das Drücken einer Taste eine log. Ø auf einer der Port-Leitungen PAØ . . . PA6.

Auf diese Weise läßt sich eine Taste eindeutig identifizieren:

1. Das Bitmuster auf den Port-Leitungen PB1 . . . PB4 bestimmt die Reihe, in der sich die gedrückte Taste befindet.
2. Die Spalte, in der sich die gedrückte Taste befindet, läßt sich leicht aufspüren. Dazu muß der Prozessor nur Port A lesen, zum Beispiel in den Accumulator. Im Accu ist dann das Bit null, in dessen Spalte eine Taste gedrückt ist.

Tabelle 3 und Tabelle 4 zeigen zusammenfassend, wie der Computer eine gedrückte Taste in der Keyboard-Matrix identifizieren kann:

Tabelle 3

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PBØ	Port B
			D	C	B	A		
X	X	X	Ø	Ø	Ø	Ø	X	Reihe Ø mit den Tasten Ø, 1, 2, 3, 4, 5 und 6
X	X	X	Ø	Ø	Ø	1	X	Reihe 1 mit den Tasten 7, 8, 9, A, B, C und D
X	X	X	Ø	Ø	1	Ø	X	Reihe 2 mit den Tasten E, F, AD, DA, +, GO und PC

Tabelle 4

	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PAØ
Tasten Ø, 7 und E	X	Ø	1	1	1	1	1	1
Tasten 1, 8 und F	X	1	Ø	1	1	1	1	1
Tasten 2, 9 und AD	X	1	1	Ø	1	1	1	1
Tasten 3, A, und DA	X	1	1	1	Ø	1	1	1
Tasten 4, B und +	X	1	1	1	1	Ø	1	1
Tasten 5, C und GO	X	1	1	1	1	1	Ø	1
Tasten 6, D und PC	X	1	1	1	1	1	1	Ø

Die Subroutine SCAND minus AK

Während der Subroutine SCAND steuert der Junior-Computer das Display und fragt das Keyboard ab. Bevor wir diese Subroutine Instruktion für Instruktion besprechen, folgt zunächst eine Übersicht, was sich in groben

Zügen in dieser Subroutine ereignet. Aus **Bild 11** ist ersichtlich, daß die Subroutine SCAND von zwei anderen Subroutinen unterstützt wird: Diese beiden Subroutinen heißen SHOW und CONVD.
Auf dem Display müssen drei Displaybuffer wiedergegeben werden. Das sind die beiden Adreßbuffer POINTH, POINTL und der Datenbuffer INH. Diese drei Displaybuffer können sechs Daten-Nibble aufnehmen und somit das gesamte Display mit Daten versorgen. Während der Subroutine CONVD wird für eine bestimmte Zeit eines der sechs Displays aktiviert. Dann leuchten zwei oder mehrere Segmente auf. In der Subroutine SHOW wird bestimmt, welches Daten-Nibble der drei Displaybuffer POINTH, POINTL und INH auf dem Display aufleuchten soll, das heißt, welches Display nach dem Sprung in die Subroutine CONVD aktiv sein soll. Da jeder Displaybuffer zwei Daten-Nibble enthält, muß von der Subroutine SHOW zweimal in die Subroutine CONVD gesprungen werden. Drei Displaybuffer sind auf dem Display anzuzeigen. Deshalb wird während der Subroutine SCAND beziehungsweise SCANDS die Subroutine SHOW dreimal aufgerufen. Während der Subroutine SCAND leuchten somit alle sechs Displays nacheinander für eine bestimmte Zeit auf. Die Displays werden also im Multiplexbetrieb vom Computer angesteuert. Beim Junior-Computer sprechen wir von einem **Software gesteuerten Multiplex-Display.** Wie wir wissen, befindet sich der Junior-Computer in einer Warteschleife, solange keine Taste auf dem Keyboard gedrückt ist (siehe Bild 7, zweiter Aufruf der Subroutine SCAND gefolgt von BEQ). Während dieser Warteschleife bedient der Computer die sechs Displays periodisch.
Bild 12 zeigt die Instruktionen der Subroutine SCAND. Wir beschränken uns zunächst auf den Teil der Subroutine, der zwischen dem Label SCAND und dem Label AK steht.
Begonnen wird mit dem Auffrischen des Datenbuffers INH. In diesen Buffer gelangen die Daten der Speicherzelle, auf die der Adreß-Pointer POINTH, POINTL zeigt. (In Editor-Mode werden die drei Displaybuffer für andere Zwecke gebraucht). Dann sind wir bei Label SCANDS angekommen. Dort erklärt der Computer Port A als Ausgang und verwendet diesen als Segmentschalter. Durch das Laden des X-Registers mit Ø8 und durch den Transport des Inhalts von X zum Port B (PB1 . . . PB4 ist Ausgang) während der Subroutine CONVD, wird der Displayschalter IC7 (Buch 1, Kapitel 1, Bild 4) in den Stand "Di1" gesetzt. Wir merken uns: Das X-Register steuert über den Port B den Displayschalter IC7. Das X-Register wird während der Subroutinen SHOW und CONVD modifiziert. Somit zeigt das X-Register immer auf das nächste anzuzeigende Display.

Jetzt gelangt der Inhalt von BYTES in das Y-Register der CPU. Der Inhalt von BYTES bestimmt, ob zwei, vier oder sechs Displays aufleuchten sollen. Nach dem Drücken der RST-Taste ist BYTES mit drei geladen, das heißt, alle drei Datenbuffer POINTH, POINTL und INH sollen auf dem Display angezeigt werden. Jetzt werden die beiden Daten-Nibble in POINTH zum Accu transportiert und wir springen zur Subroutine SHOW. Die beiden Daten-Nibble in POINTH werden hintereinander während der Subroutinen SHOW und CONVD auf den Displays Di1 und Di2 (siehe Bild 11) abgebildet. Nach einer Dekrementierung des Y-Registers mit eins rufen wir nochmals die Subroutine SHOW auf und zeigen den Inhalt von POINTL auf den Displays Di3 und Di4 an. Der Adreß-Pointer POINTH, POINTL

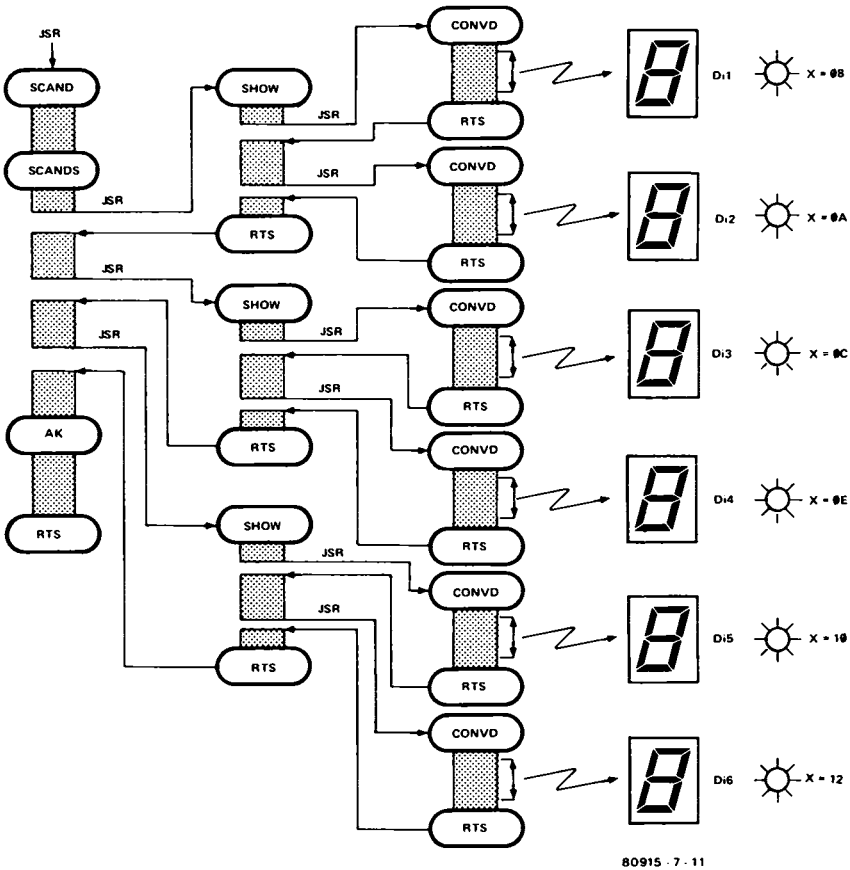


Bild 11. Während der Subroutine SCAND bzw. SCANDS bedient der Junior-Computer das sechsstellige 7-Segment-Display. Dabei sind die Subroutinen SHOW und CONVD ineinander verschachtelt (Subroutinen Nesting). Hinter dem Label SCANDS entscheidet der Computer, ob der Displaybuffer POINTH, POINTL oder INH auf dem Display angezeigt werden soll. Der Inhalt des betreffenden Displaybuffers gelangt in der Subroutine SHOW zum Display. Die Umwandlung einer Hexzahl in ein 7-Segment-Muster erfolgt in der Subroutine CONVD. Da drei Buffer auf dem Display anzuzeigen sind, wird die Subroutine CONVD während der Subroutine SCANDS sechsmal aufgerufen, da der Junior-Computer sechs Displays hat.
Die Subroutine AK ist ein Anhängsel der Subroutinen SCAND/SCANDS. In diesem Programmteil bestimmt der Computer, ob überhaupt eine Taste auf dem Tastenfeld gedrückt ist.

ist somit auf dem Display angezeigt. Dann wird das Y-Register abermals um eins dekrementiert und der Inhalt von INH auf den Displays Di5 und Di6 angezeigt. Die beiden anschließenden Befehle erklären Port A zum Eingang. Das ist eine Vorbereitung auf die Subroutine AK, deren Be-

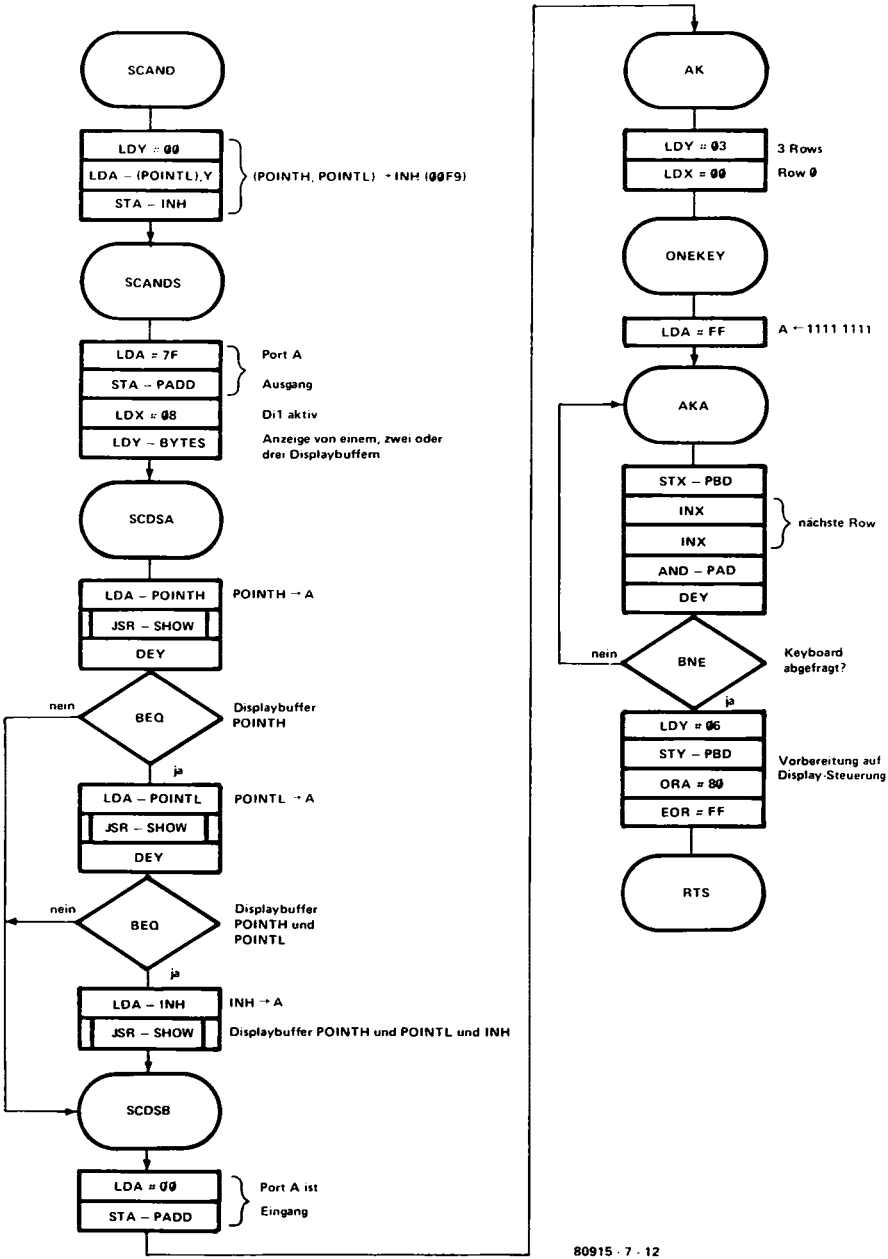


Bild 12. Das detaillierte Flußdiagramm der Subroutinen SCAND/SCANDS und AK. AK ist eine für sich selbstständige Subroutine, mit welcher der Programmierer prüfen kann, ob irgendeine Taste des Tastenfeldes gedrückt ist.

sprechung zu einem späteren Zeitpunkt folgt. Beim ersten Hinsehen erscheint der Gebrauch des Y-Registers als "Buffer-Zähler" ein Luxus zu sein, da die Anzahl der anzuzeigenden Displaybuffer bekannt ist. Die Subroutine SCANDS wird auch beim Editor verwendet. Wie bekannt, hat das Display beim Editen eine variable Länge. Das Y-Register hat somit als Buffer-Zähler seine Berechtigung.

Die Subroutine SHOW

Bild 13 zeigt die Subroutine SHOW. Bevor man diese Subroutine aufruft, muß im Accu der Inhalt des auf dem Display anzuzeigenden Datenbuffers stehen. Ein Byte besteht bekanntlich aus zwei Nibbles. Jedem Nibble ist dabei ein 7-Segment Display zugeordnet. Während der Subroutine SHOW erfolgt zweimal ein Sprung zur Subroutine CONVD. Die Subroutine CONVD arbeitet mit dem Accuinhalt als "Input", wobei das hochwertige Daten-Nibble null sein muß. Das niederwertige Accu-Nibble ist die auf dem 7-Segment Display anzuzeigende Zahl. Da der Multiplexbetrieb das sechsstellige Display von links nach rechts steuert, wird immer zuerst das hochwertige Daten-Nibble und dann das niederwertige wiedergegeben. Deshalb ist am Anfang der Subroutine SHOW eine viermalige Schiebeoperation nach rechts erforderlich, wobei das niederwertige Nibble die auf dem Display darzustellende Zahl ist und das hochwertige Nibble automatisch null ist (nach 4 maligem LSR-A). Damit danach auch das niederwertige Daten-Nibble des Displaybuffers angezeigt werden kann, muß der Inhalt

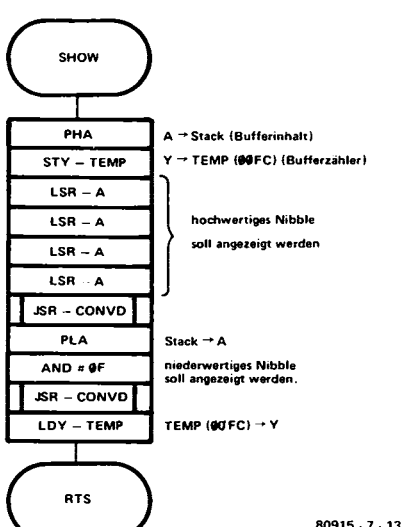


Bild 13. Detailliertes Flußdiagramm der Subroutine SHOW. Diese Subroutine setzt das anzuzeigende Daten-Nibble in den Accu und zeigt dieses anschließend in der Subroutine CONVD auf dem Display an. Dabei zeigt der Computer zuerst das hochwertige und dann das niederwertige Daten-Nibble im Accu an.

des Displaybuffers vor der Schiebeoperation gerettet werden. Das Retten erledigt eine PHA-Instruktion. Dabei verwenden wir den Stack als temporären Zwischenspeicher. Auch der Inhalt des Buffer-Zählers ist zu retten, da während der Subroutine CONVD das Y-Register für einen anderen Zweck frei sein muß. Jetzt rufen wir die Subroutine CONVD auf und zeigen das hochwertige Daten-Nibble im Display an. Während der Subroutine CONVD wandelt der Computer die anzuzeigende Hexzahl in ein 7-Segment-Muster um. Wie diese Umwandlung vor sich geht, besprechen wir später.
Nach der Rückkehr aus der Subroutine CONVD in die Subroutine SHOW wird der ursprüngliche Accuzustand wieder hergestellt. Der Accuinhalt ist jetzt wieder der gesamte Inhalt des anzuzeigenden Displaybuffers POINTH, POINTL oder INH. Nun soll das niederwertige Daten-Nibble angezeigt werden. Dazu ist der Befehl AND # ØF erforderlich. Ein abermaliger Sprung in die Subroutine CONVD gibt das niederwertige Daten-Nibble auf dem entsprechenden 7-Segment-Display wieder. Nach der Rückkehr aus der Subroutine CONVD stellt der Befehl LDY-TEMP den ursprünglichen Zustand des Buffer-Zählers wieder her. Somit kann sich der Computer um den nächsten anzuzeigenden Displaybuffer kümmern.

Die Subroutine CONVD

Wie schon erwähnt, leitet diese Subroutine das anzuzeigende Daten-Nibble zum 7-Segment-Display weiter. Dabei muß eine Hexzahl in ein entsprechendes 7-Segment-Muster umgewandelt werden. Bevor man die Subroutine CONVD aufrufen kann, muß das niederwertige Accu-Nibble die darzustellende Hexzahl enthalten. **Bild 14** zeigt den Ablauf der Subroutine CONVD. Am Anfang dieser Subroutine transportiert der Prozessor die darzustellende Hexzahl ins Y-Register, das als Indexregister Verwendung findet (TAY). Jetzt wird der Accu indiziert geladen, das heißt, wir holen aus einer Lookup Table das 7-Segment-Muster der abzubildenden Hexzahl. Sollen wir beispielsweise die Hexzahl "D" darstellen, holt die Instruktion LDA-Look,Y das 7-Segment-Muster 21. Für die eigentliche Umwandlung einer Hexzahl in das entsprechende 7-Segment-Muster ist also nur folgendes erforderlich:
TAY
LDA-LOOK,Y und eine Lookup Table.
Das Segmentmuster wandert jetzt vom Accu zu dem als Ausgang erklärten Port A. Das X-Register steuert über den PIA den Segmentschalter an. Deshalb wird der Inhalt des X-Registers auf den Port B gegeben. Erst jetzt leuchtet das entsprechende 7-Segment-Display auf und gibt die darzustellende Zahl wieder. Nächste Frage: Wie lange leuchtet das Display auf? Dafür ist die Warteschleife DELAY-BPL-DELAY zuständig. Der Prozessor durchläuft diese Programmsequenz solange, bis die N-Flag gesetzt ist. Somit ist ein momentan angesteuertes Display für etwas mehr als 63Ø µs eingeschaltet. Beim Verlassen der Programmschleife DELAY-BPL-DELAY ist der Inhalt des Y-Registers FF. Dieses Bitmuster legt der Computer auf die Port A Leitungen und schaltet somit alle Displaysegmente aus. Die Instruktion dafür ist STY-PAD. Dann bringen wir mit den Instruktionen

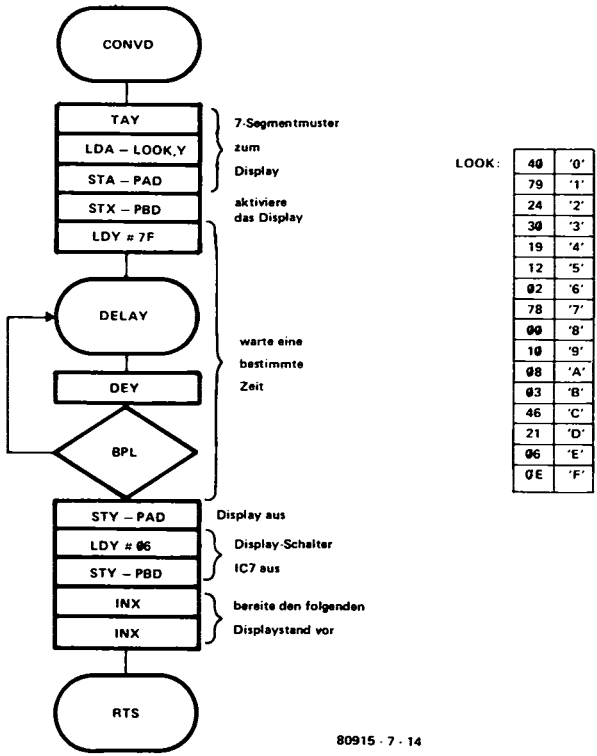


Bild 14. Detailliertes Flußdiagramm der Subroutine CONVD. Während dieser Subroutine setzt der Computer die anzuzeigende Hexzahl in den 7-Segment-Code um. Anschließend wird das entsprechende Display für ca. 600 µs eingeschaltet, bevor sich der Computer um das folgende Display kümmert.

LDY # Ø6 und STY-PBD den Displayschalter in den neutralen Stand. Jetzt ist der nichtangeschlossene Ausgang von IC7 (Buch 1, Kapitel 1, Bild 4) aktiv und keine der Display-Kathoden wird auf Masse gezogen. Am Ende der Subroutine CONVD wird durch zweimaliges Inkrementieren der Inhalt des X-Registers gleich Ø8. Der Displayschalter wird durch eine STX-PBD Instruktion bei abermaligem Aufruf der Sybroutine CONVD zurückgesetzt (Di1 leuchtet auf) und der beschriebene Vorgang wiederholt sich.
Bevor man die Subroutine CONVD aufruft, ist dafür zu sorgen, daß das hochwertige Daten-Nibble im Accu null ist. Anderenfalls kann der Inhalt des Y-Registers größer als ØF sein und die höchste Adresse der Lookup Table überschritten werden. Da hinter der Adresse 1F1E (Ende der Lookup Table) auch noch Bytes stehen, könnten also auf dem Display willkürliche Segmente aufleuchten, die zu einer Fehlanzeige führen.

AK, das Anhängsel der Subroutinen SCAND bzw. SCANDS

Die Subroutine AK schließt sich den Subroutinen SCAND bzw. SCANDS an, das heißt, die drei Subroutinen SCAND, SCANDS und AK arbeiten mit einem und demselben RTS-Befehl. Die Subroutine AK hat die Aufgabe, irgendeine beliebige gedrückte Keyboard-Taste aufzuspüren. Um die Funktionsweise von AK zu erklären, nehmen wir **Bild 9** und **Bild 16** zur Hand. Das Bitmuster auf den Port B Leitungen bestimmt einerseits, welches Display eingeschaltet ist und andererseits, welche Reihe der Keyboard-Tastatur abgefragt werden soll. Dabei steuert das X-Register Port B.

Wie aus Buch 1 bereits bekannt ist, ist am Ende der Subroutinen SCAND, SCANDS beziehungsweise AK der Accuinhalt ungleich null, wenn der Computer in irgendeiner Keyboard-Reihe eine gedrückte Taste entdeckt. Wie ist das möglich?

Am Anfang der Subroutine AK sind alle Bits auf den Port A-Leitungen eins, da diese Leitungen über interne Pullup-Widerstände "hochgezogen" sind. Dann wird das Y-Register mit Ø3 geladen, das als Reihenzähler in AK arbeitet. Drei Tastenreihen sind beim Junior-Computer abzufragen: ROWØ, ROW1 und ROW2. Welche dieser drei Reihen über Port B ausgewählt wird, bestimmt das Bitmuster im X-Register. Hinter dem Label ONEKEY laden wir den Accu mit FF. Anschließend gelangt der Inhalt des X-Registers (X=ØØ) zum Port B und die Zeile ROWØ wird nach Masse gezogen (siehe Tabelle 3). Das folgende zweimalige Erhöhen des X-Registers bedeutet eine Vorbereitung auf die nächste Reihe. Es folgt eine AND-Funktion zwischen dem Accu und dem Bitmuster auf den Port A Leitungen. Nachdem die Programmschleife AKA-BNE-AKA dreimal durchlaufen ist, kommt die Anzahl der Nullen im Accu mit den in einer beliebigen Spalte (COLØ . . . COL6) (gleichzeitig) gedrückten Tasten überein. Normalerweise ist jedoch nur eine Taste gedrückt. Ist beispielsweise in COL3 einer der Tasten 3, A oder DA gedrückt, ist nach der AND-Funktion das Bit "b3" im Accu gleich null.

Nach dem Verlassen der Programmschleife wird der Display-Reihenschalter (IC7 auf der Basisplatine des Junior-Computers) wieder in den neutralen Stand gesetzt. Die Tabelle 1 und 3 geben darüber weitere Auskunft.

Die Instruktion ORA # 8Ø macht b7 im Accu zu eins. Das ist wichtig, da an Port A beim Standart Junior-Computer periphere Geräte angeschlossen sind, wie zum Beispiel ein Printer. Dabei kann b7 bei der ORA-Instruktion von Zeit zu Zeit ändern. Für die folgende EOR-Instruktion ist somit alles vorbereitet. Dabei werden alle Bits im Accu invertiert (EOR # FF). Wie wir wissen, ist irgendein Bit im Accu gleich null, wenn während der Programmschleife AKA-BNE-AKA irgendeine Keyboard-Taste gedrückt ist. Deshalb ist nach der EOR-Instruktion der Accu ungleich null, wenn eine Taste gedrückt war; aber gleich null, wenn keine Taste gedrückt war. Somit ist auch die Subroutine AK besprochen. Diese Subroutine kann nur erkennen, ob überhaupt eine Keyboard-Taste gedrückt ist. Wie der Computer den Wert der gedrückten Taste berechnet, besprechen wir im folgenden Abschnitt.

Die Subroutine GETKEY

Nachdem der Computer irgendeine gedrückte Keyboard-Taste lokalisiert hat, muß er noch den Wert dieser Taste berechnen. Die Subroutine GETKEY berechnet diesen Tastenwert. **Bild 16** zeigt nochmals, wie die einzelnen Tasten in der Keyboard-Matrix angeordnet sind. Auch die Tastenwerte mit den entsprechenden Symbolen sind angegeben. Die einzelnen Instruktionen der Subroutine GETKEY zeigen die **Bilder 15a** und **15b**. In Bild 15b ist der Programmteil ONEKEY aus Bild 12 der Übersichtlichkeit wegen nochmals gezeichnet.

Zunächst lädt das X-Register die Hexzahl 21. Das hat zur Folge, daß beim folgenden Sprung in die Subroutine ONEKEY die Tastenzeile ROWØ abgefragt wird. Das Laden des Y-Registers mit Ø1 bedeutet, daß während der Subroutine ONEKEY nur eine Reihe abgehandelt werden soll. Nach der Rückkehr von ONEKEY ist der Accuinhalt ungleich null, wenn in ROWØ eine Taste gedrückt ist. Ist keine Taste in dieser Zeile gedrückt, ist der Accuinhalt gleich null. Mit der folgenden BNE-Instruktion verzweigt das Programm nur dann zum Label KEYIN, wenn eine Taste während ONEKEY gedrückt ist. Ist keine Taste gedrückt, ist der Inhalt des X-Registers ungleich 27, das Programm verzweigt zurück zum Label GETKEA und die Subroutine ONEKEY wird erneut aufgerufen.

Was hat der Vergleich des X-Registers mit 27 zu bedeuten? Am Anfang der Subroutine GETKEY ist X gleich 21 und die Tastenzeile ROWØ ist somit aktiv. Innerhalb der Subroutine ONEKEY wird das X-Register zweimal mit eins inkrementiert, so daß sein Inhalt jetzt 23 ist. Nun ist 23 ungleich 27 und die Subroutine ONEKEY wird abermals durchlaufen. Allerdings ist jetzt nicht mehr die Tastenzeile ROWØ, sondern die Tastenzeile ROW1 aktiv. Nach der Rückkehr von ONEKEY ist das X-Register gleich 25. Ist nun auch in der Tastenreihe ROW1 keine Taste gedrückt, durchläuft der Prozessor abermals die Subroutine ONEKEY. Nach dem dritten Durchlauf ist der Inhalt des X-Registers auf 27 angewachsen.

Jetzt sind alle Tastenreihen abgefragt. Ist nach dreimaligem Durchlaufen der Subroutine ONEKEY noch immer keine gedrückte Taste festgestellt, liegt irgendein Fehler vor. Der Computer darf nicht zum Label KEYIN verzweigen und mit der Berechnung des Tastenwertes beginnen. Deshalb wird nach der CPX # 27 Instruktion der Accu mit 15 geladen und dem Computer somit mitgeteilt, daß er die gedrückte Taste ignorieren und mit der Keyboard-Abfrage aufs neue beginnen soll. Kehrt der Computer aus der Subroutine GETKEY mit einem Accuinhalt von 15 zurück, handelt es sich um eine "falsche Taste".

Nun kommen wir wieder zum Label KEYIN. Ab diesem Label berechnet der Computer den Wert der gedrückten Taste. Dabei nehmen wir wieder **Bild 16** zur Hand, das die X- und die Y-Koordinaten der einzelnen Keyboardtasten zeigt.

Am Anfang von KEYIN wird das Y-Register mit FF geladen. Der Computer schiebt anschließend den Accuinhalt mindestens zweimal nach links. Vor dem Schieben enthält der Accu sieben Nullen und eine Eins. Der Computer schiebt den Accuinhalt solange nach links, bis in der C-Flag eine Eins steht. Der Inhalt des Y-Registers gibt nach der Schiebeoperation (BCS) den Platz der gedrückten Taste an. Der Inhalt des Y-Registers gibt also

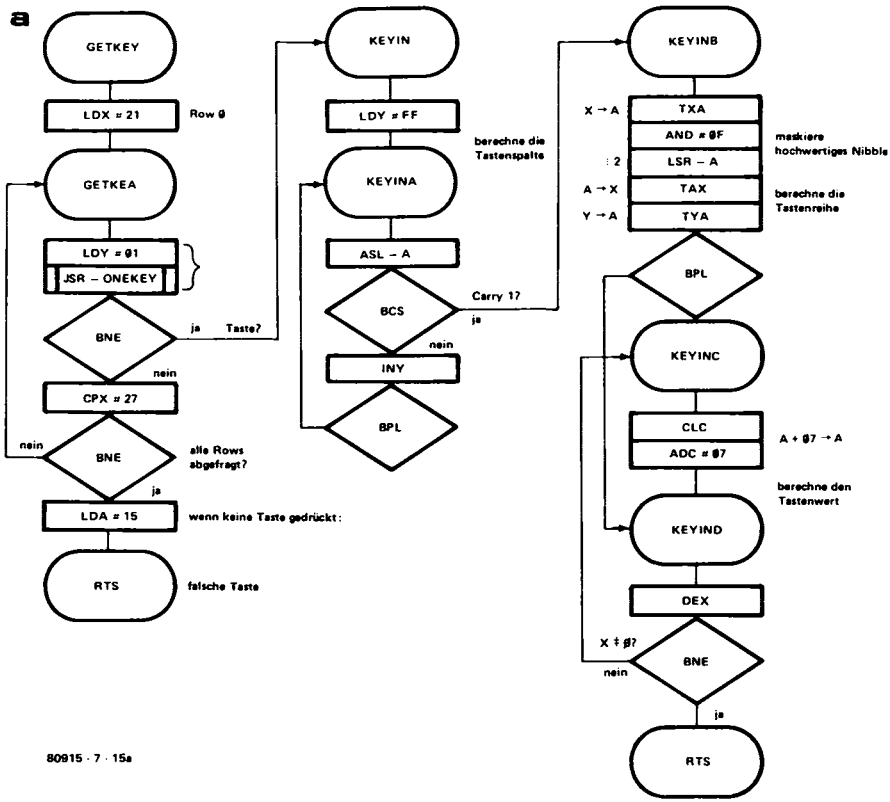
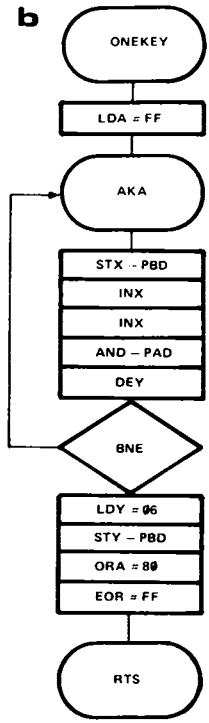


Bild 15. In der Subroutine GETKEY berechnet der Junior-Computer den Wert der momentan gedrückten Taste. Für die Abfrage der ROWs verwendet er die bekannte Subroutine ONEKEY, einen Teil der Subroutine AK.

Auskunft, in welcher Spalte (COLØ . . . COL6) sich die gedrückte Taste befindet. Bild 16 gibt ebenfalls darüber Auskunft. Für die Tasten, die in der Tastenzeile ROWØ angeordnet sind, ist der Inhalt des Y-Registers nach der Schiebeoperation gleich dem Tastenwert. Für eine gedrückte Taste in ROW1 muß Ø7 addiert, für eine Taste in ROW2 muß ØE addiert werden. Die Hexzahl ØE ist zweimal die Hexzahl Ø7. Mit diesen Additionen, die eventuell ausgeführt werden müssen, beschäftigt sich der Programmteil KEYINB in **Bild 15a**.
Doch bevor das Programm soweit fortgeschritten ist, geht der Inhalt des X-Registers, das die Reiheninformation enthält, in den Accu. Der Wert von X ist 23 für eine Taste aus ROWØ, 25 für eine Taste aus ROW1, 27 für eine Taste aus ROW2. Dabei ist leicht zu verfolgen, daß der Inhalt des X-Registers vor dem Label KEYINC Ø1 ist, wenn es sich um eine gedrückte Taste in der Reihe ROWØ handelt. Der Inhalt ist Ø2, wenn die gedrückte Taste zur Reihe ROW1 gehört und Ø3, wenn die gedrückte Taste



80915 · 7 · 15b

aus der Reihe ROW2 ist. Ab dem Label KEYINC wird dann einmal oder zweimal der Wert Ø7 zur Spaltennummer der Taste addiert, je nachdem in welcher Tastenzeile die gedrückte Taste war. Die Programmschleife KEYINC-BNE-KEYINC durchläuft der Prozessor für Tasten aus ROWØ überhaupt nicht, da hinter dem Label KEYIND wegen der DEX-Instruktion das X-Register bereits null ist. Nach der Rückkehr aus der Subroutine GETKEY ist der Accuinhalt gleich dem Tastenwert der gedrückten Taste.

Um die Subroutine GETKEY weiter verständlich zu machen, sehen wir uns an, wenn auf dem Keyboard des Junior-Computers die Taste "C" gedrückt ist. Zu Beginn ist der Inhalt des X-Registers gleich 21 und der Inhalt des Y-Registers gleich Ø1. Es folgt ein Sprung zur Subroutine ONEKEY. Nachdem Port B mit X=21 geladen ist, liegt eine Seite von allen Tasten, die sich in ROWØ befinden auf Masse. Die Taste "C" gehört aber nicht zu den Tasten von ROWØ sondern zu den Tasten von ROW1. Beim Einlesen der Tasten in ROWØ ergibt sich dann folgende Situation:

X1111111	PORT A
11111111	Accu = FF
X1111111	AND-Funktion mit FF
11111111	OR-Funktion mit 8Ø
ØØØØØØØØ	EOR-Funktion mit FF

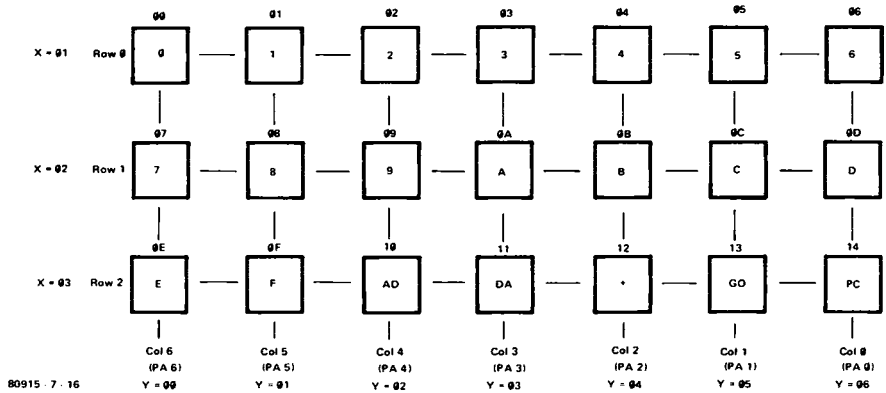


Bild 16. So sieht die Tastenmatrix des Junior-Computers aus. Jeder Taste ist eine X- und eine Y-Koordinate zugeordnet. Dabei enthält das X-Register der CPU die Y-Koordinate und das Y-Register die X-Koordinate der Taste. Die Tasten sind in einer 7 mal 3 Matrix angeordnet.

Da der Accuinhalt null ist, verzweigt das Programm nicht zum Label KEYIN. Während der Subroutine ONEKEY hat sich auch der Inhalt des X-Registers verändert:

ØØ1ØØØØ1 = 21 (für ROWØ)
ØØ1ØØØ1Ø = 22 nach INX
ØØ1ØØØ11 = 23 nach INX (für ROW1)

Bei der Rückkehr aus der Subroutine ONEKEY reagiert die BNE-Instruktion auf den zuletzt veränderten Registerinhalt oder auf die letzte Beeinflussung der Z-Flag. Die Instruktion, welcher die Z-Flag zuletzt beeinflußt, ist EOR # FF. Da die Taste "C" gedrückt ist, erfolgt wegen Z-Flag gleich eins keine Verzweigung zum Label KEYIN. Der Computer testet anschließend, ob der Inhalt des X-Registers schon 27 ist. Auch das trifft nicht zu, da das X-Register nach einmaligem Durchlaufen von ONEKEY erst 23 ist. Deshalb verzweigt das Programm zum Label GETKEA, das Y-Register wird wieder eins gemacht und wir springen abermals zur Subroutine ONEKEY. Der Accu wird wieder mit FF geladen und in Port B beschreiben wir den neuen Inhalt des X-Registers. Ein Anschluß des Tasten, die in der Tastenreihe ROW1 angeordnet sind, liegt jetzt auf Massepotential. Taste "C" liegt in ROW1 und das Bitmuster in Port A sieht wie folgt aus:

X1111ØØ1 PA1 = Spalte 1 (siehe Bild 16) = Taste "C"
und der Accuinhalt ist nach der AND-Berechnung mit PAD gleich
X1111ØØ1.

Nach der OR-Funktion mit FF ist der Inhalt des Accus schließlich 11111101. Die folgende EOR # FF Instruktion liefert einen Accuinhalt von 00000010 und das X-Register hat in der Zwischenzeit den Inhalt 25. Nach der Rückkehr aus der Subroutine ONEKEY treffen wir wieder auf eine BNE-Instruktion. Da der Inhalt des Accus (00000010) zu diesem Zeitpunkt ungleich null ist, erfolgt die Verzweigung zum Label KEYIN. Der Accuinhalt wird jetzt solange nach links geschoben, bis die C-Flag den Wert "1" annimmt:

0	00000010	FF	Start
0	00000100	FF	nach dem 1. ASL-A
0	00001000	00	nach dem 2. ASL-A
0	00010000	01	nach dem 3. ASL-A
0	00100000	02	nach dem 4. ASL-A
0	01000000	03	nach dem 5. ASL-A
0	10000000	04	nach dem 6. ASL-A
1	00000000	05	nach dem 7. ASL-A
↑		↑	
C-Flag		Y-Register nach dem ASL-A und vor der BCS-Instruktion.	

Nach 7-maligem Schieben ist die C-Flag eins geworden und das Programm verzweigt zum Label KEYINB. Da man die "C"-Taste aktiviert hat, ist der Inhalt des X-Registers gleich 23 und der Inhalt des Y-Registers gleich 5. Dann wird das X-Register in den Accu kopiert:

00100101	Accu vor dem ANDen
00000101	Accu nach dem ANDen mit 0F
00000010	Accu nach LSR-A (= 02)

Der Accuinhalt wandert wieder zum X-Register (TAX). In diesem Moment ist das X-Register 02. Jetzt wird das Y-Register in den Accu kopiert (TYA). Im Accu-Inhalt ist 05 und das ist keine negative Zahl. Deshalb verzweigt das Programm zum Label KEYIND, wobei das X-Register um eins dekrementiert. Der neue Inhalt des X-Registers ist jetzt eins. Da eins ungleich null ist, verzweigt das Programm zum Label KEYINC, hinter dem in Abhängigkeit des X-Registers 07 oder 0E zur Tastenspalte addiert wird. Wir erinnern uns:
5 mal wurde der Accuinhalt nach links geschoben, bis die C-Flag gesetzt war. Die "C"-Taste hat somit den vorläufigen Wert von 05. Die "C"-Taste befindet sich in ROW1 der Keyboardmatrix (Bild 16).
Zum vorläufigen Tastenwert 05 wird hinter dem Label KEYINC 07 addiert. Wir erhalten somit 05 + 07 = 0C. Das ist bereits der entgültige Tastenwert der Keyboardtaste "C". Nach der Addition wird das X-Register abermals dekrementiert. Es ist in unserem Beispiel null geworden. Im Accu der CPU steht nach dem Rücksprung ins Hauptprogramm der richtige Tastenwert der gedrückten Taste.
Damit ist das Kapitel "Monitorprogramm" abgeschlossen. Von den 1024

EPROM-Zellen, die fast vollständig belegt sind, haben wir wichtige Programmteile kennengelernt. 181 Bytes benötigte die Monitor-Hauptroutine und 152 Bytes belegen die Monitor-Subroutinen. Übrigens, die Routinen für die Abfrage des Keyboards und für die Steuerung des Displays verwenden wir auch in einem späteren Kapitel, wenn wir die Software des Editors besprechen.

8

Das Editor-Programm

Dieses Kapitel beschreibt das Editor-Programm. Was geschieht im Junior-Computer, wenn in Editor-Mode Daten eingegeben werden? Was spielt sich im EPROM ab, wenn eine der Kommandotasten INSERT, INPUT, DELETE usw. gedrückt ist? Wie legt der Editor im Arbeitsspeicher des Junior-Computers ein eingegebenes Programm ab, das anschließend der Assembler verarbeiten kann? Auf alle diese Fragen geht das Kapitel an Hand von Flußdiagrammen Byte für Byte ein.

Wie bereits bekannt, enthält das EPROM drei größere Programme: den Monitor für die Adreß- Datenanzeige, den Editor und den Assembler. Von diesen Programmen nimmt der Editor den meisten Speicherraum ein. Aus Kapitel 5 ist der Editor bereits als wirkungsvolles Hilfsmittel bei der Eingabe von größeren Programmen bekannt. Jetzt folgt die Erklärung des Editor-Programms, das sich aus vielen Subroutinen zusammensetzt. Diese Subroutinen kann auch der Programmierer in seine eigenen Programme einbauen und sich somit viel Zeit bei der Programmentwicklung sparen. Eine dieser Subroutinen ist bereits aus Buch 1 bekannt: die Subroutine GETBYT oder lese zwei Tastenwerte in den Accu der CPU.
Der Editor arbeitet mit den fünf Kommandotasten SEARCH, INSERT, INPUT, SKIP und DELETE sowie den Datentasten Ø . . . F zusammen. Das Drücken der Kommandotasten SEARCH, INSERT und INPUT führte dazu, daß der Computer auf zwei, vier oder sechs Datentasten wartet. Der Grund dafür ist die Instruktionslänge der CPU 65Ø2. Instruktionen können bei diesem Prozessor ein, zwei oder drei Bytes lang sein. Beim SEARCH-Kommando wartet der Junior-Computer immer auf vier Datentasten, da im Arbeitsspeicher ein bestimmtes Zwei-Byte-Muster ausgesucht werden soll.
Vor der Besprechung des Editor-Hauptprogramms und der diversen Subroutinen folgt eine Übersicht über spezielle Editor-Eigenschaften, auf die in Kapitel 5 nicht näher eingegangen ist. Kapitel 5 erklärt lediglich, wie der Editor zu gebrauchen ist. Das folgende Kapitel beschreibt, wie der Prozessor das Editor-Programm im EPROM abarbeitet.

Eigenschaften des Editors

Bevor man mit dem Editor arbeiten kann, muß man in den Speicherzellen BEGADH, BEGADL sowie ENDADH, ENDADL die Anfangs- und die Endadresse des Speicherbereiches eingeben, in welchem der Editor das Programm ablegen soll. Es ist auch möglich, mit dem Editor Instruktionen in den Computer einzugeben, ohne danach den Assembler zu starten. Allerdings darf der Programmierer dann keine Labels eingeben, da die CPU 65Ø2 mit dem Label-Identifikator FF nichts anfangen kann. Deshalb muß der Assembler nach dem Editen immer gestartet werden, wenn sich Labels im zuvor definierten Speicherbereich befinden. Bekanntlich entfernt der Assembler alle Labels aus einem Programm, das man zuvor mit dem Editor eingetippt hat. Ein weiterer Pseudobefehl der CPU 65Ø2 findet beim Editor Verwendung: Der EOF-Charakter 77 (EOF = end of file = Ende der Seite). Der EOF-Charakter steht am Ende aller eingegebenen Instruktionen. Wie aus Kapitel 5 bekannt ist, müssen am Ende einer Programmiereingabe zwischen dem EOF-Charakter und der Endadresse ENDADH, ENDADL noch mindestens sechs Speicherzellen vorhanden sein. Sonst entstehen beim anschließenden Starten des Assemblers große Probleme: Die vom Assembler angefertigte Symbol-Table überschreibt das Ende des eingetippten Programms (siehe Kapitel 9).

Ist der Editor einmal gestartet, können wir mit den Kommandotasten INSERT und INPUT Instruktionen in den Junior-Computer eingeben. Nachdem eine ganze Instruktion eingegeben ist, breitet sich der besetzte Speicherraum um einen, zwei oder drei Speicherplätze aus. Um wieviele Speicherplätze sich der besetzte Speicherraum ausbreitet, bestimmt der OP-Code der soeben eingegebenen Instruktion. Ferner hat die Ausbreitung des Speicherraumes nach der Eingabe einer Instruktion ein Verschieben des EOF-Charakters 77 zur Folge. Wird beispielsweise eine Instruktion mit einer Länge von zwei Bytes in den Junior-Computer eingegeben, ist nach der Eingabe dieser Instruktion der EOF-Charakter um zwei Speicherplätze nach unten geschoben.

Das Gegenteil geschieht beim Drücken der DELETE-Taste. Ein Drücken dieser Taste hat zur Folge, daß die momentan auf dem Display angezeigte Instruktion aus dem Arbeitsspeicher des Junior-Computers geholt wird. Die so entstehende Lücke füllt der Computer auf, indem er den der Instruktion folgenden Datenblock um ein, zwei oder drei Bytes nach oben schiebt. Auch die Position des EOF-Charakters wird dieser neuen Situation angepaßt: Der Computer schiebt den EOF-Charakter um ein, zwei oder drei Bytes nach oben.

Mit dem SEARCH-Kommando lassen sich Zwei-Byte-Muster im Speicher des Junior-Computers aufspüren. Das kann ein OP-Code + erstes Operanden-Byte oder ein Label mit dem Pseudo-OP-Code FF + Labelnummer sein. Die Suchaktion startet bei der Beginnadresse BEGADH, BEGADL und endet an der Adresse, an der sich der OP-Code des zu suchenden Zwei-Byte-Musters befindet.

Ein anderer Suchbefehl ist das SKIP-Kommando. Es spürt im Arbeitsspeicher die Instruktion auf, die der momentan angezeigten Instruktion folgt und zeigt diese auf dem Display an. Sowohl das SEARCH- wie das SKIP-Kommando kennen eine Fehlermeldung. Wird beispielsweise mit dem SEARCH-Kommando ein im Arbeitsspeicher nicht vorhandenes

Zwei-Byte-Muster aufgesucht, zeigt das Display solange EEEEEE, bis die SEARCH-Taste losgelassen ist. Nach dem Loslassen der SEARCH-Taste ist dann eine beliebige Instruktion auf dem Display sichtbar. Bei dieser Instruktion hat der Junior-Computer die Suchaktion abgebrochen. Auch das SKIP-Kommando kennt eine Fehlermeldung: Der EOF-Charakter ist das Ende eines eingegebenen Datenblocks, der sich aus Instruktionen und Labels zusammensetzt. Betrachtet der Programmierer mit dem SKIP-Kommando hinter dem EOF-Charakter Instruktionen, erscheint auf dem Display ebenfalls EEEEEE, solange die SKIP-Taste gedrückt ist.

Die Adreß-Pointer des Editors

Der Editor legt in einem Speicherbereich, der bei BEGAD beginnt und bei ENDAD endet, Daten ab. Der Speicherbereich zwischen BEGAD und ENDAD heißt "File" (File = Seite oder Datenblock). Für das Management dieser File benötigt der Editor vier Adreß-Pointer. Diese vier Adreß-Pointer heißen:

1. BEGAD: Dieser Adreß-Pointer ist in den Speicherzellen BEGADL mit der Adresse ØØE2 und BEGADH mit der Adresse ØØE3 abgelegt. BEGAD legt die Anfangsadresse der File fest, in der der Editor Daten ablegen soll.

2. ENDAD: Dieser Adreß-Pointer ist in den Speicherzellen ENDADL mit der Adresse ØØE4 und ENDADH mit der Adresse ØØE5 abgelegt. ENDAD legt die Endadresse der File fest, in der der Editor Daten ablegen soll.

3. CURAD: Dieser Adreß-Pointer ist in den Speicherzellen CURADL mit der Adresse ØØE6 und CURADH mit der Adresse ØØE7 abgelegt. CURAD ist eine Abkürzung des englischen Ausdrücks "current address", was soviel wie momentane Adresse bedeutet. Den Adreß-Pointer CURAD benötigt der Editor zur Steuerung des sechsstelligen Displays auf der Basisplatine des Junior-Computers. Da das Display alle Instruktionen der File anzeigt, ändert sich der Adreß-Pointer CURAD ständig beim Editen. Das Editor-Programm richtet den Pointer CURAD immer so ein, daß dieser auf den OP-Code der momentan auf dem Display sichtbaren Instruktion zeigt.

4. CEND: Dieser Adreß-Pointer ist in den Speicherzellen CENDL mit der Adresse ØØE8 und CENDH mit der Adresse ØØE9 abgelegt. CEND ist eine Abkürzung des englischen Ausdrucks "current end address", was soviel wie "momentane Endadresse" bedeutet. Der Pointer CEND ändert sich wie der Pointer CURAD beim Editen ständig. CEND verändert sich, wenn der Programmierer die Kommando-Tasten INSERT, INPUT oder DELETE betätigt, um Instruktionen in den Computer einzugeben oder aus diesem wieder herauszuholen. Der Adreß-Pointer CEND wird vom Editor-Programm so eingerichtet, daß dieser stets auf den Speicherplatz zeigt, der unmittelbar dem EOF-Charakter 77 folgt. Steht der EOF-Charakter beispielsweise an der Adresse Ø259, zeigt CEND auf die Adresse Ø25A.

Der Display-Pointer CURAD und der variable Endadreß-Pointer CEND werden beim Starten des Editors (Cold Start Entry siehe Kapitel 5) in eine bestimmte Anfangsposition gebracht. Bei Cold Start Entry zeigt CURAD auf die Beginnadresse der File. Der Computer legt dann an der Beginnadresse der File den EOF-Charakter 77 ab. Deshalb sehen wir auch nach einem Cold Start Entry den EOF-Charakter auf dem Display. Der Pointer CEND richtet sich jetzt so aus, daß er auf den Speicherplatz hinter dem

EOF-Charakter zeigt. Da noch keine Instruktionen eingegeben sind, zeigt CEND auf die Adresse BEGAD + 1. **Bild 2a** zeigt diese Beginnsituation.
Neben dem Cold Start Entry kennt der Editor auch noch einen Warm Start Entry. Beim Warm Start Entry beginnt der Editor wie gewohnt zu arbeiten, aber der Display-Pointer CURAD, der Adreß-Pointer CEND sowie der EOF-Charakter werden dabei **nicht** beeinflußt. Wie bereits aus Kapitel 5 bekannt, kann man den Editor durch Drücken der GO-Taste oder durch Drücken der ST-Taste starten. Die Startadresse für Cold Start Entry ist 1CD5. Auf diese Adresse muß auch der NMI-Vektor zeigen, wenn der Editor mit der ST-Taste gestartet werden soll. Dasselbe gilt natürlich auch für den Warm Start Entry. Mit der Startadresse 1CCA. Das Verlassen des Editors kann über die RST-Taste oder die ST-Taste erfolgen. Will man den Editor mit der ST-Taste verlassen, muß der NMI-Vektor gesetzt werden (siehe Kapitel 5).
Die Displaybuffer enthalten beim Editen nicht mehr Adressen und Daten, sondern Instruktionen, die der Computer auf dem sechsstelligen Display anzeigt. Die Displaybuffer haben beim Editor ebenfalls POINTH, POINTL und INH. Dabei enthält POINTH die Daten für die Displays Di1 und Di2, POINTL die Daten für die Displays Di3 und Di4 und INH die Daten für die Displays Di5 und Di6. Ansonsten weicht die Aufgabe der Displaybuffer von der in Kapitel 7 beschriebenen ab:

* Bei der Eingabe von Daten wird das Display von links nach rechts aufgefüllt.
* Der Computer liest die Daten byteweise ein. Man muß also zwei Datentasten drücken, bevor das eingegebene Byte auf dem Display erscheint.
* Beim Eingeben von Instruktionen liest der Computer zunächst den OP-Code und berechnet daraus die Länge der Instruktion. Somit ist bekannt, ob die auf dem Display darzustellende Instruktion ein, zwei oder drei Bytes lang ist. Der Computer weiß also nach der Eingabe des OP-Codes, ob die Displays Di1 . . . Di2, Di1 . . . Di4, oder Di1 . . . Di6 aufleuchten sollen. Beim Editen haben die Displaybuffer folgende Bedeutung:

POINTH: enthält immer den OP-Code einer Instruktion, den Pseudo-OP-Code eines Labels oder den EOF-Charakter.
POINTL: enthält immer das erste Operanden-Byte (falls anwesend) oder eine Label-Nummer.
INH: enthält immer das zweite Operanden-Byte (falls anwesend) oder das Begrenzerbyte eines Labels.
Bevor wir uns näher mit der Editor-Software befassen, noch ein paar Bemerkungen über die Tastenkommandos INSERT, INPUT und DELETE. Dazu betrachten wir **Bild 1**, das in drei Teile aufgeteilt ist.
***Bild 1a** zeigt die Beginnsituation. Dabei wird der OP-Code ③ mit den beiden folgenden Operanden-Bytes auf dem Display angezeigt. Auf den Op-Code ③ zeigt der Adreß-Pointer CURAD. Der EOF-Charakter steht an der Adresse XXXX-1 und der Inhalt des CEND-Pointers ist XXXX.
***Bild 1b** Der Programmierer will hinter die momentan angezeigte Instruktion eine weitere Instruktion eingeben: LDA # FF. Zur Eingabe der neuen Instruktion verwendet er die Kommandotaste INPUT. Die Instruktion LDA # FF kommt hinter die zuvor angezeigte Instruktion (OP-Code ③ mit seinen beiden Operandenbytes) zu stehen. Alle übrigen Bytes nach den beiden Operandenbytes von OP-Code ③ verschiebt der Computer um

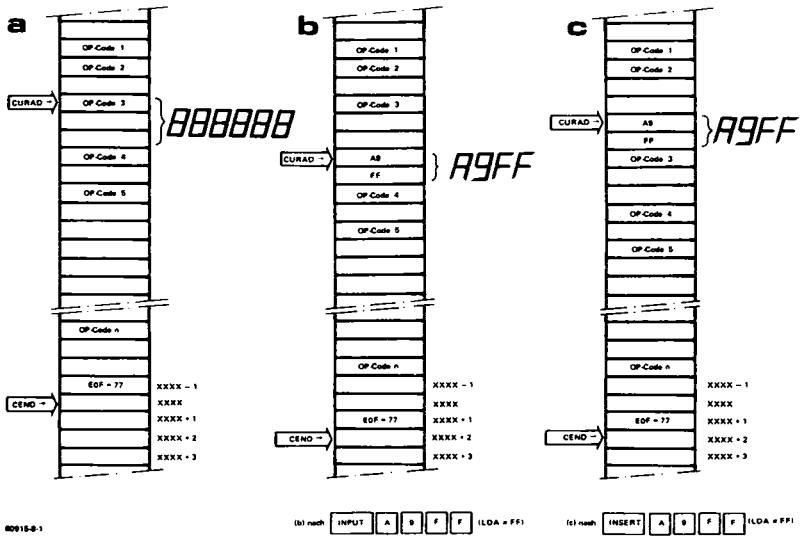


Bild 1. Bild 1a zeigt einen Ausschnitt aus der File, die der Programmierer mit dem Editor in den Junior-Computer eingegeben hat. In dieser File können an jeder beliebigen Stelle Instruktionen eingefügt werden.
Bild 1b und Bild 1c zeigen den Unterschied zwischen dem INPUT- und dem INSERT Kommando. In beiden Fällen wird die Instruktion LDA # FF oder A9 FF in die bestehende File eingefügt. Das Kommando INPUT setzt die Instruktion LDA # FF hinter die Instruktion mit dem OP-Code ③ ein. Dagegen setzt das INSERT-Kommando die Instruktion LDA # FF vor die Instruktion mit dem OP-Code ③ ein. Nach dem INSERT- oder INPUT-Kommando zeigt der Junior-Computer die eingegebene Instruktion auf dem Display an.

zwei Speicherplätze nach unten. Auch der EOF-Charakter wandert um zwei Stellen nach unten, da die Instruktion LDA # FF zwei Bytes lang ist.
* **Bild 1c** zeigt den soeben beschriebenen Vorgang für das INSERT-Kommando. Beim INSERT-Kommando kommt die neue Instruktion vor den OP-Code ③ zu stehen. Die Instruktion LDA # FF steht also unmittelbar vor dem OP-Code ③ im Arbeitsspeicher des Junior-Computers. Nach der Ausführung des INSERT-Kommandos ist der EOF-Charakter ebenfalls um zwei Stellen nach unten verschoben.
Aus Kapitel 5 ist bekannt, daß nach Cold Start Entry die erste Instruktion **immer** mit einem INSERT-Kommando eingegeben werden muß. **Bild 2** macht deutlich, warum diese Prozedur nötig ist. Ausgehend von der Beginnsituation in **Bild 2a** sehen wir was geschieht, wenn der Label FF 15 ØØ mit der INPUT-Taste in den Junior-Computer eingegeben wird: Der EOF-Charakter bleibt an derselben Stelle, wo er immer war und der Label FF 15 ØØ kommt hinter dem EOF-Charakter zu stehen **(Bild 2b)**. Und diese Situation führt beim Editen zu schlimmen Folgen.
Bild 2c zeigt das richtige Vorgehen, nachdem der Editor via Cold Start Entry gestartet wurde. Der Programmierer gibt die erste Instruktion (in

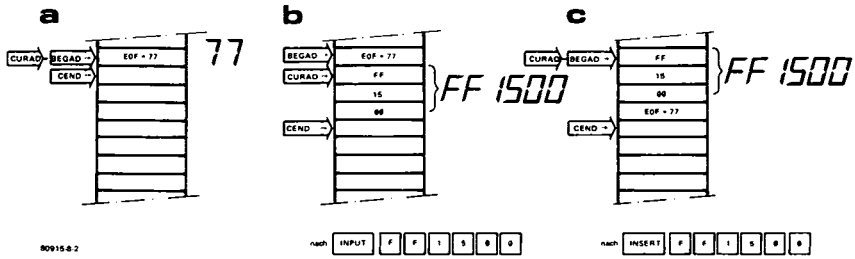


Bild 2. Bild 2a zeigt die Beginnsituation, wenn der Editor über Cold Start Entry gestartet ist. Da CURAD = BEGAD ist, erscheint zunächst der EOF-Charakter auf dem Display. Würde am Anfang die Instruktion LDA # FF mit der INPUT-Taste in den Computer eingegeben, käme diese Instruktion hinter den EOF-Charakter zu stehen. Das ist jedoch verboten, da der Editor dann bei jeder weiteren Dateneingabe eine Fehlermeldung macht. Bild 2b zeigt diese verbotene Situation.
In Bild 2c erfolgt die erste Dateneingabe mit dem INSERT-Kommando. Der Label FF 15 00 kommt somit vor dem EOF-Charakter zu stehen. Erst jetzt kann das INPUT-Kommando verwendet werden.

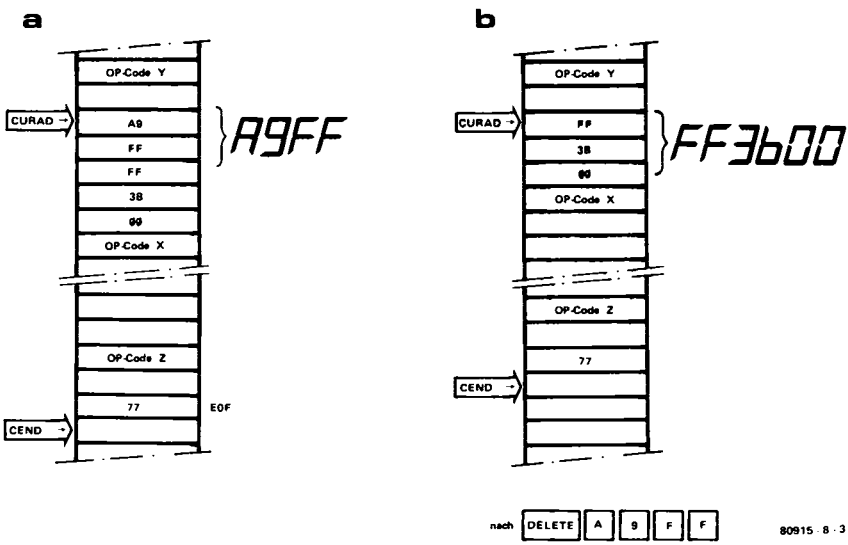


Bild 3. In diesem Beispiel wird die Instruktion LDA # FF oder A9 FF mit dem DELETE-Kommando aus der File entfernt. Dazu wird der Datenblock zwischen CURAD + 2 und CEND um zwei Bytes nach oben geschoben. Somit wird die Instruktion LDA # FF durch den Label FF 38 00 überschrieben.

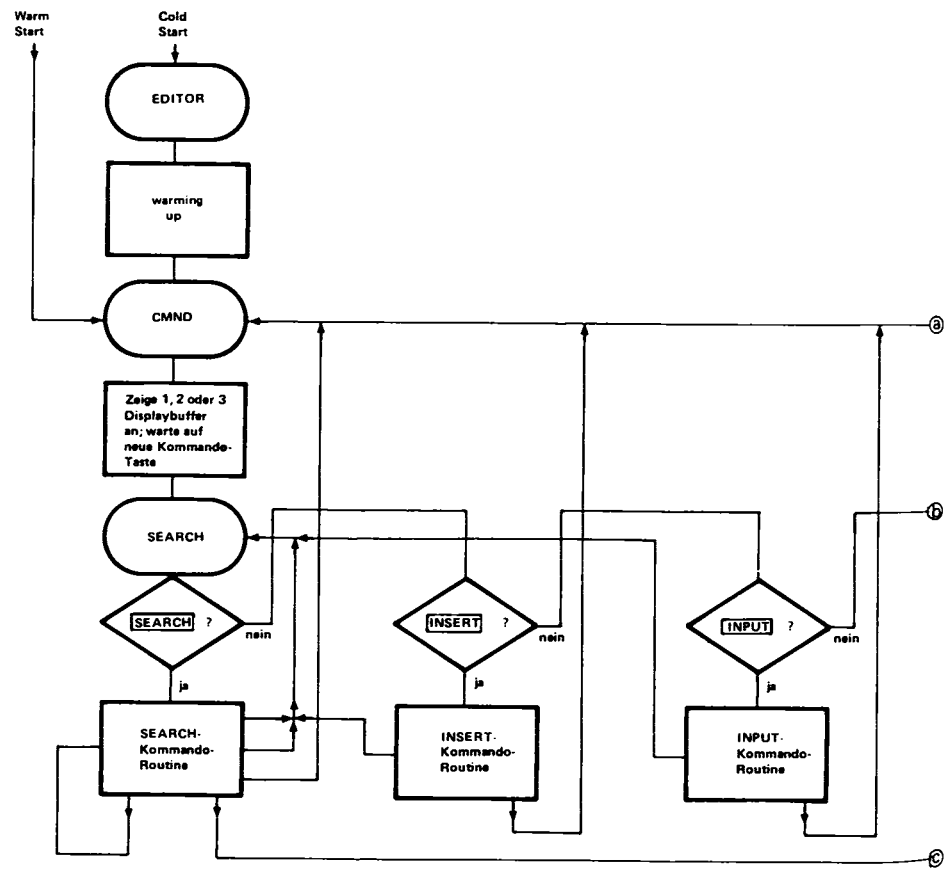
diesem Fall ein Label) mit der INSERT-Taste. Der Label FF 15 ØØ steht jetzt vor dem EOF-Charakter 77. Da ein Label drei Bytes lang ist, ist der EOF-Charakter um drei Bytes nach unten verschoben. Zu diesem Zeitpunkt enthält die File eine Instruktion FF 15 ØØ und den EOF-Charakter 77. Dabei beginnt die File bei der Adresse BEGAD und endet an der Adresse CEND. Je mehr Instruktionen in der File abgelegt werden, desto weiter wandert der EOF-Charakter 77 nach unten. Auch der CEND-Pointer wandert vom BEGAD-Pointer immer weiter weg, was ein Zeichen dafür ist, daß die File immer größer wird.
Bild 3 zeigt den Effekt des DELETE-Kommandos. Dabei steht an irgendeiner Stelle in der File die Instruktion LDA # FF, gefolgt vom Label FF 3B ØØ. Wir wollen die Instruktion LDA # FF aus der File entfernen. Dazu spüren wir diese Instruktion mit einem SEARCH-Kommando auf (weitere Möglichkeit siehe Kapitel 5). Nachdem das Display die Instruktion LDA # FF mit A9 FF anzeigt, können wir die DELETE-Taste drücken. Beim Drücken der DELETE-Taste stellt sich folgende Situation ein:

* Die Instruktion LDA # FF ist zwei Byte lang. Hinter A9 FF steht der Label FF 3B ØØ.
* A9 wird aus der File geholt, indem der Computer den Datenblock, beginnend an der Adresse CURAD+2 und endend an der Adresse CEND um zwei Bytes nach oben schiebt. Die Instruktion A9 ØØ wird also vom folgenden Datenblock überschrieben, und verschwindet somit aus der File.
* Nach der Verschiebe-Operation wird auch der CEND-Pointer um zwei Stellen nach oben geschoben. Somit ist nach dem DELETE-Kommando die File um zwei Bytes kürzer geworden.

Das globale Flußdiagramm des Editors

Das globale Flußdiagramm des Editors zeigt **Bild 4**. Dieses Flußdiagramm ist mit dem globalen Flußdiagramm des Monitorprogramms (Bild 1, Kapitel 7) vergleichbar. Zwischen beiden globalen Flußdiagrammen bestehen Ähnlichkeiten und Verschiedenheiten. Zunächst die Ähnlichkeiten der beiden Flußdiagramme: Die Vorbereitung des Computers, in Editor-Mode zu arbeiten (warming up), ist vergleichbar mit dem RESET-Programm des Monitors. Weiter ist der zentrale Label CMND beim Editor vergleichbar mit dem Label START beim Monitor. Auch die einzelnen Aktionen, die der Junior-Computer zwischen den Labels CMND und SEARCH beim Editor ausführt, sind vergleichbar mit Block "C" des Monitor-Programms: In beiden Blöcken steuert der Computer das Display an und fragt das Tastenfeld ab. Nach dem Label SEARCH in Bild 4 fragt der Computer alle Kommando-Tasten ab und erkennt so, ob der Programmierer ein INSERT-, ein INPUT-, ein SEARCH-, ein SKIP-, oder ein DELETE-Kommando ausführen will.
Damit sind alle Ähnlichkeiten zwischen Editor- und Monitor-Flußdiagramm aufgezählt. Nun beschreiben wir die Verschiedenheiten zwischen dem globalen Flußdiagramm des Editors und dem globalen Flußdiagramm des Monitors: Das Editor-Programm hat im Gegensatz zum Monitor-Programm keinen deutlichen Ausgang. Ist man einmal via Cold- oder Warm Start Entry in den Editor gesprungen, kann man diesen nicht mit einer Kommando-

Taste wie den Monitor (GO-Taste) verlassen. Der Editor ist für den Computer eine unendliche Programmschleife und kann nur über die Hardware-Tasten RST und ST verlassen werden.
Ein weiterer Unterschied ist der Label ERRA. Dieser Label gehört zu einem Programmteil des Editors, der auf dem Display EEEEEE anzeigt, wenn der Programmierer irgendeinen Bedienungsfehler macht. Die "Intelligenz" des Monitors ist so niedrig und elementar, daß man auf eine Fehlermeldung gänzlich verzichten kann. Beim Editen können bei der Eingabe von Instruktion leicht Fehler auftreten. Der Computer muß dann dem Programmierer mitteilen, daß ein Bedienungsfehler vorliegt. Fehlermeldungen treten bei folgenden Ereignissen auf:
* Der Programmierer sucht mit dem SEARCH-Kommando ein bestimmtes



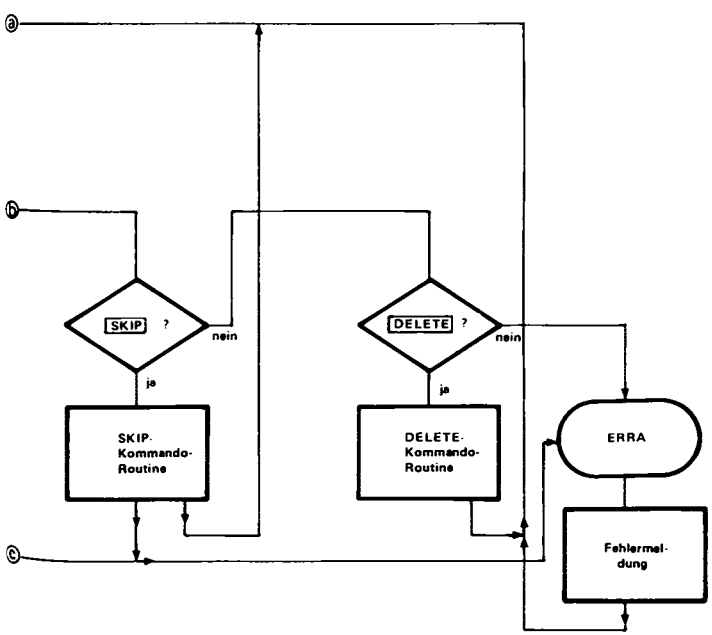
Zwei-Byte-Muster in der File auf. Ist das aufzuspürende Zwei-Byte-Muster nicht in der File vorhanden, dann zeigt der Junior-Computer EEEEEE auf dem Display, solange die SEARCH-Taste gedrückt ist.

*Der Programmierer durchläuft mit dem SKIP-Kommando eine File. Schließlich ist er beim EOF-Charakter angekommen und betätigt weiterhin die SKIP-Taste. Hinter dem EOF-Charakter ist bekanntlich die File zu Ende. Deshalb zeigt auch in diesem Fall der Junior-Computer EEEEEE auf dem Display an. Der Programmierer weiß also, daß er sich außerhalb der File befindet, die von BEGAD und CEND begrenzt ist.

*Der Computer macht auch eine Fehlermeldung, wenn der Programmierer eine falsche Taste betätigt. Das ist dann der Fall, wenn der Junior-Computer auf eine Kommandotaste wartet, der Programmierer jedoch eine Datentaste drückt.

Es besteht noch ein weiterer Unterschied zum globalen Flußdiagramm des Monitors. In **Bild 4** werden zwar die Kommando-Tasten getestet, wo verarbeitet jedoch der Computer die Daten-Tasten Ø . . . F? Daten-Tasten dienen bekanntlich zur Eingabe von Instruktionen, also zur Eingabe von OP-Codes und Operanden-Bytes.

Bild 4. Das große Flußdiagramm des Editors. Aus diesem Flußdiagramm ist klar ersichtlich, wie die fünf Kommando-Tasten des Editors ausgefiltert werden. Danach durchläuft der Computer für jede Kommando-Taste eine Kommando-Routine, um anschließend wieder zum zentralen Punkt CMND zu verzweigen. Macht der Programmierer einen Bedienungsfehler, verzweigt der Editor zum Label ERRA und gibt eine Fehlermeldung.



80915-8-4

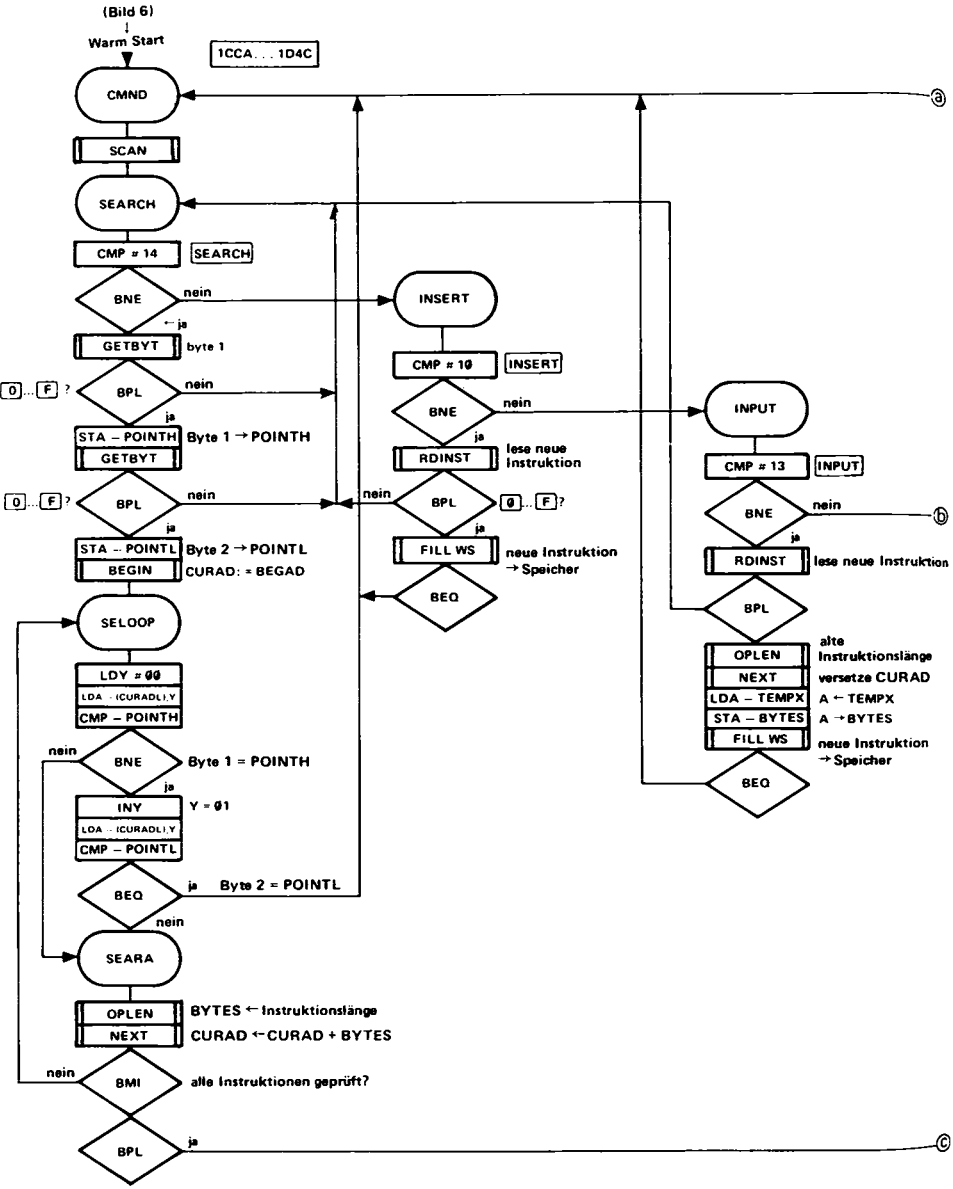
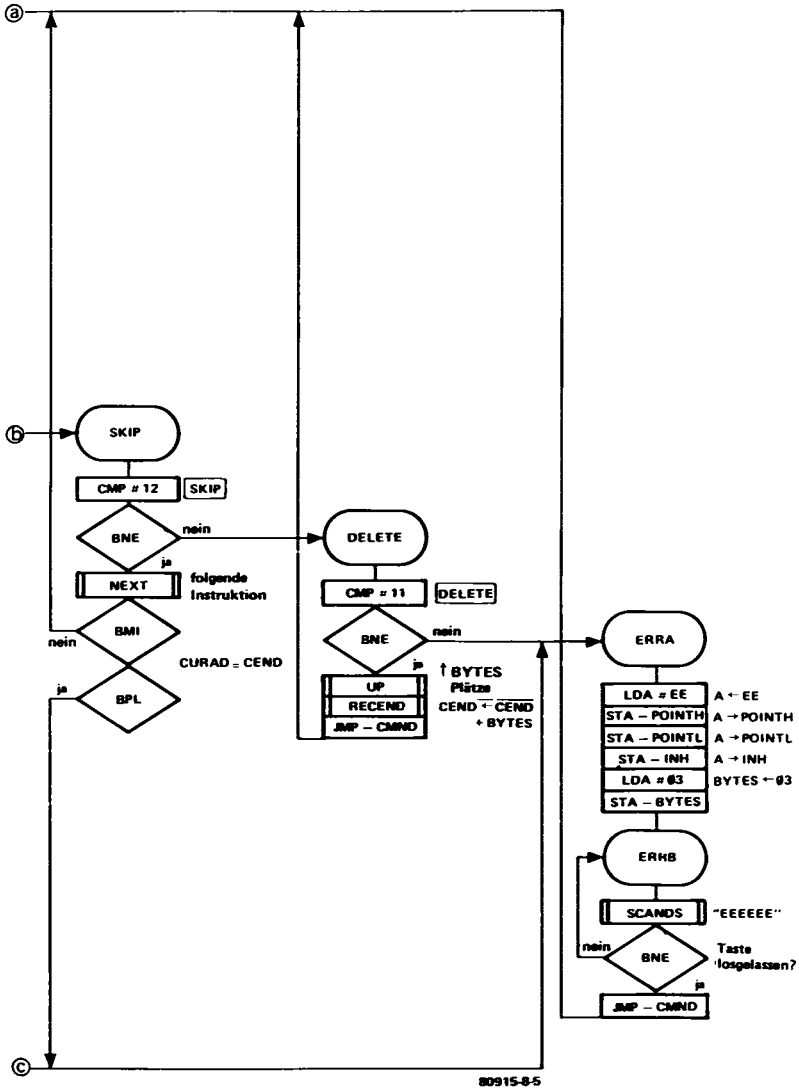


Bild 5. Das detaillierte Flußdiagramm der Editor-Hauptroutine. Dabei ist CMND der Label für Warm Start Entry.


ⓐ
ⓑ
ⓒ
SKIP
CMP # 12
SKIP
BNE
nein
ja
NEXT
folgende Instruktion
BMI
nein
CURAD = CEND
ja
BPL
DELETE
CMP # 11
DELETE
BNE
nein
ja
UP
RECEND
JMP – CMND
↑ BYTES Plätze
CEND ← CEND + BYTES
ERRA
LDA # EE
A ← EE
STA – POINTH
A → POINTH
STA – POINTL
A → POINTL
STA – INH
A → INH
LDA # Ø3
BYTES ← Ø3
STA – BYTES
ERRB
SCANDS
"EEEEEE"
BNE
nein
Taste losgelassen?
ja
JMP – CMND
80915-8-5

Wie aus dem globalen Flußdiagramm (Bild 4) ersichtlich ist, behandelt der Junior-Computer die Daten-Tasten in den Routinen SEARCH, INSERT und INPUT. Während dieser Routinen nimmt die aus Buch 1 bekannte Subroutine GETBYT Daten-Tasten in Empfang und baut zwei hintereinander gedrückte Daten-Tasten zu einem Byte zusammen. Weiter akzeptiert die Subroutine GETBYT nur Daten-Tasten, ignoriert jedoch Kommando-Tasten! Das hat den Effekt, daß der Programmierer **während** der Daten-Eingabe (= Eintippen einer Instruktion) zur Kommando-Eingabe (= Drücken einer der Kommando-Tasten INSERT, INPUT, SEARCH, SKIP und DELETE) übergehen kann, ohne fehlerhafte Daten in der File abzulegen. Das ist leicht einzusehen, da wir aus Kapitel 5 wissen, daß der Display-buffer und die File voneinander getrennt sind. Der Inhalt der drei Display-buffer wird erst dann vom Display in die File kopiert, wenn der Programmierer die gesamte Instruktion eingegeben hat. Drückt also der Programmierer während der Eingabe einer Instruktion eine Kommando-Taste, verzweigt das Editor in **Bild 4** zum Label SEARCH und kümmert sich danach um die neue Kommando-Taste. Hat aber der Programmierer die ganze Instruktion in den Junior-Computer eingegeben, verzweigt das Programm zum zentralen Punkt SEARCH. Der Editor wartet auf die Eingabe einer neuen Kommando-Taste und verzweigt dann wieder zu einer der Tasten-Routinen SEARCH, INSERT, INPUT, SKIP oder DELETE.

Das detaillierte Flußdiagramm des Editors

Nachdem das globale Flußdiagramm des Editors bekannt ist, wenden wir uns dem detaillierten Flußdiagramm zu. **Bild 5** zeigt das detaillierte Flußdiagramm des Editors ohne die Initialisierungs-Routine bei Cold Start Entry. Diese Initialisierungs-Routine, die den Arbeitsspeicher des Junior-Computers auf die Dateneingabe vorbereitet, zeigt **Bild 6**. Wie aus diesen beiden Bildern ersichtlich ist, setzt sich das Editor-Programm aus vielen Subroutinen zusammen, die wir nach und nach bei der Programmbesprechung kennen lernen. Übrigens kann auch der Programmierer die zahlreichen Editor-Subroutinen in seine Programme einbauen und somit viel Zeit bei der Entwicklung eines Programms sparen.

Cold Start Entry

Bild 6 zeigt das detaillierte Flußdiagramm für den Editorstart via Cold Start Entry. Das Editor-Programm beginnt mit der Subroutine BEGIN, die **Bild 7** zeigt. Die Subroutine BEGIN besteht aus nur vier Instruktionen, ist jedoch für den Editor und Assembler enorm wichtig. Sie sorgt dafür, daß der Display-Pointer CURAD mit dem Adreß-Pointer BEGAD zusammen fällt. Der Adreß-Pointer BEGAD zeigt bekanntlich auf die Beginnadresse der File, in der der Programmierer Instruktionen und Labels ablegen möchte. Nach der Rückkehr von der Subroutine BEGIN macht der Computer mit Hilfe des X- und Y-Registers den variablen Endadreß-Pointer CEND = BEGAD + 1. Das ist logisch, wenn man bedenkt, daß der Speicherplatz, auf den der Display-Pointer CURAD zeigt, mit 77 geladen wird, und CEND auf die Speicherzelle zeigt, die dem EOF-Charakter folgt.
Wie wird CEND = BEGAD + 1 realisiert? Zuerst inkrementiert man BEGAD

mit eins. War BEGADL vor der Inkrementierung FF, ist der Inhalt dieser Speicherzelle nach der Inkrementierung ØØ. Somit muß auch BEGADH um eins inkrementiert werden. Diese Situation testet der Computer mit einer BNE-Instruktion, was zu einer eventuellen Inkrementierung des Y-Registers führt. Da sich BEGAD während des gesamten Editor-Programms nicht ändern darf, verwenden wir das X- und Y-Register als temporären Zwischenspeicher. Die beiden folgenden Instruktionen STX- und STY- sorgen dafür, daß CEND = BEGAD + 1 ist. Die drei letzten Instruktionen von diesem Programmteil haben die Aufgabe, in der Speicherzelle, auf die CURAD zeigt, den EOF-Charakter 77 abzulegen. Somit ist die Ausgangssituation von **Bild 2a** hergestellt.

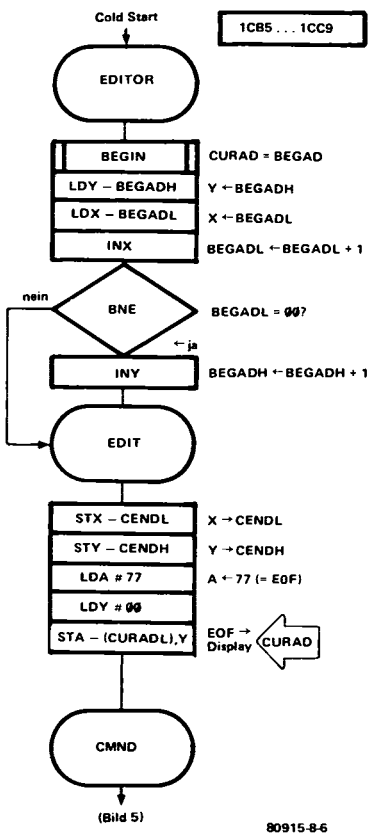


Bild 6. Bei Cold Start Entry muß der Arbeitsspeicher auf die Dateneingabe vorbereitet werden. Die Pointer CURAD sowie CEND setzt der Computer selbstständig auf die richtige Adresse. Auch der EOF-Charakter wird in die File geschrieben.

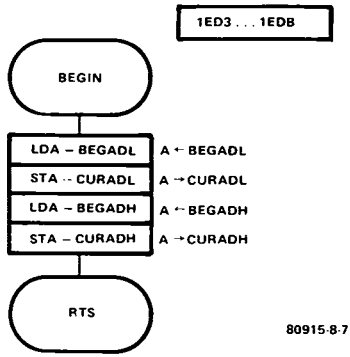


Bild 7. Die Subroutine BEGIN setzt den Display-Pointer CURAD auf die Adresse, auf die der Pointer BEGAD zeigt, also: CURAD = BEGAD.

Display und Tasten des Editors

Der Programmteil zwischen den Labels CMND und SEARCH in den **Bildern 4** und **5** beschränkt sich ausschließlich auf die Steuerung des Displays und die Abfrage des Tastenfeldes. In diesem Programmteil bestimmt der Computer ob ein, zwei oder alle drei Displaybuffer auf dem Display abgebildet werden sollen und ob irgendeine Taste im Tastenfeld des Junior-Computers gedrückt ist. Nachdem irgendeine gedrückte Taste festgestellt ist, berechnet der Computer in gewohnter Weise den Tastenwert der gedrückten Taste. All diese beschriebenen Aufgaben führt der Computer in der Subroutine SCAND aus, deren detailliertes Flußdiagramm **Bild 8** zeigt.

Am Anfang dieser Subroutine wird das X-Register mit Ø2 und das Y-Register mit ØØ geladen. Das hat zur Folge, daß ab dem Label FILBUF der Inhalt von drei aufeinanderfolgenden Speicherzellen aus der File in die drei Displaybuffer POINTH, POINTL und INH kopiert wird:

1. X = Ø2, Y = ØØ: kopiere den Inhalt der Speicherzelle, auf die der Display-Pointer CURAD zeigt, in den Displaybuffer POINTH.
2. X = Ø1, Y = Ø1: Kopiere den Inhalt der Speicherzelle CURAD + 1 in den Displaybuffer POINTL.
3. X = ØØ, Y = Ø2: kopiere den Inhalt der Speicherzelle CURAD + 2 in den Displaybuffer INH.
4. X = FF: die folgende BPL-Instruktion führt zu keiner Verzweigung mehr und der Computer gelangt zur Subroutine OPLEN. Diese Subroutine ist fast identisch mit der Subroutine LENACC aus Buch 1. Die Subroutine OPLEN berechnet die Länge der Instruktion, auf deren OP-Code der Display-Pointer CURAD zeigt. Nach der Rückkehr von der Subroutine OPLEN steht die Länge der Instruktion, die auf dem Display angezeigt werden soll, in der RAM-Zelle BYTES (Adresse: ØØF6). Ist die Instruktions-

länge der auf dem Display anzuzeigenden Instruktion bekannt, weiß der Computer, ob er mit Hilfe der Subroutine SCANDS allein den Inhalt von POINTH, den Inhalt von POINTH und POINTL oder den Inhalt von POINTH, POINTL und INH auf dem Display anzeigen soll. Dabei enthalten die drei Displaybuffer folgende Daten:
POINTH: enthält den OP-Code der Instruktion;
POINTL: enthält das erste Operanden-Byte;
INH: enthält das zweite Operanden-Byte.
Wie bereits erwähnt, übernimmt die Subroutine SCAND die Steuerung des Displays und die Abfrage des Tastenfeldes. SCANDS ist eine bekannte Subroutine aus Kapitel 7, nämlich die Subroutine SCAND minus dem ersten Programmabschnitt. In diesem ersten Programmabschnitt wird der Displaybuffer INH mit den Daten geladen, die in der Speicherzelle stehen, auf die der Adreß-Pointer POINTH, POINTL zeigt. Dieser Programmabschnitt in SCAND benötigt der Computer nicht im Editor-Programm. Im Prinzip ist SCANDS mit all seinen Subroutinen aus Kapitel 7 bekannt. Wer die genaue Funktionsweise dieser wichtigen Subroutine vergessen hat, kann dort nochmals nachschlagen. Das Wesentliche aus der Subroutine SCANDS ist nachfolgend kurz zusammengefaßt:

SCANDS übernimmt zusammen mit der Speicherzelle BYTES die gesamte Steuerung des Displays. Ist der Inhalt von BYTES =Ø1, zeigt der Computer nur den Inhalt des Displaybuffers POINTH auf dem Display an. Die anzuzeigende Instruktion hat eine Länge von einem Byte und die vier rechten Displays bleiben dunkel. Ist der Inhalt von BYTES =Ø2, zeigt der Computer den Inhalt der Displaybuffer POINTH und POINTL auf den Display an. Die anzuzeigende Instruktion hat eine Länge von zwei Bytes und die zwei rechten Displays bleiben dunkel. Ist aber der Inhalt von BYTES =Ø3, zeigt der Computer den Inhalt der Displaybuffer POINTH, POINTL und INH auf dem Display an. Die anzuzeigende Instruktion hat eine Länge von drei Bytes und alle sechs Displays leuchten auf.
Am Ende der Subroutine SCANDS testet der Computer, ob irgendeine Taste im Tastenfeld gedrückt ist. Wenn ja, ist nach der Rückkehr aus der Subroutine SCANDS der Accu-Inhalt ungleich null. Ist dagegen keine Taste gedrückt, ist der Accu-Inhalt gleich null. Das sollte man sich bei der folgenden Besprechung des Editor-Programms ständig vor Augen halten. Nach der Subroutine OPLEN (Bild 8) gelangen wir zum Label SCANA. Der Programmabschnitt zwischen dem Label SCANA und dem abschließenden RTS-Befehl kennen wir noch aus Kapitel 7. Der erste Programmabschnitt SCANA-BNE-SCANA wird solange durchlaufen, bis die letzte gedrückte Taste losgelassen ist. In dieser Programmschleife zeigt der Computer bereits einen Teil der eingegebenen Instruktion oder die gesamte Instruktion auf dem Display an.
Ist die Taste losgelassen, dann kommen wir zum Label SCANB. In der Programmschleife SCANB-BEQ-SCANB hält sich der Computer die meiste Zeit auf. In dieser Programmschleife wartet er solange, bis der Programmierer eine neue Taste niederdrückt. Ist eine neue Taste gedrückt, wird die Subroutine SCANDS abermals aufgerufen. Wie aus Kapitel 7 noch bekannt ist, hat dieser zweite Aufruf der Subroutine SCANDS eine Tastenprellung zur Folge. Ist nach der Rückkehr von SCANDS die Taste noch immer gedrückt, erfolgt in der anschließenden Subroutine GETKEY

die Berechnung des Tastenwertes. Die Subroutine SCAN verläßt der Computer nach der Rückkehr aus der Subroutine GETKEY mit dem Tastenwert im Accu: Dabei haben die Editor-Kommando-Tasten die folgenden Tastenwerte:
SEARCH: Tastenwert 14 (wie PC-Taste)
INSERT: Tastenwert 1Ø (wie AD-Taste)
INPUT: Tastenwert 13 (wie GO-Taste)
SKIP: Tastenwert 12 (wie + Taste)
DELETE: Tastenwert 11 (wie DA-Taste).
Das Testen einer bestimmten Kommando-Taste geschieht mit einer CMP-Instruktion, gefolgt von einer BNE-Instruktion. Ist eine Kommando-Taste gefunden, dann findet die Programmverzweigung bei der folgenden BNE-Instruktion nicht statt und der Computer kümmert sich um die Kommando-Taste (siehe auch Bild 4 und 5).

Die Fehlermeldung EEEEEE

Der Programmteil ab dem Label ERRA in **Bild 5** tritt dann in Funktion, wenn der Anlaß zu irgendeiner Fehlermeldung besteht, wenn also ein Bedienungsfehler vorliegt. Wann eine Fehlermeldung ausgelöst wird, haben wir bereits ausführlich besprochen. Die ersten Instruktionen hinter dem Label ERRA laden die drei Displaybuffer POINTH, POINTL und INH mit EE. Da alle sechs Displays bei einer Fehlermeldung aufleuchten sollen, lädt die Speicherzelle BYTES den Wert Ø3. Der anschließende Programmteil hinter ERRB besteht aus einer Warteschleife, bei der die Display- und Keyboard-Routine SCANDS durchlaufen wird. Die Fehlermeldung EEEEEE bleibt deshalb solange auf dem Display stehen, bis die Taste, die die Fehlermeldung ausgelöst hat, losgelassen ist. Nach dem Loslassen dieser Taste verzweigt das Programm zurück zum zentralen Punkt CMND.

Die SEARCH-Routine

Hinter dem Label SEARCH startet eine Routine, die der Computer jedesmal durchläuft, wenn der Programmierer die SEARCH-Kommando-Taste drückt. Wie wir wissen, kehrt der Computer aus der Subroutine SCAN mit dem Tastenwert einer Befehls-Taste im Accu zurück. Mit den beiden Instruktionen CMP # 14 und BNE wird die SEARCH-Taste ausgefiltert. Das zum SEARCH-Kommando gehörende Programmteil muß folgende Bedingungen erfüllen:

1. Mit dem SEARCH-Kommando muß sich in der File ein beliebiges Zwei-Byte-Muster aufsuchen lassen.
2. Dabei soll die Suchaktion bei BEGAD starten und sobald das gewünschte Zwei-Byte-Muster gefunden ist, soll die Suchaktion beendet werden.
3. Ist das Zwei-Byte-Muster in der File zwischen BEGAD und CEND nicht vorhanden, soll der Computer eine Fehlermeldung geben.

Am Anfang der SEARCH-Routine treffen wir zweimal auf die Subroutine GETBYT. Das ist eine bereits aus Buch 1 bekannte Subroutine. Bisher wissen wir zwar, welchen Effekt die Subroutine GETBYT hat, was sich jedoch in ihr abspielt, ist noch nicht besprochen. Das wollen wir jetzt nachholen!

Die Subroutine GETBYT

Das detaillierte Flußdiagramm der Subroutine GETBYT zeigt **Bild 9**. Die Aufgabe dieser Subroutine ist, zwei hintereinander gedrückte Daten-Tasten zu einem Byte im Accu zusammenzusetzen. GETBYT akzeptiert nur Daten-Tasten, verweigert aber Kommando-Tasten.

Die beiden gedrückten Daten-Tasten werden wie folgt im Accu zusammengesetzt: Der Tastenwert der ersten gedrückten Daten-Taste wandert in das hochwertige Accu-Nibble und der Tastenwert der zweiten gedrückten Daten-Taste wandert in das niederwertige Accu-Nibble. Am Anfang der Subroutine GETBYT wird die Subroutine SCANA aufgerufen. Das ist der zweite Teil der zuvor besprochenen Subroutine SCAND in **Bild 8**. Ab dem Label SCANA in Bild 8 führt der Computer folgende Aufgaben aus:

* Er stellt fest, ob die zuvor gedrückte Taste (alte Taste) schon losgelassen ist.
* Ferner wird festgestellt, ob der Programmierer bereits eine weitere (neue) Taste gedrückt hat.
* Ist eine neue Taste gedrückt, berechnet der Computer den Tastenwert durch Aufrufen der Subroutine GETKEY.
* Während die beschriebenen Aktionen ablaufen, steuert der Computer das Display an und fragt das Tastenfeld ab. Welcher der drei Displaybuffer POINTH, POINTL oder INH auf dem Display erscheinen sollen, bestimmt wieder der Inhalt der Speicherzelle BYTES.

Kommando-Tasten haben einen Tastenwert von größer oder gleich 1Ø. Daten-Tasten haben dagegen einen Tastenwert von kleiner oder gleich ØF. Mit den beiden Instruktionen CMP # 1Ø und BPL lassen sich somit die Kommando-Tasten von den Daten-Tasten unterscheiden. Wird während der Subroutine GETBYT eine beliebige Kommando-Taste gedrückt, dann ist nach der CMP # 1Ø Instruktion die N-Flag N=Ø und das Programm verzweigt über den RTS-Befehl zur SEARCH-Routine.

Ist aber während der Subroutine GETBYT eine Daten-Taste gedrückt, kommt der Computer aus der Subroutine SCANA mit dem Wert der Daten-Taste im Accu zurück. War es die erste Daten-Taste, ändert sich der Accuinhalt nach einer viermaligen Schiebeoperation (4 mal ASL-A) von ØX nach XØ. Dabei ist X der Wert der ersten gedrückten Daten-Taste.

Für das Lesen der zweiten Daten-Taste wird abermals die Subroutine SCANA aufgerufen. Während dieser Subroutine fungiert der Accu als Arbeitsregister. Deshalb muß vor dem zweiten Aufruf der Subroutine SCANA der Accu in die Speicherzelle NIBBLE gerettet werden. Erst dann wird SCANA abermals aufgerufen und danach mit den Instruktionen CMP # 1Ø und BPL festgestellt, ob eine Daten- oder Kommando-Taste gedrückt ist. Eine eventuelle gedrückte Kommando-Taste wird auch an dieser Stelle während der Subroutine GETBYT ignoriert und das Programm verzweigt mit N=Ø zur SEARCH-Routine zurück.

War aber die zweite Taste eine Daten-Taste mit dem Wert Y, erhalten wir nach der Instruktion ORA-NIBBLE ein Datenbyte von XY. Dabei ist X der Tastenwert der ersten Daten-Taste und Y der Tastenwert der zweiten Daten-Taste. Am Ende der Subroutine GETBYT wird durch die Instruktion LDX # FF die N-Flag N=1 gesetzt. Somit läßt sich nach dem RTS-Befehl feststellen, ob während der Subroutine GETBYT eine Kommando-Taste gedrückt wurde.

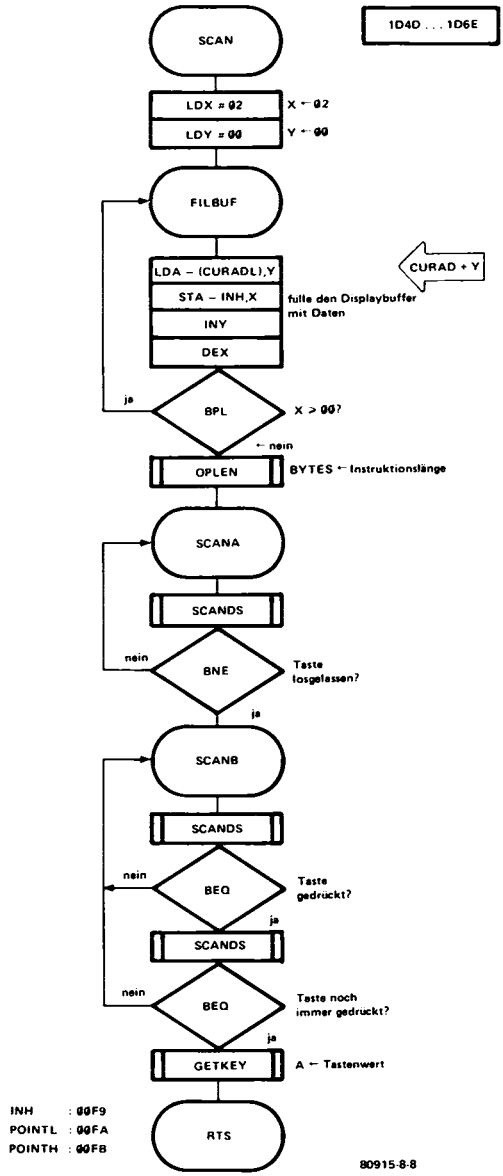


Bild 8. Die Subroutine SCAN füllt den Displaybuffer POINTH, POINTL und INH mit dem OP-Code und dem Operanden einer Instruktion auf. Anschließend zeigt sie diese Instruktion auf dem Display an. Das Display hat dabei eine variable Länge. Gleichzeitig berechnet SCAN den Wert einer Taste, falls irgendeine Taste auf dem Tastenfeld gedrückt ist. Für die Steuerung des Displays und die Abfrage des Tastenfeldes finden die Monitor-Subroutinen SCANDS und GETKEY Verwendung.

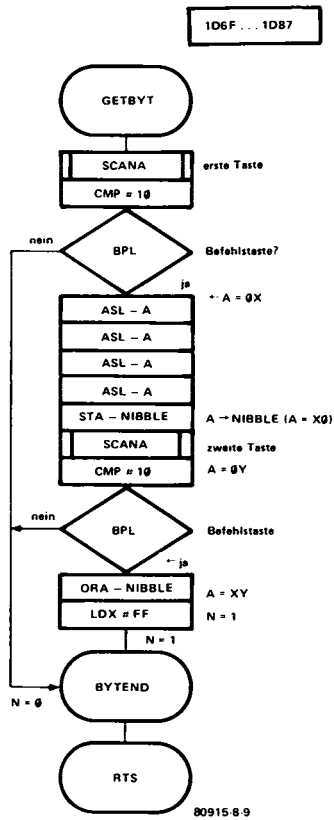


Bild 9. Die Subroutine GETBYT füllt den Accu mit den Tastenwerten von zwei hintereinander gedrückten Tasten. Die N-Flag ist dabei gesetzt, wenn nur Daten-Tasten betätigt wurden. Kommando-Tasten ignoriert die Subroutine GETBYT, indem sie nach der Rückkehr in die Hauptroutine die N-Flag zurücksetzt.

Nebenbei noch eine Bemerkung: GETBYT ist eine sehr universelle Subroutine. Der Programmierer stellt sicher bald fest, wie nützlich diese Subroutine beim Entwickeln von "Do It Yourself Software" in eigenen Programmen ist.
Doch nun wieder zurück zur SEARCH-Routine in **Bild 5**! Nach dem ersten Aufruf der Subroutine GETBYT ist die N-Flag N=Ø, wenn während dieser Subroutine eine Kommando-Taste gedrückt wurde. Waren aber zwei Daten-Tasten gedrückt, ist die N-Flag N=1. Somit läßt sich nach der Rückkehr von GETBYT leicht feststellen, ob via einer BPL-Instruktion zum Label SEARCH zurückverzweigt wird oder nicht.
Waren während der Subroutine GETBYT zwei Daten-Tasten gedrückt, sind diese beiden Daten-Tasten der erste Teil des aufzuspürenden Zwei-Byte-Musters. Dieser erster Teil des Zwei-Byte-Musters wandert in den Display-

buffer POINTH und wird während des zweiten Aufrufs der Subroutine GETBYT auf dem Display angezeigt. Während sich der Computer wieder in GETBYT befindet, wartet er auf die Eingabe des zweiten Teiles des aufzuspürenden Zwei-Byte-Musters. Ist beim zweiten Aufruf wieder eine Kommando-Taste gedrückt, verzweigt der Computer zum Label SEARCH und wartet auf die Eingabe eines neuen Zwei-Byte-Musters. Hat man aber zwei Daten-Tasten gedrückt, dann sind diese beiden Tasten der zweite Teil des aufzuspürenden Zwei-Byte-Musters. Der Wert dieser beiden Daten-Tasten gelangt zum Displaybuffer POINTL. Die beiden Displaybuffer POINTH und POINTL sind somit Datenbuffer für das aufzuspürende Zwei-Byte-Muster.
Das Zwei-Byte-Muster, das in der File zwischen BEGAD und CEND aufgespürt werden soll, ist somit bekannt und die Suchaktion kann beginnen. Am Anfang dieser Aktion wird die Subroutine BEGIN aufgerufen und CURAD = BEGAD gemacht. Der folgende Programmteil der SEARCH-Routine wird ab dem Label SELOOP sofort durchlaufen, bis das gewünschte Zwei-Byte-Muster in der File gefunden ist. Ist das Zwei-Byte-Muster gefunden, verzweigt das Programm zurück zum zentralen Punkt CMND und zeigt dieses in der folgenden Subroutine SCAN auf dem Display an. Es ist jedoch auch möglich, daß das Zwei-Byte-Muster nicht in der von den beiden Adreß-Pointern BEGAD und CEND begrenzten File enthalten ist. Dann verzweigt die Suchroutine hinter dem Label SELOOP zum Label ERRA und macht somit eine Fehlermeldung.
Wie wird aber die Suchaktion im einzelnen durchgeführt? Dazu vergleicht der Computer alle Instruktionen in der File mit dem Zwei-Byte-Muster, das in den beiden Displaybuffern POINTH und POINTL abgelegt ist. In Worte läßt sich das so fassen:

1. Lade den Accu mit dem OP-Code auf den CURAD zeigt. Vergleiche diesen OP-Code mit dem Inhalt des Displaybuffers POINTH.
2. Ist der Inhalt von POINTH **nicht gleich** dem OP-Code, auf den CURAD zeigt, dann berechne die Instruktionslänge der momentanen Instruktion in der Subroutine OPLEN. Setze den Pointer CURAD um ein, zwei oder drei Speicherplätze weiter nach unten. Man könnte auch sagen, setze CURAD auf den OP-Code der folgenden Instruktion. CURAD wird mit Hilfe der Subroutine OPLEN auf den OP-Code der folgenden Instruktion gesetzt. Während der Subroutine NEXT prüft auch der Computer, ob CURAD schon über CEND hinausgewachsen ist. Tritt diese Situation ein, verzweigt das Programm zum Label ERRA und macht eine Fehlermeldung. Das aufzusuchende Zwei-Byte-Muster ist dann in der File nicht vorhanden.
3. Ist der Inhalt von POINTH **gleich** dem OP-Code, auf den CURAD zeigt, dann hole das dem OP-Code folgende Byte in den Accu. Vergleiche jetzt dieses Byte mit dem Inhalt des Displaybuffers POINTL. Stimmt auch das Byte hinter dem OP-Code, auf den CURAD zeigt, mit dem in Displaybuffer POINTL überein, dann ist das Zwei-Byte-Muster gefunden. Die SEARCH-Routine verzweigt jetzt zum zentralen Punkt CMND und zeigt das gefundene Muster an. Mit dem SEARCH-Kommando lassen sich somit Instruktionen mit einer Länge von zwei oder drei Bytes in der File aufspüren.
4. Ist der Inhalt von POINTH **gleich** dem OP-Code, auf den CURAD zeigt

und der Inhalt von POINTL **nicht gleich** dem Inhalt der Speicherzelle CURAD + 1, dann setze die Suchaktion fort. Bestimme wieder die Instruktionslänge der Instruktion, auf die CURAD zeigt (OPLEN) und bewege den Pointer CURAD um zwei oder drei Bytes nach unten. (NEXT) CURAD zeigt jetzt auf den OP-Code der folgenden Instruktion. Wiederhole den Vergleich zwischen der Instruktion, auf deren OP-Code CURAD zeigt und dem Inhalt der Displaybuffer POINTH und POINTL solange, bis das gewünschte Zwei-Byte-Muster gefunden oder bis CURAD größer als der Pointer CEND ist.

Das SEARCH-Kommando vergleicht also vier Bytes miteinander. Verglichen werden die beiden Bytes, die in den Displaybuffern POINTH und POINTL abgelegt sind, mit den beiden Bytes, auf die die Pointer CURAD beziehungsweise CURAD + 1 zeigen. CURAD wird in der Subroutine NEXT auf den OP-Code der folgenden Instruktion eingerichtet. Da bei der Suchaktion indirekt indiziert der Accu geladen wird, ist das Y-Register beim Vergleich zuerst null und anschließend eins. Bei Y=ØØ findet der Vergleich zwischen dem momentanen OP-Code mit dem Inhalt des Displaybuffers POINTH und bei Y=Ø1 zwischen dem ersten Operandenbyte der aufzuspürenden Instruktion (Zwei-Byte-Muster) und dem Inhalt des Displaybuffers POINTL statt. Damit ist der Ablauf der Suchoperation klar. Bei der Suchaktion verzweigt der Computer in die beiden Subroutinen OPLEN und NEXT. Nach der Funktionsbeschriebung beider Subroutinen beschäftigen wir uns jetzt mit dem Aufbau. Und das Byte für Byte.

Die Subroutine NEXT

Das detaillierte Flußdiagramm der Subroutine NEXT zeigt **Bild 10**. Nach dem Zurücksetzen der Carry-Flag wird der Pointer CURAD um den Inhalt der Speicherzelle BYTES erhöht. In der Speicherzelle BYTES steht immer die Länge der auf dem Display angezeigten Instruktion. Die 16-Bit-Addition CURAD + BYTES richtet also den Display-Pointer CURAD auf die folgende Instruktion ein.
Der zweite Teil der Subroutine NEXT prüft, ob der Display-Pointer CURAD schon über den momentanen Endadreß-Pointer CEND hinausgewachsen ist. Diese Prüfung erledigt eine 16-Bit-Subtraktion: CURAD-CEND. Diese Subtraktion beeinflußt die N-Flag wie folgt:

1. N=1: Curad ist größer als CEND; es soll eine Fehlermeldung nach dem RTS-Befehl erfolgen.
2. N=Ø: CURAD ist noch nicht größer als CEND. CURAD bewegt sich somit zwischen BEGAD und CEND. Nach dem RTS-Befehl ist alles in Ordnung und es soll normal im Programm weitergearbeitet werden.

Die Subroutine OPLEN berechnet wieder die Länge der Instruktion, auf deren OP-Code der Pointer CURAD zeigt.
Ein Teil dieser Subroutine ist uns noch aus Buch 1 bekannt. Am Ende dieses Kapitels werden wir nochmals gesondert auf diese interessante Subroutine eingehen.

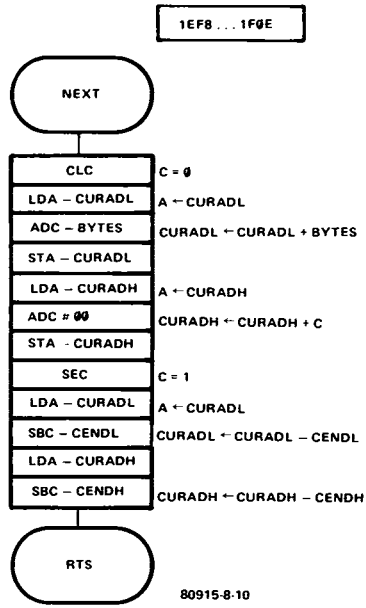


Bild 10. Die Subroutine NEXT setzt den Display-Pointer CURAD um ein, zwei oder drei Adressen nach unten. Ferner wird noch geprüft, ob CURAD schon größer als CEND ist.

Die INSERT-Routine

Für die Besprechung der INSERT-Routine nehmen wir wieder **Bild 5** zur Hand. Die Kommando-Taste INSERT wird durch die beiden Instruktionen CMP # 10 und BNE herausgefiltert. Kommt der Computer aus der Subroutine SCAN mit einem Accuinhalt von 10 in das Editor-Hauptprogramm zurück, läuft anschließend das INSERT-Kommando ab. Das INSERT-Kommando hat die folgende Aufgabe:

1. Lese eine Instruktion vom Keyboard in den Displaybuffer des Junior-Computers. Bestimme dabei die Länge der Instruktion.
2. Ist die gesamte Instruktion in den Displaybuffer eingelesen, dann kopiere diese unmittelbar vor die momentan angezeigte Instruktion in den Arbeitsspeicher.

Um diese beiden Forderungen zu erfüllen, sind im Editor-Programm zwei Subroutinen vorgesehen: Die Subroutine RDINST (= Read Instruction = lese eine Instruktion) und die Subroutine FILLWS (=Fill Workspace = fülle den Arbeitsspeicher auf bzw. kopiere den Displaybuffer in die File). Was hat es nun mit diesen beiden Subroutinen auf sich?

Die Subroutine RDINST

Das detaillierte Flußdiagramm der Subroutine RDINST zeigt **Bild 11.** Die Aufgabe dieser Subroutine ist, eine Instruktion mit einer Länge von ein,

zwei oder drei Bytes in den Displaybuffer des Junior-Computer zu kopieren. Die Subroutine RDINST beginnt mit der Subroutine GETBYT. Dadurch werden zwei Daten-Tasten in den Computer gelesen. Diese beiden Daten-Tasten entsprechen dem OP-Code der einzulesenden Instruktion. Nach der Rückkehr von GETBYT steht der OP-Code im Accu und wird zum Displaybuffer POINTH gebracht. Hat der Programmierer während der Subroutine GETBYT eine Kommando-Taste betätigt, kehrt der Computer in der gewohnten Weise mit gesetzer N-Flag aus der Subroutine RDINST ins Editor-Hauptprogramm zurück.
Nachdem der OP-Code der Instruktion im Displaybuffer POINTH steht, bestimmt der Computer die zu diesem OP-Code gehörende Instruktionslänge. Dazu wird die Subroutine LENACC aufgerufen, die ein Teil der zuvor erwähnten Subroutine OPLEN ist. Beide Subroutinen besprechen wir am Ende dieses Kapitels. Die nun bekannte Instruktionslänge verteilt sich in die drei Speicherzellen BYTES, COUNT und TEMPX.
Nach der Dekrementierung der Speicherzelle COUNT testet der Computer, ob nach dem OP-Code noch Operandenbytes in den Displaybuffer einzulesen sind. Ist die Instruktion nun ein Byte lang, verzweigt das Programm zum Label RDA und das X-Register wird mit FF geladen. Durch diese Operation ist die N-Flag gesetzt. Das hat folgenden Grund:

* Kehrt der Computer aus der Subroutine RDINST mit gesetzter N-Flag zurück, dann ist die gesamte Instruktion eingegeben. Die eingetippte Instruktion steht jetzt im Displaybuffer des Junior-Computers.
* Kehrt der Computer mit zurückgesetzter N-Flag aus der Subroutine RDINST ins Editor-Hauptprogramm zurück, dann hat der Programmierer bei der Eingabe der Instruktion eine Kommando-Taste gedrückt. Ein Teil der eingegebenen Instruktion steht jetzt im Displaybuffer, soll jedoch nicht vom Display zum Arbeitsspeicher (= File) des Junior-Computers gebracht werden.

Durch das Setzen oder Zurücksetzen der N-Flag während der Subroutine RDINST läßt sich nach der Rückkehr ins Hauptprogramm testen, ob die eingegebene Instruktion vom Displaybuffer in den Arbeitsspeicher kopiert werden soll, oder ob sich der Computer um eine neue Kommando-Taste kümmern soll.
Aber zurück zum Einlesen einer Instruktion, die länger als ein Byte ist: Ist nach dem ersten Dekrementieren der Inhalt der Speicherzelle COUNT = ØØ, ist die Instruktion zwei oder drei Bytes lang. Deshalb wird nochmals die Subroutine GETBYT aufgerufen. Beim zweiten Aufruf der Subroutine GETBYT liest der Junior-Computer das erste Operandenbyte der Instruktion in den Displaybuffer. Dabei geht das erste Operandenbyte in dem Displaybuffer POINTL. Dann wird der Inhalt von COUNT abermals dekrementiert. Ist COUNT zu diesem Zeitpunkt gleich null, dann ist die einzulesende Instruktion zwei Bytes lang; der Computer kehrt mit gesetzer N-Flag ins Editor-Hauptprogramm zurück.
Ist aber COUNT immer noch ungleich null, dann ist die einzulesende Instruktion drei Bytes lang. Deshalb wird die Subroutine GETBYT ein drittes Mal aufgerufen. Jetzt kann der Computer das zweite Operandenbyte einlesen. Ist das zweite Operandenbyte eingelesen, gelangt es in den Displaybuffer INH. Danach kehrt der Computer mit gesetzter N-Flag ins Editor-Hauptprogramm zurück.

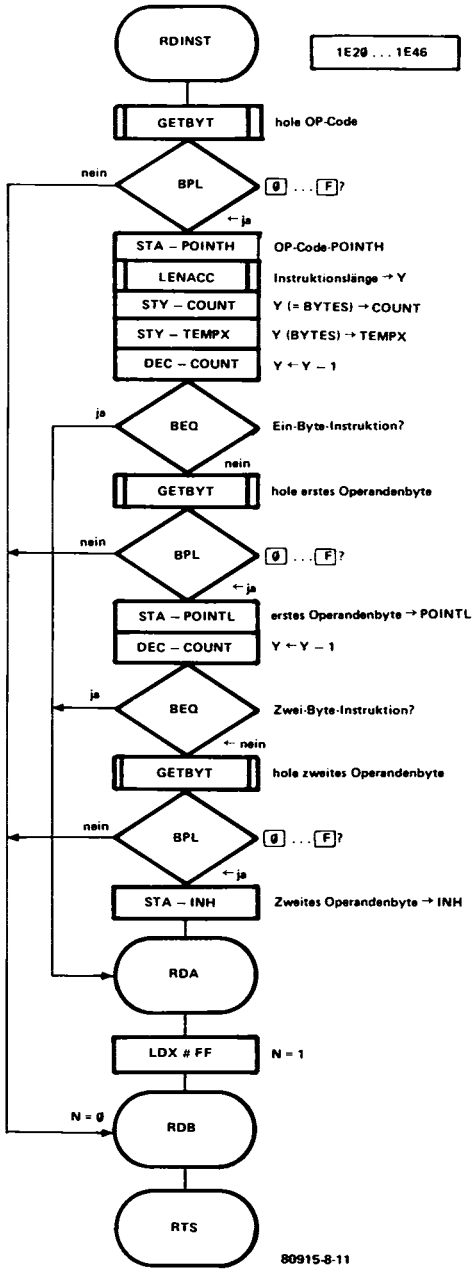


Bild 11. Die Subroutine RDINST liest eine ganze Instruktion vom Keyboard in den Computer. Ist die gesamte Instruktion gelesen, kehrt der Prozessor mit gesetzter N-Flag aus dieser Subroutine zurück.

Die Subroutinen FILLWS und ADCEND

Nehmen wir nochmals **Bild 5** zur Hand! Nach dem Label INSERT filterten wir das INSERT-Kommando aus und sind zur Subroutine RDINST verzweigt. Während dieser Subroutine lasen wir eine Instruktion in den Displaybuffer ein. Aus dieser Subroutine kehrten wir mit gesetzter N-Flag zurück, sobald die gesammte Instruktion im Displaybuffer abgelegt war. Unter dieser Vorraussetzung führt der Prozessor die folgende BPL-Verzweigung nicht aus und gelangt zur Subroutine FILLWS. Diese Subroutine transportiert die eingegebene Instruktion vom Displaybuffer in die File des Junior-Computers und korrigiert dann den Adreß-Pointer CEND. **Bild 12** und **Bild 13** zeigen die detaillierten Flußdiagramme der Subroutinen FILLWS und ADCEND.

Am Anfang der Subroutine FILLWS steht die Subroutine DOWN. Diese relativ komplizierte Subroutine werden wir später noch im Detail be-

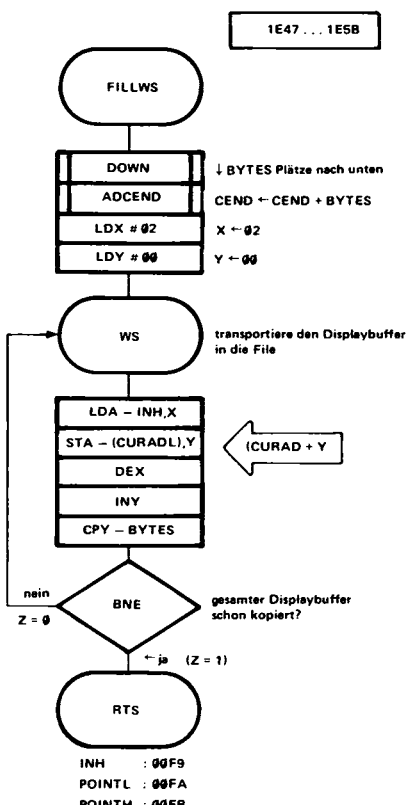


Bild 12. Die Subroutine FILLWS macht in der File Platz, um eine neue Instruktion vom Display in den Arbeitsspeicher zu kopieren.

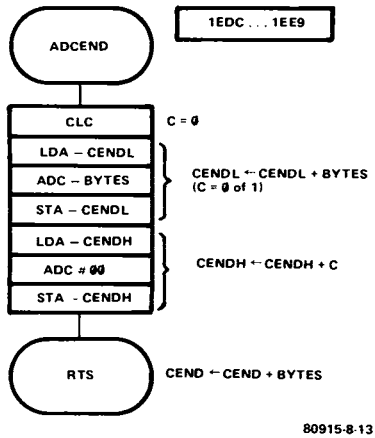


Bild 13. Die Subroutine ADCEND ist ein Unterprogramm der Subroutine FILLWS. ADCEND bewegt den CEND-Pointer um ein, zwei oder drei Adressen nach unten.

sprechen. Was wir jetzt über die Subroutine DOWN wissen müssen, läßt sich in einfache Worte fassen:

1. Beim INSERT-Kommando soll eine neue Instruktion vor der auf dem Display abgebildeten Instruktion eingesetzt werden. Dazu macht im Arbeitsspeicher die Subroutine DOWN Platz.
2. Auf den OP-Code der momentan auf dem Display abgebildeten Instruktion zeigt der Display-Pointer CURAD. Da vor die abgebildete Instruktion eine neue Instruktion mit einer Länge von einem, zwei oder drei Bytes eingefügt werden soll, muß die Subroutine DOWN den Datenblock zwischen CURAD und CEND um ein, zwei oder drei Speicherplätze nach unten bewegen.
3. Nachdem somit im Arbeitsspeicher Platz gemacht ist, kann der Junior-Computer die zuvor eingetippte Instruktion vom Displaybuffer in den Arbeitsspeicher (= File) kopieren.

Soviel müssen wir fürs erste von der Subroutine DOWN wissen, um die Subroutine FILLWS zu verstehen. Bevor bei der INSERT-Routine (**Bild 5**) die Subroutine FILLWS aufgerufen wird, hat der Computer während der Subroutine RDINST die Instruktionslänge der soeben eingegebenen Instruktion berechnet. Diese Instruktionslänge steht vor dem Aufruf der Subroutine FILLWS in der Speicherzelle BYTES. Ist der Computer bei der Subroutine FILLWS angelangt, und dann zur Subroutine DOWN verzweigt, bewegt sich der Datenblock zwischen CURAD und CEND um die Anzahl der Speicherplätze, die in der Speicherzelle BYTES stehen, nach unten. Ist die eingegebene Instruktion beispielsweise zwei Bytes lang, steht in der Speicherzelle BYTES die Hexzahl Ø2. Somit bewegt die Subroutine DOWN den Datenblock zwischen CURAD und CEND um zwei Speicherplätze nach unten.

Jetzt ist klar, wie die Subroutine DOWN Platz macht für eine neue Instruktion, die vom Displaybuffer in die File kopiert wird. Nach der Rückkehr

von der Subroutine DOWN zur Subroutine FILLWS (**Bild 12**) muß noch der Adreß-Pointer CEND neu justiert werden. Diese Aufgabe übernimmt die Subroutine ADCEND (= Advance Current End Address = setze den Pointer CEND um ein, zwei oder drei Bytes nach unten). In der Subroutine ADCEND (**Bild 13**) führt der Computer eine 16-Bit-Addition durch, indem er zum Adreß-Pointer CEND den Inhalt der Speicherzelle BYTES addiert: CEND = CEND + BYTES.
Hinter dem Label WS (**Bild 12**) wird die soeben eingetippte Instruktion vom Displaybuffer in die File kopiert. Am Anfang ist das X-Register X=Ø2 und das Y-Register Y=ØØ. Der folgende Ablauf läßt sich wieder in drei Punkte zusammenfassen:

1. Der Inhalt von POINT (= OP-Code der eingegebenen Instruktion) wird zu der Speicherzelle transportiert, auf die der Display-Pointer CURAD zeigt (X=Ø2, Y=ØØ).
2. In Abhängigkeit der Instruktionslänge (= Inhalt von BYTES) wird der Inhalt von POINTL (= erstes Operandenbyte) zu der Speicherzelle transportiert, auf die CURAD + 1 zeigt (X=Ø1, Y=Ø1).
3. In Abhängigkeit der Instruktionslänge (= Inhalt von BYTES) wird der Inhalt von INH (= zweites Operandenbyte) zu der Speicherzelle transportiert, auf die CURAD + 2 zeigt (X=ØØ, Y=Ø2).

Der Vergleich des Y-Registers mit dem Inhalt der Speicherzelle BYTES gibt darüber Auskunft, ob nach dem Kopieren des OP-Codes aus dem Displaybuffer POINTH noch weitere Operandenbytes aus den Displaybuffern POINTL oder INH in die File transportiert werden sollen.
Sind alle Bytes vom Displaybuffer zur File transportiert, kehrt der Computer aus der Subroutine FILLWS in die INSERT-Routine zurück (**Bild 5**). Beim Verlassen der Subroutine FILLWS ist immer die Z-Flag gesetzt. Deshalb verzweigt das Programm am Ende der INSERT-Routine über eine BEQ-Instruktion zurück zum zentralen Punkt CMND und der Editor wartet wieder auf eine Kommando-Taste. Das INSERT-Kommando ist somit in der Maschine ausgeführt.

Die INPUT-Routine

Die INPUT-Routine ist der INSERT-Routine sehr ähnlich. Es genügt daher, wenn wir uns auf die Unterschiede zwischen beiden Routinen beschränken. Welche Aufgabe muß die INPUT-Routine erfüllen? Erinnern wir uns dazu an Kapitel 5:

1. Lese eine Instruktion vom Keyboard in den Displaybuffer.
2. Setze die soeben gelesene Instruktion hinter die soeben auf dem Display angezeigte Instruktion ein.

Zur Besprechung der INPUT-Routine betrachten wir wieder **Bild 5**. Das INPUT-Kommando filtert die beiden Instruktionen CMP#13 und BNE aus und verzweigt wieder zur bekannten Subroutine RDINS. Diese Subroutine liest vom Tastenfeld eine Instruktion und kopiert diese in den Displaybuffer. Drückt der Programmierer während der Subroutine RDINS eine Kommando-Taste, dann wird die Subroutine RDINS verlassen und die folgende BPL-Instruktion führt den Computer zum zentralen Punkt CMND.
Ist aber die gesamte Instruktion in den Displaybuffer eingegeben, folgt

ein Sprung zur Subroutine OPLEN. Der Display-Pointer CURAD zeigt in diesem Moment noch auf die alte Instruktion in der File. Somit berechnet der Computer in der Subroutine OPLEN nochmals die Länge der zuvor auf dem Display angezeigten Instruktion. Das Aufrufen der Subroutine NEXT bewegt den Display-Pointer CURAD um die Länge der alten auf dem Display angezeigten Instruktion nach unten. CURAD zeigt auf die Adresse, an der in der File die neue Instruktion eingesetzt werden soll. Nach der Rückkehr von der Subroutine NEXT wird der Inhalt der Speicherzelle TEMPX zur Speicherzelle BYTES transportiert. BYTES gibt bekanntlich die Länge einer Instruktion an. Was steht aber in der Speicherzelle TEMPX? Betrachten wir dazu nochmals die Subroutine RDINST (**Bild 11**). Während dieser Subroutine wird beim Einlesen des OP-Codes der neuen Instruktion die Länge in der Speicherzelle TEMPX abgelegt. Dieser Längeinformation bedient sich der Computer bei der INPUT-Routine, bevor er zur Subroutine FILLWS verzweigt. In der Subroutine FILLWS wird somit hinter der zuvor auf dem Display angezeigten Instruktion die eingegebene neue Instruktion eingesetzt. Da der Display-Pointer CURAD seit der Subroutine NEXT keine Veränderung mehr erfahren hat, zeigt er nach der Rückkehr zum zentralen Punkt CMND bereits auf den OP-Code der soeben eingegebenen Instruktion. Diese Instruktion zeigt der Computer jetzt während der Subroutine SCAN auf dem Display an und wartet auf eine neue Kommando-Taste. Das INPUT-Kommando ist somit abgeschlossen.

Die SKIP-Routine

Zur Besprechung der SKIP-Routine nehmen wir wieder **Bild 5** zur Hand. Wie noch aus Kapitel 5 bekannt, verwenden wir das SKIP-Kommando bei der Überprüfung eines mit dem Editor eingegebenen Programms. Welche Aufgabe muß die SKIP-Routine erfüllen? Die beiden folgenden Punkte geben darüber Auskunft:

1. Beim Drücken der SKIP-Taste zur folgenden Instruktion springen.
2. Eine Fehlermeldung geben, wenn man mit dem SKIP-Kommando über den EOF-Charakter hinausgesprungen ist.

Das SKIP-Kommando wird mit den beiden Instruktionen CMP # 12 und BNE ausgefiltert. Das einzige, was beim SKIP-Kommando verändert werden muß, ist der Display-Pointer CURAD. Die Subroutine, die diesen Pointer um eine, zwei oder drei Adressen nach unten bewegt kennen wir bereits: Es ist die Subroutine NEXT (**Bild 10**). In dieser Subroutine prüft der Computer, ob der Display-Pointer CURAD schon über den momentanen Endadreß-Pointer CEND hinausblickt. Ist CURAD größer als CEND, muß eine Fehlermeldung erfolgen. Über eine BPL-Instruktion verzweigt das Programm zum Label ERRA. Das Display zeigt dann solange EEEEEE, bis man die SKIP-Taste losläßt. Im Normalfall verzweigt aber das Programm über die NMI-Instruktion von der SKIP-Routine zum zentralen Punkt CMND und der Junior-Computer wartet auf eine neue Kommando-Taste.

Die DELETE-Routine

Welche Aufgabe muß die DELETE-Routine erfüllen? Aus Kapitel 5 wissen

wir, daß beim Drücken der DELETE-Taste die momentan auf dem Display angezeigte Instruktion aus dem Arbeitsspeicher des Junior-Computers geholt wird. Die Aufgabe, die der Junior-Computer dabei erfüllen muß, läßt sich leicht in Worte fassen:

1. Bewege den Datenblock zwischen der Instruktion, die direkt der momentan auf dem Display angezeigten Instruktion folgt und den Endadreß-Pointer CEND um ein, zwei oder drei Bytes nach oben.
2. Um wieviele Bytes dieser Speicherblock nach oben bewegt wird, entscheidet die Länge der momentan auf dem Display angezeigten Instruktion.
3. Die auf dem Display dargestellte Instruktion wird dadurch ausgelöst, indem der nach oben geschobene Datenblock (siehe Punkt 1) diese überschreibt.

Die Subroutine UP bewegt den erwähnten Datenblock nach oben und überschreibt somit die soeben auf dem Display angezeigte Instruktion. Auf dem Display ist dann die Instruktion sichtbar, die der entfernten Instruktion folgt. Da sich beim Entfernen einer Instruktion die File um ein, zwei oder drei Bytes verkürzt, muß auch der momentane Endadreß-Pointer CEND nach oben bewegt werden. Diese Aufgabe übernimmt die Subroutine RECEND (=Reduce Current End Address = vermindere den Pointer CEND um die Länge der Instruktion, die aus der File entfernt wurde). **Bild 14** zeigt das detaillierte Flußdiagramm von RECEND. In dieser Subroutine führt der Computer eine 16-Bit-Subtraktion durch. Somit ist die Subroutine RECEND (16-Bit-Subtraktion) eine Umkehrung der Subroutine ADCEND von Bild 13 (16-Bit Addition). Die Subroutine RECEND schiebt den Pointer CEND um ein, zwei oder drei Speicherplätze nach oben, während die Subroutine ADCEND den Pointer CEND um dieselbe Anzahl von Speicherplätzen nach unten bewegt.

Nun sind alle fünf Kommando-Routinen des Editors bekannt. Es fehlt noch die Erläuterung der Subroutinen DOWN, UP und OPLEN/LENACC.

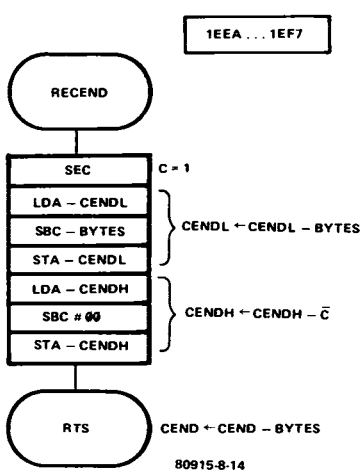


Bild 14. RECEND ist die Umkehrung der Subroutine ADCEND. RECEND bewegt den Pointer CEND um ein, zwei oder drei Adressen nach oben.

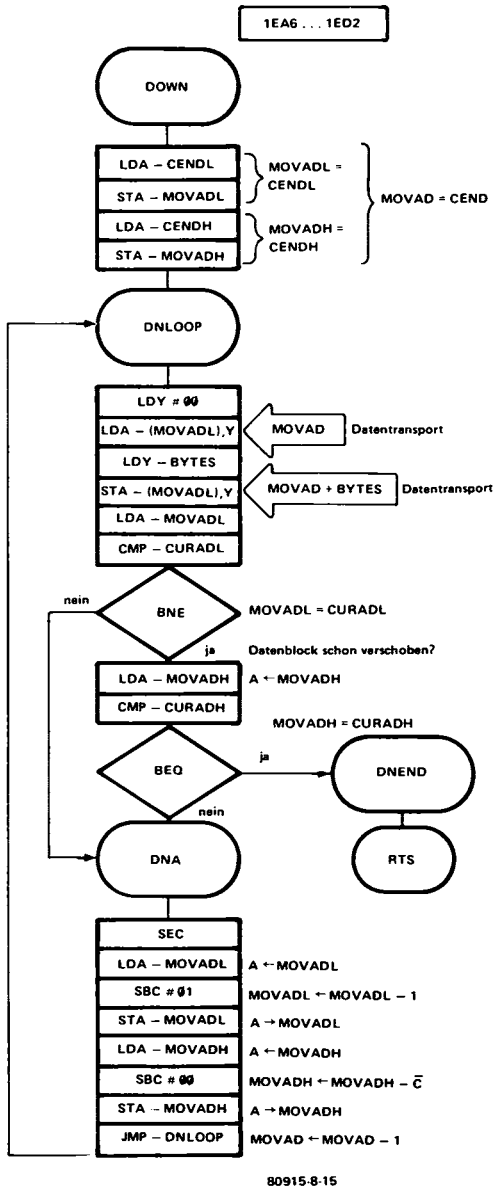


Bild 15. Die Subroutine DOWN schiebt einen Datenblock der File um ein, zwei oder drei Bytes nach unten. Somit wird in der File Platz gemacht, um vom Display eine neue Instruktion einsetzen zu können.

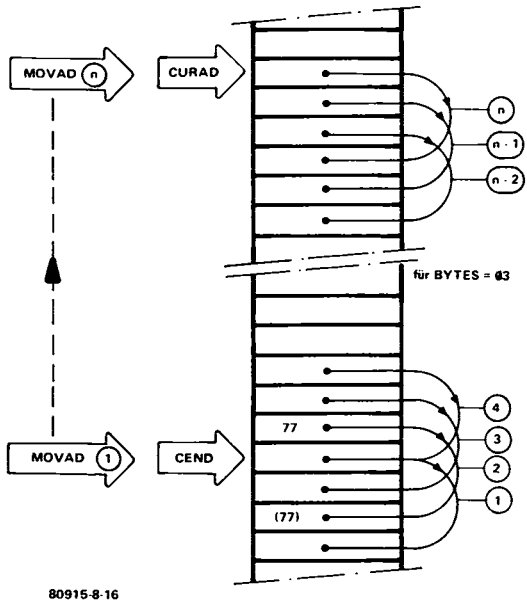


Bild 16. So sieht der Schiebemechanismus der Subroutine DOWN aus. Im Beispiel wird ein zusammenhängender Datenblock der File um drei Adressen nach unten geschoben, da der Inhalt der Speicherzelle BYTES = 03 ist.

Diese Subroutinen vereinigen in sich die gesamte Intelligenz des Editors und Assemblers.

Die Subroutine DOWN

Die Subroutine DOWN ist Bestandteil der zuvor besprochenen Subroutine FILLWS. FILLWS wurde beim INSERT- und INPUT-Kommando benötigt. Dabei mußte bei der Eingabe einer neuen Instruktion im Arbeitsspeicher des Junior-Computers Platz gemacht werden. Das heißt, der Computer mußte einen Datenblock in der File um ein, zwei oder drei Bytes nach unten bewegen. Diese Aufgabe erledigt der Computer mit Hilfe der Subroutine DOWN.
Bild 15 zeigt die einzelnen Instruktionen der Subroutine DOWN, **Bild 16** veranschaulicht die Funktionsweise dieser Routine. Bei der Subroutine DOWN bedient sich der Computer des Adreß-Pointers MOVAD (= Move Address). Dieser Adreß-Pointer zeigt immer auf ein Byte in der File, das momentan um ein, zwei oder drei Speicherplätze nach unten bewegt werden soll. Deshalb machen die ersten vier Instruktionen der Subroutine DOWN den Pointer MOVAD gleich dem Pointer CEND. MOVAD zeigt anfangs auf die gleiche Adresse wie CEND, das ist die Speicherzelle hinter dem EOF-Charakter 77.

Jetzt ist der Computer beim Label DNLOOP angekommen. Es wird der Inhalt der Speicherzelle, auf die MOVAD zeigt LDA-(MOVADL),Y in den Accu geladen (Y=ØØ). Um wieviele Speicherzellen das soeben geladene Byte nach unten verschoben werden soll, hängt vom Inhalt der Speicherzelle BYTES ab. In BYTES steht die Länge der Instruktion, die vom Displaybuffer in die File transportiert werden soll. Deshalb laden wir das Y-Register mit dem Inhalt von BYTES und speichern den Accu-Inhalt um BYTES-Speicherplätze weiter unten: STA-(MOVADL),Y.
Anschließend folgt ein Vergleich zwischen den beiden Pointern CURAD und MOVAD. Ist der Pointer MOVAD ungleich dem Pointer CURAD, müssen noch weitere Bytes in der File nach unten bewegt werden. Dann macht der Computer ab dem Label DNA den Pointer MOVAD um eins kleiner, das heißt, MOVAD wandert vom File-Ende nach oben (siehe auch Bild 16). Nach dem Rücksprung zum Label DNLOOP bewegt der Computer das Byte in der File, auf das MOVAD nun zeigt ebenfalls um ein, zwei oder drei Speicherplätze nach unten. Dabei wiederholt sich der soeben beschriebene Vorgang. Es folgt wieder ein Vergleich der beiden Pointer MOVAD und CURAD. Ist schließlich nach mehrmaligen Dekrementieren MOVAD gleich CURAD, dann sind alle Bytes ab dem Display-Pointer CURAD nach unten bewegt. Somit ist hinter CURAD ein Speicherplatz von ein, zwei oder drei Bytes entstanden, in den sich der Inhalt des Displaybuffers kopieren läßt.
Bild 16 zeigt grafisch, wie die Subroutine DOWN die einzelnen Bytes zwischen den Pointern CURAD und CEND nach unten transportiert. Dabei zeigt der Pointer MOVAD immer auf das Byte, das soeben von oben nach unten verschoben werden soll. Nach jeder Verschiebungsoperation dekrementiert der Computer den Pointer MOVAD um eins. Somit wandert MOVAD von unten nach oben. Dabei ist der Start-Wert von MOVAD gleich CEND und der End-Wert gleich CURAD. In **Bild 16** ist der Inhalt der Speicherzelle BYTES gleich Ø3. Ab CURAD werden darum drei aufeinanderfolgende Speicherplätze frei, in die sich eine neue Instruktion mit einer Länge von drei Bytes schreiben läßt.
Beim Standart-Junior-Computer ist die längste zusammenhängende File 1/2-KByte lang. Bei solch kurzen Files hält sich der Computer nur kurze Zeit in der Subroutine DOWN auf, da nur wenige Bytes in der File zu verschieben sind. Bei einer Erweiterung des Junior-Computers kann aber eine File ohne weiteres zwei oder vier KByte lang sein. Wird am Beginn einer solch langen File eine Instruktion mit einem INSERT- oder INPUT-Kommando eingesetzt, verweilt der Junior-Computer einige Sekundenbruchteile in der Subroutine DOWN. Das ist auch auf dem Display erkennbar: es erlischt für eine kurze Zeit!

Die Subroutine UP

Die Subroutine UP ist eine Umkehrung der Subroutine DOWN. Der Junior-Computer verwendet die Subroutine UP:
* wenn der Programmierer die DELETE-Taste drückt,
* beim Assemblieren, wenn die Labels aus der File geholt werden müssen (wird im folgenden Kapitel besprochen).

Bild 17 zeigt die einzelnen Instruktionen der Subroutine UP. Diese Sub-

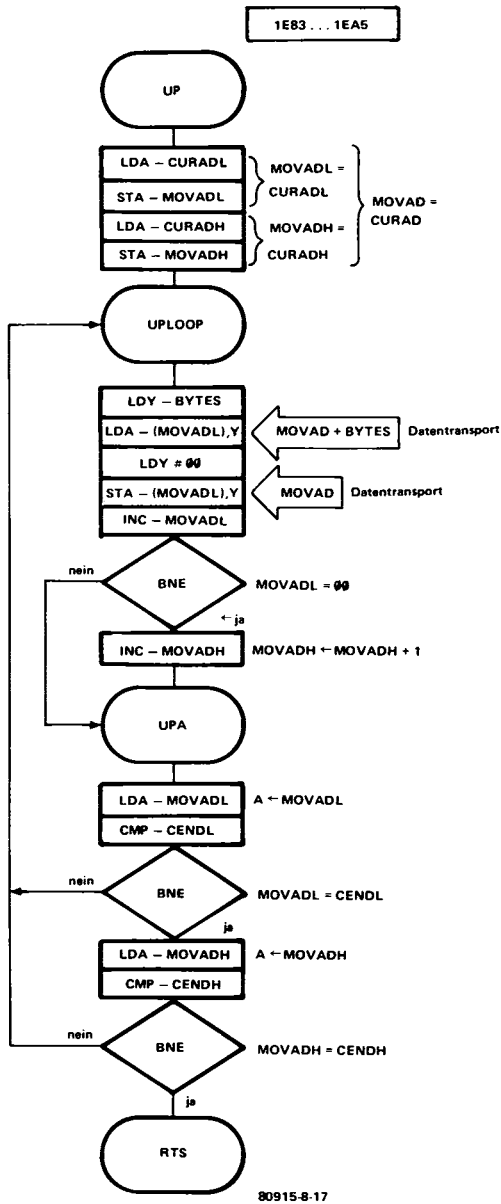


Bild 17. Die Subroutine UP ist die Umkehrung der Subroutine DOWN. UP schiebt einen zusammenhängenden Datenblock der File um ein, zwei oder drei Adressen nach oben. Somit lassen sich aus der File Labels oder Instruktionen entfernen. Die Subroutine UP wird beim Editor und beim Assembler benötigt.

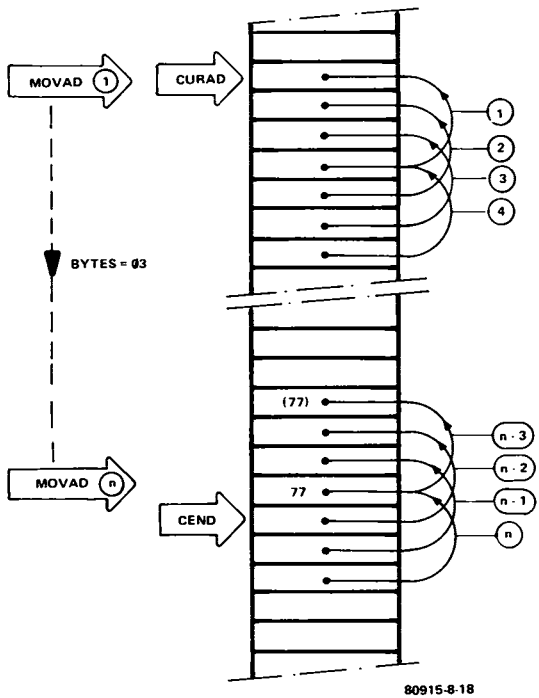


Bild 18. So sieht der Schiebemechanismus der Subroutine UP aus. Im Beispiel wird ein Datenblock der File um drei Adressen nach oben geschoben, da der Inhalt der Speicherzelle BYTES = Ø3 ist.

routine bedient sich wieder des Pointers MOVAD, der immer auf das Byte in der File zeigt, das um ein, zwei oder drei Speicherplätze nach oben bewegt werden soll. **Bild 18** zeigt, welchen Datenblock der Computer von unten nach oben bewegt. Dabei bewegt der Computer den Datenblock beginnend bei CURAD + BYTES und endend bei CEND + BYTES um ein, zwei oder drei Speicherplätze nach oben. Somit wird eine Instruktion, deren OP-Code von CURAD adressiert ist, aus der File entfernt bzw. überschrieben.
Hinter dem Label UP wird der Pointer MOVAD gleich dem Display-Pointer CURAD gemacht. Ab UPLOOP lädt der Accu den Inhalt der Speicherzelle, auf die der Pointer MOVAD + BYTES zeigt: LDA-(MOVADL),Y. Anschließend schreibt die CPU den Accu-Inhalt in die Speicherzelle, auf die MOVAD zeigt: STA-(MOVADL),Y (wobei Y=ØØ ist). Somit transportiert der Computer einzelne Bytes der File um ein, zwei oder drei Speicherplätze nach oben, indem er ein Byte mit MOVAD + BYTES aus der File in den Accu lädt und anschließend mit MOVAD + BYTES – BYTES = MOVAD in die File zurückschreibt. Nach dieser Bewegungsoperation wird der Pointer MOVAD um eins inkrementiert. MOVAD wandert somit in der

File von oben nach unten. Ab dem Label UPA führt der Computer einen Vergleich zwischen MOVAD und CEND durch. Ist dabei MOVAD ungleich CEND, sind noch weitere Bytes in der File von unten nach oben zu bewegen. Deshalb verzweigt das Programm zum Label UPLOOP und führt dort eine neue Bewegungsoperation durch. Ist aber nach mehrmaligem Inkrementieren der Pointer MOVAD gleich dem Pointer CURAD, sind keine Bytes mehr zu bewegen und der Computer kehrt aus der Subroutine UP ins Hauptprogramm zurück.

Die Subroutine OPLEN/LENACC

Einen Teil der Subroutine OPLEN kennen wir noch aus Buch 1, Kapitel 4: die Subroutine LENACC. Während dieser Subroutine berechnet der Computer die Länge einer Instruktion. Vor dem Aufruf der Subroutine LENACC muß der OP-Code im Accu stehen; nur dann arbeitet LENACC. OPLEN ist eine Verfeinerung der Subroutine LENACC. Beim Aufruf von OPLEN wird der OP-Code der Instruktion, auf die der Display-Pointer CURAD zeigt, automatisch in den Accu geladen.
Zwischen der Subroutine LENACC in EPROM des Junior-Computers und der Subroutine LENACC in Buch 1, Seite 146 bestehen aber ein paar Unterschiede: Zunächst sind die beiden Index-Register X und Y miteinander vertauscht. Auf die Funktionsweise der Subroutine LENACC hat das jedoch keine Folgen. Wie noch aus Buch 1 bekannt, sind die Längen der CPU-Instruktionen in einer Lookup Table, genannt LEN, abgelegt. Einige Plätze dieser Lookup Table sind mit anderen Bytes gefüllt, da dem Label-Identifikator FF eine Instruktionslänge von drei Bytes zugeordnet sein muß und dem EOF-Charakter eine Länge von einem Byte. In welchen Spalten der OP-Code-Tabelle sich der Label-Indentifikator und der EOF-Charakter befinden, zeigt Bild 20. Auch die Längen, die diesen Pseudo-OP-Codes zugeordnet sind, zeigen die Spalten 7 und F.
An dieser Stelle soll nicht nochmals eine ausführliche Beschreibung der Subroutine OPLEN/LENACC erfolgen. In Buch 1 beschreiben wir ausführlich die Funktionsweise dieser Subroutine. Da aber in Zukunft diese Subroutine häufig zur Anwendung kommt, sei nochmals stenoartig der Ablauf beschrieben:

1. Nach dem Aufruf der Subroutine OPLEN wird der Accu mit dem OP-Code der Instruktion geladen, auf die der Display-Pointer CURAD zeigt. Dabei zeigt CURAD immer auf einen normalen OP-Code der CPU 6502 oder auf einen Pseudo-OP-Code, wie den Label-Identifikator FF oder den EOF-Charakter 77.
2. Hinter dem Label LENACC wird das Y-Register mit Ø1 geladen und anschließend unregelmäßige Instruktionen (siehe Buch 1, Seite 145) wie BRK, RTI und RTS herausgefiltert. Ist eine dieser drei Instruktionen im Accu, wird das Y-Register in der Speicherzelle BYTES abgelegt und den drei Instruktionen somit eine Instruktionslänge von einem Byte zugeordnet. Eine weitere unregelmäßige Instruktion ist JSR. Diese Instruktion ist drei Bytes lang und wird ebenfalls gesondert herausgefiltert. Dazu laden wir das Y-Register mit Ø3 und legen diese anschließend in der Speicherzelle BYTES ab, falls im Accu der OP-Code einer JSR-Instruktion (= 2Ø) steht.

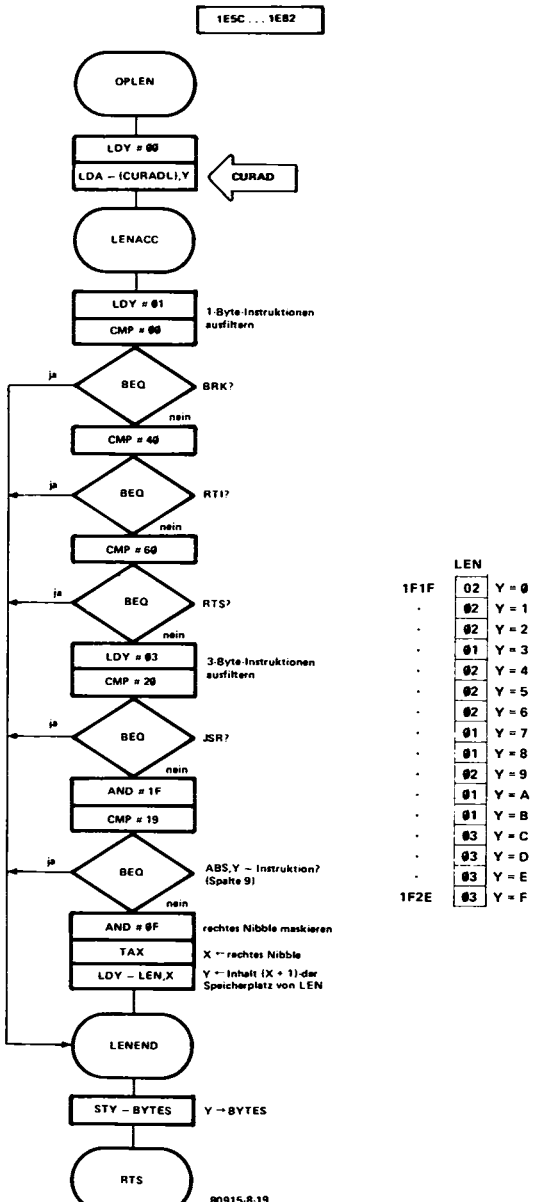


Bild 19. Die Subroutine OPLEN berechnet die Länge der Instruktion, auf deren OP-Code der Display-Pointer CURAD zeigt. Die Berechnung der Instruktionslänge beginnt hinter dem Label LENACC.

niederwertiges Datennibble

hochwertiges Datennibble	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)	EOF 77 (1)
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)	

niederwertiges Datennibble

hochwertiges Datennibble	8	9	A	B	C	D	E	F	
0	PHP (1)	ORA # (2)	ASL A (1)			ORA ABS (3)	ASL ABS (3)		0
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)	Label FF (3)	F

80915-20

Bild 20. Die OP-Code-Tabelle der CPU 6502. Auch der Pseudo-OP-Code FF und der EOF-Charakter sind in dieser Tabelle eingetragen.

3. Steht im Accu zu diesem Zeitpunkt keine der unregelmäßigen Instruktionen BRK, RTI, RTS und JSR, sind nur noch die niederwertigen fünf Bits des Accus interessant. Die folgende Instruktion in der Subroutine OPLEN ist deshalb AND # 1F. Das Y-Register ist noch immer Ø3. Zusammen mit der anschließenden Instruktion CMP # 19 werden alle Instruktionen mit einer Länge von drei Bytes aus der Spalte 9 von **Bild 20** ausgefiltert. Das sind Instruktionen, welche die Adressierungsart ABS,Y (Absolute Indexed, Y Addressing siehe Buch 1, Seite 113) haben.
4. Steht im Accu keine Instruktion mit ABS,Y Addressing, dann sind nur noch die niederwertigen vier Bits des Accus interessant. Die folgende Instruktion ist deshalb AND # ØF. Die so entstandene Zahl wird als Index für die Lookup Table LEN verwendet (TAX). Dann holt der Computer aus der Lookup Table LEN die Instruktionslänge und legt diese wieder in der Speicherzelle BYTES ab. Die Instruktionen dafür sind: LDY-LEN,X und STY-BYTES.

Somit haben wir auch den Editor im EPROM des Junior-Computers kennengelernt. Die zahlreichen Editor-Subroutinen dürften in Zukunft für den Programmierer eine große Erleichterung bei der Programment-

wicklung sein: Sie lassen sich leicht in "Do It Yourself Programme" einbauen. Der Programmierer muß dazu nur wissen, wie die diversen CPU-Register oder verschiedene Speicherzellen zu laden sind, bevor er die Editor-Subroutinen aufruft. **Das gesamte Source Listing des Editors kann man im Anhang dieses Buches nachschlagen.**

9

Das Assembler-Programm

Das Assembler-Programm ist ebenfalls im EPROM des Junior-Computers enthalten. Die Aufgabe des Assemblers ist, ein mit dem Editor eingegebenes Programm so zu bearbeiten, daß es die CPU 6502 verstehen kann (assemblieren = anpassen). Beim Arbeiten mit dem Editor ist gewöhnlich der Operand einer Sprung-Instruktion oder einer Verzweigungs-Instruktion eine Label-Nummer, also eine symbolische Adresse. Auch die Labels sind Pseudo-Instruktionen, die der Prozessor nicht verstehen kann. Deshalb muß der Assembler beim assemblieren eines eingegebenen Programms folgende Aufgaben erfüllen:

* **Alle Labels müssen aus dem Programm entfernt werden.**
* **Bei JMP- und JSR-Instruktionen müssen die symbolischen Adressen (Label-Nummern) durch die tatsächlichen Adressen ersetzt werden.**
* **Bei Branch-Instruktionen müssen die symbolischen Offsets (Label-Nummern) durch die tatsächlichen Offsets ersetzt werden.**

Wie sich der Assembler durch eine mit dem Editor eingegebene File arbeitet, besprechen wir in diesem Kapitel.

Aus Kapitel 5 ist bereits der große Vorteil des Editors und des Assemblers bekannt. Beim Eingeben eines Programms muß man nicht mehr die Startadressen von Subroutinen beziehungsweise die Offsets von Branch-Instruktionen kennen. An deren Stelle verwenden wir Labels, sogenannte symbolische Adressen. Das Einfügen von Labels in eine File durch den Computer erspart uns eine Menge Rechen- und Zählarbeit. Mit dem Editor lassen sich mühelos größere Programme im Speicher des Junior-Computers ablegen. Nach dem Assemblieren kann man sicher sein, daß alle Offsets und Start-

adressen von Subroutinen richtig sind. Ein mühevolles und zeitraubendes Durchlaufen von Programmen in Step by Step Mode ist somit auf ein Minimum reduziert. Ein weiterer Vorteil von Editor und Assembler ist, daß Programme so kompakt wie möglich im Speicher des Junior-Computers abgelegt sind.

Das globale Assembler-Flußdiagramm

Das Assemblieren einer mit dem Editor eingegebenen File erfolgt in zwei Phasen. In Kapitel 5 sprachen wir von einem Two Pass Assembler (= Zwei-Schritt-Assembler). Um die beiden Phasen beim Assemblieren übersichtlich zu zeigen, haben wir das globale Flußdiagramm in zwei Bilder aufgeteilt: **Bild 1** für die erste Assembler-Phase und **Bild 2** für die zweite Assemblerphase.
Beginnen wir mit **Bild 1**! Hinter dem Label ASSEMB mit der Startadresse 1F51 befindet sich ein Programmteil, der alles für den Assemblierungsvorgang vorbereitet. Diese Vorbereitung besteht darin, daß verschiedene Adreß-Pointer bestimmte Stellungen einnehmen.
Es folgt der Label PASSA, das eigentliche Assemblerprogramm für die erste Phase. In dieser ersten Phase des Assemblierens konzentriert sich der Junior-Computer nur auf die Labels, die sich in der File befinden. Wie findet aber der Junior-Computer die einzelnen Labels in der File auf? Für den Computer ist das eine leichte Aufgabe, da er für jede Instruktion, die sich in der File befindet, die Instruktionslänge berechnen kann. Die Subroutine dafür ist bekannt: OPLEN. Beim Aufspüren der Labels startet der Computer am oberen Ende der File. Dann fragt er: hat die erste Instruktion in der File den OP-Code FF? Wenn ja, handelt es sich um einen Label, wenn nein, ist es ein normaler OP-Code.
Handelt es sich um einen Label, also eine Instruktion mit dem OP-Code FF, notiert der Junior-Computer die Adresse, an der sich der Pseudo-OP-Code FF befindet. Auch die Label-Nummer, die dem Pseudo-OP-Code FF folgt, notiert der Computer. Nächste Frage: Wohin notiert der Junior-Computer die Adresse, an der sich der Pseudo-OP-Code FF befindet und die darauf folgende Label-Nummer? Adresse des Pseudo-OP-Codes und die Label-Nummer legt der Computer auf dem sogenannten Symbol-Stack ab. Der Symbol-Stack ist nichts anderes als ein Notizzettel, der mit dem normalen Stack des Junior-Computers in Page 1 vergleichbar ist. Der einzige Unterschied zwischen beiden Stacks ist, daß der Stack-Pointer des Stacks in Page 1 von der CPU beeinflußt (hardwaremäßig) und der Stack-Pointer des Symbol-Stacks von einem Programm (softwaremäßig) beeinflußt wird. Gleich ist aber bei beiden Stacks, daß diese von unten nach oben wachsen. Aber nun wieder zurück zur ersten Phase des Assemblierens!
Nachdem der Computer einen Label in der File aufgespürt, die Adresse des Pseudo-OP-Codes sowie die Label-Nummer auf dem Symbol-Stack abgelegt hat, muß der Label aus der File (zwischen BEGAD und CEND) entfernt werden. Die Subroutine, die einen Label entfernen kann, kennen wir bereits vom DELETE-Kommando. Es handelt sich um die Subroutine UP. Da ein Label immer drei Bytes lang ist, wird der Datenblock, der hinter dem Label beginnt und bei der Adresse endet, auf die CEND zeigt, um drei Bytes nach oben geschoben. Somit ist der Label aus der File entfernt und

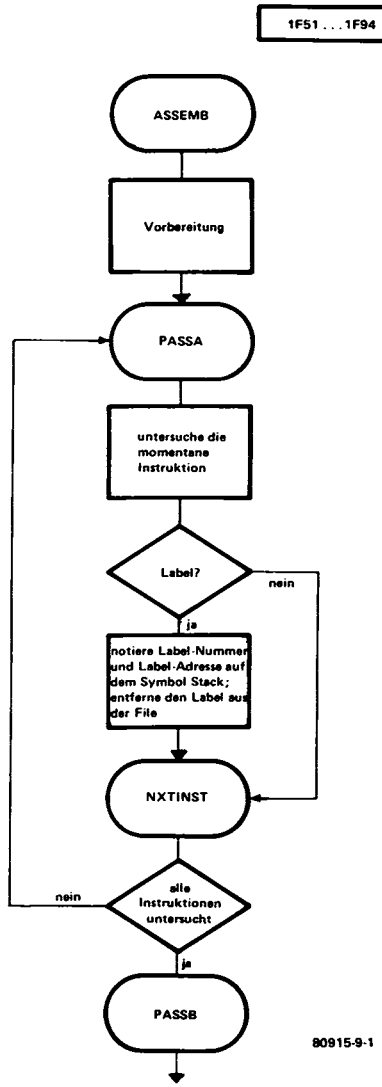


Bild 1. Der Assembler bearbeitet die mit dem Editor eingegebene File in zwei Schritten. Dieses Bild zeigt das globale Flußdiagramm der ersten Phase des Assemblierens. Während dieser Phase holt der Junior-Computer alle Labels aus der File und legt die Label-Nummer sowie die Adresse, an der sich der Label befindet, auf dem Symbol-Stack ab.

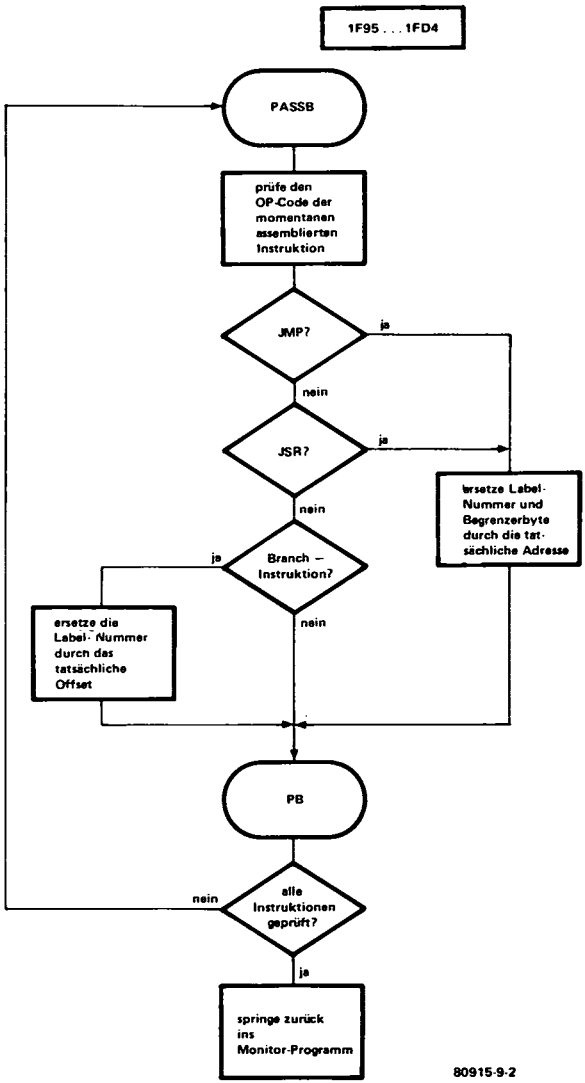


Bild 2. Das globale Flußdiagramm der zweiten Phase des Assemblierens. Hinter Sprung-Instruktionen wird die Label-Nummer und das Begrenzerbyte durch die tatsächliche Sprung-Adresse ersetzt. Auch die Label-Nummer hinter Branch-Instruktionen wird in dieser Phase durch das tatsächliche Offset ersetzt. In beiden Fällen bedient sich der Junior-Computer des Symbol-Stacks.

die File ist um drei Bytes kürzer geworden. Es ist sofort ersichtlich, daß auch der Adreß-Pointer CEND der neuen Situation angepaßt werden muß und ebenfalls um drei Speicherzellen nach oben wandert (siehe Kapitel 8, Bild 5 und Bild 18). Jetzt steht der OP-Code der Instruktion, die dem entfernten Label folgte, an der Adresse, an der sich zuvor der Pseudo-OP-Code FF des Labels befand. Bild 3a und Bild 3b veranschaulichen deutlich den soeben beschriebenen Vorgang.
Nachdem der Label aus der File entfernt und alles wissenswerte über diesen entfernten Label auf dem Symbol-Stack abgelegt ist, sucht der Junior-Computer einen weiteren Label in der File auf. Dazu springt er von Instruktion zu Instruktion und prüft jedesmal, ob diese den Pseudo-OP-Code FF hat. Sobald es sich um den Pseudo-OP-Code FF handelt, wiederholt sich derselbe Vorgang:

1. Lege die Adresse, an der sich der Pseudo-OP-Code FF befindet auf dem Symbol-Stack ab (Zuerst das hochwertige und dann das niederwertige Adreß-Byte).
2. Hole die dem Pseudo-OP-Code folgende Label-Nummer aus der File und lege diese ebenfalls auf dem Symbol-Stack ab. Somit sind vom Label drei wichtige Merkmale auf dem Symbol-Stack – dem Label-Notizzettel für den Junior-Computer – abgelegt: das hoch- und das niederwertige Adreß-Byte der Adresse, an der sich der Pseuo-OP-Code des Labels befindet sowie die Label-Nummer. Mit dem genauen Aufbau des Symbol-Stacks beschäftigen wir uns zu einem späteren Zeitpunkt.
3. Entferne den Label aus der File. Die dem Label folgende Instruktion steht jetzt an der Adresse, an der zuvor der Label stand.
4. Untersuche die File auf weitere Labels und beende die Suchaktion, wenn keine weiteren Labels in der File vorhanden sind.

Sehen wir uns dazu nochmals Bild 3 an. Dieses Bild zeigt vier Etappen einer Label-Suchoperation während der ersten Phase der Assemblierung.

Bild 3a: Der Computer trifft an der Adresse Ø2ØØ auf den Pseudo-OP-Code eines Labels. Nachdem alles wissenswerte über den Label auf dem Symbol-Stack abgelegt ist, verschwindet der Label aus der File.
Bild 3b: Der Junior-Computer hat den Label aus der File entfernt. Da ein Label immer drei Bytes lang ist, verschiebt sich der Datenblock hinter dem Label um drei Bytes nach oben. Dadurch rückt an die ursprüngliche Label-Adresse Ø2ØØ der OP-Code, dessen Instruktion unmittelbar dem Label folgt. Die File ist nach der Entfernung des Labels um drei Bytes kürzer geworden.
Bild 3c: Der Junior-Computer setzt die Label-Suchoperation fort. Alle Instruktionen, deren OP-Code **ungleich FF** ist, bleiben bei der Such-Operation unberücksichtigt. An der Adresse Ø217 trifft der Computer wieder auf einen Label, gekennzeichnet durch den Pseudo-OP-Code FF. Auch alles wissenswerte von diesem Label wird auf dem Symbol-Stack abgelegt und der Label anschließend aus der File entfernt.
Bild 3d: Der Label mit der Label-Nummer 12 ist aus der File entfernt. Der dem Label folgende Datenblock wird abermals um drei Bytes nach oben geschoben. Somit steht wieder der OP-Code der Instruktion, die dem Label folgte, an der Adresse Ø217, an der zuvor der Pseudo-OP-Code des Labels stand.
Somit ist alles wissenswerte über die erste Phase des Assemblierens gesagt.

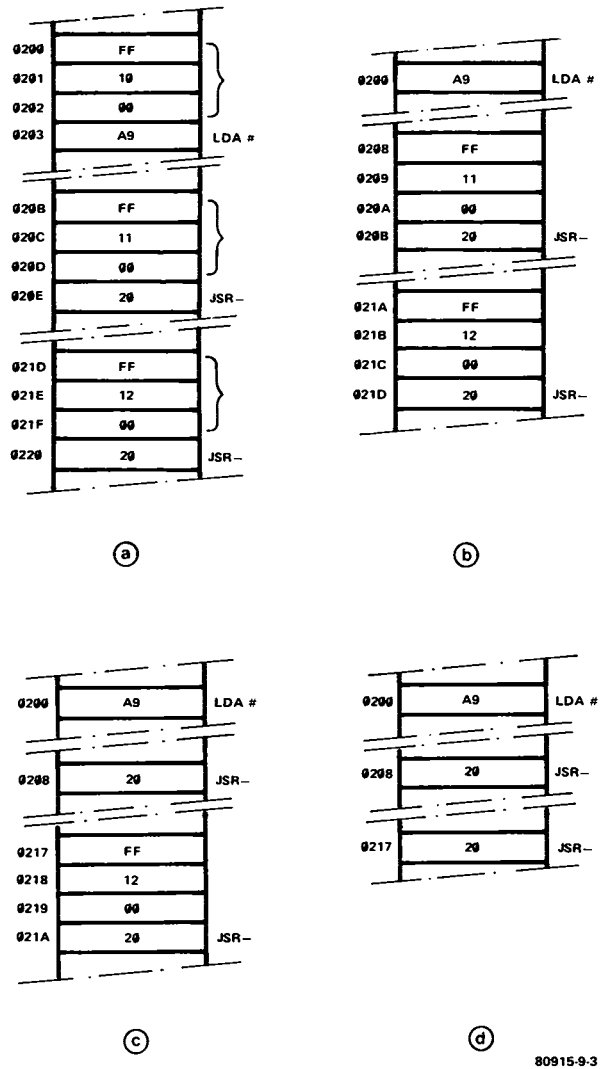


Bild 3. Vor der ersten Phase des Assemblierens sind noch alle Labels in der File vorhanden. Bild 3a zeigt diesen Sachverhalt. Dann bearbeitet der Computer die File von oben nach unten und entfernt dabei alle Labels. Alles, was er über die Labels wissen muß, legt der Computer auf dem Symbol-Stack ab. Bild 3b zeigt die Situation, nachdem der Label mit der Nummer 10 aus der File entfernt wurde. Bild 3c zeigt, wie die File aussieht, nachdem der Label mit der Nummer 11 und Bild 3d nachdem der Label mit der Nummer 12 aus der File entfernt wurden. Die den Labeln folgende OP-Codes stehen jetzt an der Adresse, an der sich zuvor der Pseudo-OP-Code FF befand.

Jetzt wenden wir uns der zweiten Phase (PASSB) zu. **Bild 2** zeigt das globale Flußdiagramm der zweiten Phase. In dieser Phase sind bereits alle Labels aus der File entfernt. Direkt nach dem Label PASSB untersucht der Junior-Computer alle Instruktionen in der File, beginnend bei der ersten. Welche Instruktionen sind jetzt für den Computer interessant? Das sind alle Instruktionen, hinter deren OP-Code eine Label-Nummer steht. Aus Kapitel 5 kennen wir noch diese Instruktionen:

* JMP-Instruktion mit OP-Code 4C
* JSR-Instruktion mit OP-Code 2Ø
* die Branch-Instruktionen BPL, BMI, BEQ, BNE, BCC, BCS, BVC und BVS.

In der zweiten Phase des Assemblierens muß der Computer diesen Instruktionen die tatsächlichen Adressen und Offsets zuordnen. Deshalb werden diese drei Arten von Instruktionen aus der File nacheinander herausgefiltert. Der Computer kann diese besonderen Instruktionen leicht am OP-Code erkennen:

1. **Der Computer trifft auf eine JMP-Instruktion:** Hinter dem OP-Code 4C steht eine Label-Nummer. Zu dieser Label-Nummer gehört eine Adresse, die während der ersten Phase des Assemblierens (PASSA) zusammen mit der Label-Nummer des entfernten Labels auf dem Symbol-Stack abgelegt wurde. Jetzt sucht der Computer auf dem Symbol-Stack eine Label-Nummer auf, die gleich der Label-Nummer ist, die dem OP-Code 4C folgt. Ist diese Label-Nummer auf dem Symbol-Stack gefunden, holt der Computer die zu dieser Label-Nummer gehörende Adresse vom Symbol-Stack und setzt diese aus zwei Bytes bestehende Adresse hinter dem OP-Code 4C ein. Die Label-Nummer und das Begrenzer-Byte, die zuvor hinter dem OP-Code 4C standen, werden dabei von der tatsächlichen Sprungadresse überschrieben. Die JMP-Instruktion mit OP-Code 4C ist somit assembliert.
2. **Der Computer trifft auf eine JSR-Instruktion:** Hinter dem OP-Code 2Ø steht wie bei der JMP-Instruktion eine Label-Nummer. Um diese Instruktion zu assemblieren, holt der Computer vom Symbol-Stack die zur Label-Nummer gehörende absolute Adresse. Diese Adresse setzt er hinter dem OP-Code der JSR-Instruktion ein. Die JSR-Instruktion ist somit assembliert. Dabei ist der Vorgang der Assemblierung derselbe wie bei der JMP-Instruktion.
3. **Der Computer trifft auf eine BRANCH-Instruktion:** Die CPU 6502 kennt bekanntlich auch Branch-Instruktionen. Trifft der Computer während der zweiten Phase des Assemblierens auf den OP-Code einer Branch-Instruktion, muß die Label-Nummer hinter dem OP-Code durch das tatsächliche Offset ersetzt werden. Die Assemblierung einer Branch-Instruktion läßt sich wie folgt beschreiben:
   * Der Computer sieht nach, welche Label-Nummer hinter dem OP-Code der Branch-Instruktion steht. Für diese Label-Nummer ist auf dem Symbol-Stack eine absolute Adresse abgelegt.
   * Der Computer sucht die Label-Nummer, die hinter dem OP-Code der Branch-Instruktion steht, auf dem Symbol-Stack auf. Ist diese Label-Nummer auf dem Symbol-Stack gefunden, ist auch die zur Label-Nummer gehörende Absolut-Adresse bekannt.
   * Ferner kennt der Computer die Adresse, an der sich der OP-Code der

zu assemblierenden Branch-Instruktion befindet. Aus der Adresse, die zur Label-Nummer gehört und der Adresse, an der sich der OP-Code der Branch-Instruktion befindet, kann der Computer das tatsächliche Offset berechnen.

*Das berechnete Offset setzt der Computer hinter die zu assemblierende Branch-Instruktion ein. Somit ist auch die Branch-Instruktion assembliert.

4. **Der Computer trifft auf eine JMP-, JSR- oder Branch-Instruktion, deren Label-Nummer sich nicht auf dem Symbol-Stack befindet:** Aus Kapitel 5 ist bekannt, daß der Junior-Computer unter bestimmten Umständen eine JMP, JSR- oder Branch-Instruktion nicht assembliert. Das ist der Fall, wenn man die Label-Nummer hinter dem OP-Code **nicht** als Labelnummer hinter dem Label-Identifikator FF verwendet. Oder anders gesagt, der Label ist nicht in der File vorhanden. Deshalb konnte auch der Computer bei der ersten Phase der Assemblierung (PASSA, siehe Bild 1) die Label-Nummer und die dazugehörende Absolut-Adresse nicht auf dem Symbol-Stack ablegen. Trifft der Junior-Computer während der zweiten Phase der Assemblierung auf eine Instruktion wie JMP, JSR oder Branch, deren Label-Nummer er nicht auf dem Symbol-Stack auffinden kann, bleibt diese Instruktion unassembliert. Das heißt, hinter dem OP-Code werden keine Daten des Symbol-Stacks eingesetzt. Der Einfachheit halber ist diese Assembler-Sperre nicht in Bild 2 eingezeichnet.

Bisher ist in großen Zügen klar geworden, was sich nach dem Starten des Assemblers im Computer abspielt. Der Assembler bearbeitet die vom Editor erzeugte File in zwei Schritten. In der ersten Phase des Assemblierens entfernt der Computer alle Labels aus der File und legt die Label-Nummern und die zu diesen Labeln gehörenden Absolut-Adressen auf dem Symbol-Stack ab. Während PASSA sind für den Computer nur die Instruktionen interessant, die mit dem Pseudo-OP-Code FF beginnen. Sind alle Label-Nummern mit den dazugehörenden absoluten Adressen auf den Symbol-Stack abgelegt, geht der Computer zur zweiten Phase der Assemblierung über (PASSB).

Während der zweiten Phase der Assemblierung interessiert sich der Computer nur für die OP-Codes 4C, 2Ø und die OP-Codes der acht Branch-Instruktionen. Hinter diesen OP-Codes befinden sich die Label-Nummern (und bei den Sprung-Instruktionen die Begrenzer-Bytes). Jetzt sucht der Computer diejenige Label-Nummer auf dem Symbol-Stack auf, die hinter dem OP-Code der momentan assemblierten Instruktion steht. Zu jeder Label-Nummer auf dem Symbol-Stack gehört eine absolute Adresse. Diese absolute Adresse wird bei Sprunginstruktionen direkt hinter dem OP-Code eingesetzt. Bei Branch-Instruktionen berechnet der Junior-Computer aus dieser Adresse das tatsächliche Offset. Der Symbol-Stack und dessen Organisation blieben bisher noch unberücksichtigt. Mit dem jetzigen Vorwissen läßt sich der Symbol-Stack bei der folgenden Beschreibung des detaillierten Flußdiagramms leicht erklären.

Das detaillierte Flußdiagramm des Assemblers

Das detaillierte Flußdiagramm des Assemblers unterteilt sich wieder in zwei Phasen. **Bild 4** zeigt den Teil des Assemblers, der die Labels aus der

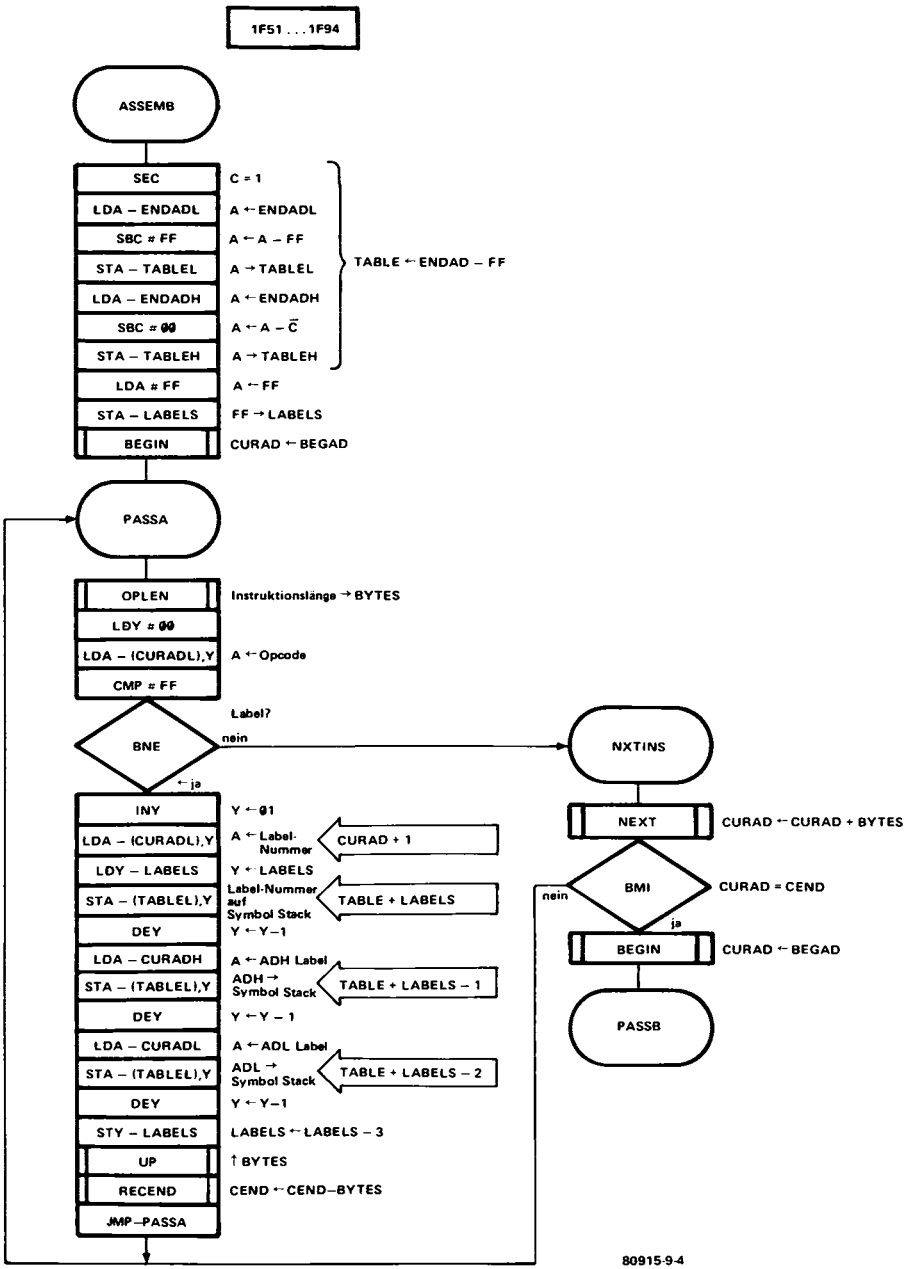


Bild 4. Das detaillierte Flußdiagramm der ersten Phase des Assemblers.

File entfernt und die Label-Nummern sowie die zu diesen Nummern gehörenden absoluten Adressen auf dem Symbol-Stack ablegt.
Der Assembler startet beim Label ASSEMB mit der Adresse 1F51. In folgenden Vorprogramm bis zum Label PASSA setzt der Computer verschiedene Adreß-Pointer und bereitet den Symbol-Stack auf seine Arbeitsweise vor. Bevor einige Details folgen, noch ein paar allgemeine Bemerkungen: Das Assembler-Programm hängt eng mit dem Editor-Programm zusammen. Das heißt, der Assembler benötigt diverse Subroutinen des Editors. Es ist deshalb sinnvoll, Kapitel 8 nochmals zu studieren, wenn die Funktionsweise der einzelnen Editor-Subroutinen nicht bekannt ist.
Der Assembler verwendet einen neuen Adreß-Pointer: TABLE. Dieser Adreß-Pointer ist in den beiden Page-Zero-RAM-Zellen TABLEH, TABLEL abgelegt. Die Adressen des TABLE-Pointers sind:
TABLEL hat die Adresse ØØEC und
TABLEH hat die Adresse ØØED.
Der Inhalt dieser beiden Speicherzellen ist die Adresse, auf die der Pointer TABLE zeigt. Dieser Adreß-Pointer formt die oberste Begrenzung des Arbeitsspeicherbereichs, in dem nach der Entfernung der Labels aus der File alles wissenswerte über diese Labels abgelegt ist. TABLE ist deshalb die oberste Adresse des Symbol-Stacks. Die unterste Adresse des Symbol-Stacks ist TABLE+FF. Somit ist der Symbol-Stack maximal 256 Bytes lang.
Die ersten sieben Instruktionen hinter dem Label ASSEMB sorgen dafür, daß der Adreß-Pointer TABLE auf eine Adresse zeigt, die 256 (FF) Speicherzellen vor der Adresse ENDAD liegt. **Bild 5** veranschaulicht grafisch,

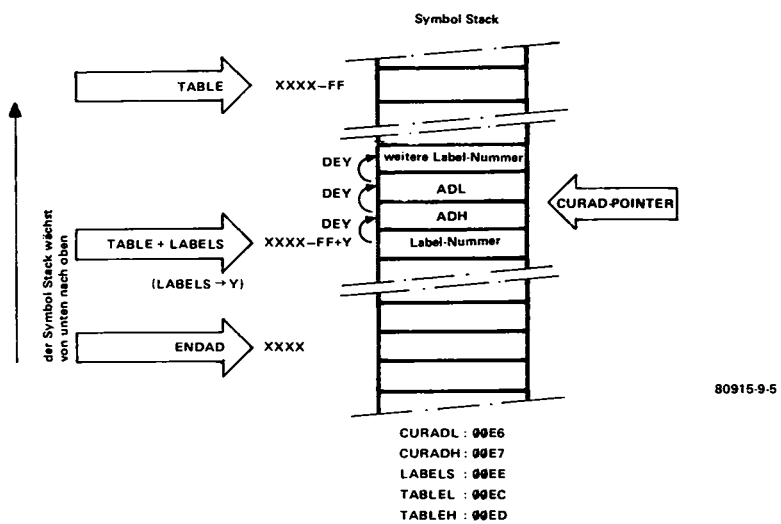


Bild 5. Mit dem Pointer TABLE+LABELS steuert der Junior-Computer den Datenverkehr zum oder vom Symbol-Stack. Es wächst der Symbol-Stack von unten nach oben. Dabei wird zuerst die Label-Nummer und anschließend der Inhalt des Display-Pointers CURAD auf den Symbol-Stack transportiert.

wie der Computer den Adreß-Pointer TABLE in Abhängigkeit der absoluten File-Endadresse ENDAD setzt.
Außerdem verwendet der Assembler noch eine neue RAM-Zelle mit dem Namen LABELS. LABELS hat die Adresse ØØEE. Addiert man den Inhalt der Speicherzelle LABELS zum Inhalt des Stack-Pointers TABLE, erhöht man die Adresse, auf die der Symbol-Stack-Pointer momentan zeigt. Die Adresse, auf die der Symbol-Stack-Adreß-Pointer effektiv zeigt, ist also TABLE+LABELS. Während der ersten Phase des Assemblierens zeigt der Symbol-Stack-Pointer immer auf den obersten unbesetzten Speicherplatz des Symbol-Stacks. Ist während der ersten Phase ein Label in der File gefunden, so kommt immer auf den obersten, unbesetzten Platz des Symbol-Stacks die Label-Nummer, über die Label-Nummer das hochwertige Adreßbyte des momentan assemblierten Labels und über das hochwertige Adreßbyte das niederwertige Adreßbyte des Labels zu stehen. Der Symbol-Stack ist also wie folgt organisiert:

TABLE		Oberster Platz des Symbol-Stacks
niederwertiges Nibble		hochwertiges Nibble
	GH	niederwertiges Label-Adreßbyte
Label (x)	I J	hochwertiges Label-Adreßbyte
	KL	Label-Nummer
	MN	niederwertiges Label-Adreßbyte
Label (y)	OP	hochwertiges Label-Adreßbyte
	QR	Label-Nummer
	S T	niederwertiges Label-Adreßbyte
Label (z)	UV	hochwertiges Label-Adreßbyte
TABLE+LABELS	WX	Label-Nummer
	•	
	•	
	•	
	•	
TABLE+FF		= ENDAD

Damit ist alles über den Aufbau des Symbol-Stacks gesagt. Nachdem alle für den Assembler wichtigen Pointer gesetzt sind, zeigt der Symbol-Stack-Pointer auf dieselbe Adresse, wie der Adreß-Pointer ENDAD (siehe auch Bild 5). Unmittelbar vor dem Label PASSA wird die Subroutine BEGIN aufgerufen. Sie ist bereits aus Kapitel 8 (Bild 7) bekannt. Nach Durchlaufen dieser Subroutine zeigt der Display-Pointer CURAD auf dieselbe Adresse, auf die der Pointer BEGAD zeigt. BEGAD ist bekanntlich die Startadresse der File.

PASSA

Jetzt ist der Computer beim Label PASSA angekommen. Ab diesem Label beginnt die erste Phase der Assemblierung. Zunächst verzweigt das Programm zur Subroutine OPLEN (Kapitel 8, Bild 19). Diese Subroutine berechnet die Länge einer Instruktion und legt das Ergebnis in der Speicherzelle BYTES ab. Es ist somit bekannt, um wieviele Plätze der Display-Pointer CURAD verschoben werden muß, wenn die folgende Instruktion zu untersuchen ist.

Die Instruktion LDA-(CURADL),Y mit Y=ØØ lädt den OP-Code der Instruktion in den Accu, auf die momentan der Display-Pointer CURAD zeigt. Die beiden folgenden Instruktionen CMP # FF und BNE testen, ob es sich um den Pseudo-OP-Code eines Labels handelt oder nicht. Ist es kein Pseudo-OP-Code FF, verzweigt das Programm in **Bild 4** zum Label NXTINS (Next Instruction). In der Subroutine NEXT (Kapitel 8, Bild 10) setzt der Computer den Display-Pointer CURAD auf die folgende Instruktion, indem er zum alten Inhalt von CURAD den Inhalt der Speicherzelle BYTES addiert. Während der Subroutine NEXT wird auch noch geprüft, ob sich der Display-Pointer CURAD, der immer auf den OP-Code einer Instruktion zeigt, noch in der File zwischen BEGAD und CEND befindet, oder schon über CEND hinauszeigt. Befindet sich der Display-Pointer CURAD noch innerhalb der File zwischen BEGAD und CEND, untersucht der Computer die folgende Instruktion auf den Pseudo-OP-Code FF und verzweigt zurück zum Label PASSA. Die Instruktionslänge der folgenden Instruktion wird wieder in der Subroutine OPLEN berechnet und anschließend prüft der Computer, ob es sich um den Pseudo-OP-Code FF handelt. Der oben beschriebene Vorgang wiederholt sich.
Trifft aber der Computer auf den Pseuod-OP-Code FF, dann führt er die Programmverzweigung BNE-NEXTINS nach der Instruktion CMP # FF nicht aus, sondern kümmert sich um den Label in der File. Wie geschieht dies nun im Einzelnen? Dazu nehmen wir **Bild 4** und **Bild 5** zur Hand. Dabei zeigt Bild 4 die erste Phase der Assemblierung und Bild 5 einen Ausschnitt aus dem **Symbol-Stack:**

1. Mit der Instruktion INY wird der Inhalt des Y-Registers von ØØ auf Ø1 inkrementiert. Die folgende Instruktion LDA-(CURADL),Y hat zur Folge, daß der Accu mit dem Inhalt der Speicherzelle, auf die CURAD+1 zeigt, geladen wird. Die dem Pseudo-OP-Code folgende Label-Nummer steht somit im Accu der CPU.
2. Der Inhalt der Speicherzelle LABELS wandert ins Y-Register. Die anschließende Instruktion STA-(TABLEL),Y bringt die Label-Nummer des momentan untersuchten Labels auf den Symbol-Stack. Die Symbol-Stack-Adresse, an der die Label-Nummer abgelegt wird, ist die Adresse, auf die der momentane Symbol-Stack-Pointer TABLE+LABELS zeigt. Da am Anfang des Assemblierens der Inhalt der Speicherzelle LABELS gleich FF ist, und zu diesem Zeitpunkt der Symbol-Stack-Pointer TABLE auf die Adresse ENDAD-FF zeigt, kommt die erste Label-Nummer an die Adresse auf den Symbol-Stack, auf die der Endadreß-Pointer ENDAD zeigt. Diese Speicherzelle und fünf weitere haben wir in Kapitel 5 für den Assembler reserviert, damit kein Teil des mit dem Editor eingegebenen Programms vom Symbol-Stack überschrieben werden kann.
3. DEY
   LDA-CURADH
   STA-(TABLEL),Y
   Der Junior-Computer hat in der File einen Label mit dem Pseudo-OP-Code FF gefunden. Der Display-Pointer CURAD zeigt noch immer auf den Pseudo-OP-Code des Labels. Nach der Dekrementierung des Y-Registers wird das hochwertige Adreßbyte des Display-Pointers CURAD in den Accu geladen. Dabei ist der Inhalt der Speicherzelle CURADH

das hochwertige Adreßbyte der Adresse, an der sich der Pseudo-OP-Code des Labels befindet. Dieses hochwertige Adreßbyte der Label-Adresse legt der Computer nun auf dem Symbol-Stack ab. Die Symbol-Stack-Adresse, an der das hochwertige Adreßbyte des Labels abgelegt wird, ist TABLE+LABELS−1. Somit liegt das hochwertige Adreßbyte der Label-Adresse einen Speicherplatz über der zuvor abgelegten Label-Nummer.

4. DEY
   LDA-CURADL
   STA-(TABLE),Y
   Jetzt lädt der Computer das niederwertige Adreßbyte des Display-Pointers CURAD in den Accu. Anschließend wird dieses auf dem Symbol-Stack abgelegt. Die Symbol-Stack-Adresse, an der das niederwertige Adreßbyte der Label-Adresse abgelegt wird, ist TABLE+LABELS−2. Somit liegt das niederwertige Adreßbyte der Label-Adresse einen Speicherplatz über dem zuvor abgelegten hochwertigen Adreßbyte des Labels.
5. DEY
   STY-LABELS
   Das Y-Register ist jetzt hintereinander dreimal dekrementiert worden. Oder anders gesagt, der Symbol-Stack-Pointer hat sich somit um drei Speicherplätze von unten nach oben geschoben. TABLE+LABELS zeigt jetzt wieder auf eine Adresse des Symbol-Stacks, an der die nächste Label-Nummer abgelegt wird, falls in der File noch ein weiterer Label mit dem Pseudo-OP-Code FF vorhanden ist.
6. JSR-UP
   JSR-RECEND
   Alle benötigten Informationen des Labels sind jetzt auf dem Symbol-Stack abgelegt. Der Label ist somit überflüssig und kann aus der File zwischen BEGAD und CURAD entfernt werden. Das Entfernen des Labels übernimmt die bekannte Subroutine UP (Kapitel 8, Bild 17). Die Subroutine UP schiebt den Datenblock der File um drei Speicherzellen nach oben, der bei der dem Label folgenden Instruktion beginnt und bei CEND endet. Die File ist nach dieser Verschiebeoperation um drei Bytes kürzer geworden und der momentane Endadreß-Pointer CEND (Kapitel 8, Bild 14) wandert in der folgenden Subroutine RECEND ebenfalls um drei Speicherplätze nach oben.
7. JMP-PASSA
   Der Junior-Computer untersucht jetzt die übrigen Instruktionen in der File und überprüft diese auf den Pseudo-OP-Code FF. Die Label-Suchaktion wird also fortgesetzt. Findet der Computer wieder einen Label in der File, transportiert er die Label-Nummer und die Label-Adresse auf den Symbol-Stack. Anschließend wird der Label wieder aus der File entfernt.
   Handelt es sich bei der untersuchten Instruktion um keinen Label, verzweigt das Assembler-Programm (Bild 4) zum Label NEXTINS und springt somit von Instruktion zu Instruktion, bis wieder ein Label gefunden ist. Während der Subroutine NEXT, die den Display-Pointer CURAD auf den OP-Code der folgenden Instruktion setzt, überprüft der Computer, ob CURAD schon über CEND hinausgewachsen ist. Hat

der Display-Pointer CURAD das Fileende erreicht, ist die erste Phase der Assemblierung abgeschlossen und der Junior-Computer gelangt nach dem Durchlaufen der Subroutine BEGIN zum Label PASSB. Die zweite Phase der Assemblierung beginnt.

Aus Kapitel 5 ist bekannt, daß am Ende eines eingetippten Programms ENDAD und CEND frei bleiben müssen. Das heißt, ist man sich nicht sicher, ob das eingetippte Programm zu lang geworden ist, muß man prüfen, ob der Pointer CEND noch mindestens sechs Speicherplätze über dem Pointer ENDAD steht. Durch Nachsehen der Pointer-Inhalte in Page Zero läßt sich das Anwachsen des Pointers CEND leicht kontrollieren:
CENDL hat die Adresse ØØE8
CENDH hat die Adresse ØØE9 und
ENDADL hat die Adresse ØØE4
ENDADH hat die Adresse ØØE5.

Was hat es mit diesen sechs Speicherzellen auf sich? Während der ersten Phase des Assemblierens kommen bekanntlich die Label-Nummer und die absolute Adresse des Labels auf den Symbol-Stack. Das sind insgesamt drei Speicherzellen, die diese Information aufnehmen. Dabei beginnt der Symbol-Stack bei der Adresse, auf die der Pointer ENDAD zeigt. Nachdem der Label aus der File entfernt ist, wird die File um drei Bytes kürzer, das heißt, der Pointer CEND wandert um drei Bytes nach oben. Wird ein weiterer Label aus der File entfernt, ist anscheinend wieder Platz auf dem Symbol-Stack, die Label-Nummer und die absolute Adresse des Labels darauf abzulegen. Die Frage ist also, warum sechs Speicherplätze, wenn anscheinend drei genügen? Als Antwort darauf sehen wir uns nochmals die Subroutine UP in Kapitel 8, Bild 17 an. Diese Subroutine entfernt die Labels aus der File. Ein Label ist bekanntlich drei Bytes lang. Beim Entfernen eines Labels aus der File wird deshalb der dem Label folgende Datenblock um drei Bytes nach oben geschoben. Bild 18 in Kapitel 8 zeigt diesen Vorgang: Beim Schieben des Datenblocks nach oben transportiert die Subroutine UP auch drei weitere Bytes hinter dem Pointer CEND nach oben. Diese drei Bytes sind für das Assembler-Programm überflüssig. Wären deshalb zwischen den beiden Pointern CEND und ENDAD nur drei Speicherplätze frei, würden am Ende der File die letzten Daten vor CEND mit unerlaubten Daten überschrieben werden.

Bild 6 zeigt einen Teil des Symbol-Stacks nach der ersten Phase des Assemblierens des Programms aus Kapitel 5, Bilder 1 . . . 4. Alle sieben Label sind auf dem Symbol-Stack mit der Label-Nummer und der dazugehörenden absoluten Adresse abgelegt. Dabei ist deutlich zu erkennen, daß der Symbol-Stack von unten nach oben mit Daten gefüllt wird. Der Label am "Boden" des Symbol-Stacks ist zuerst aus der File entfernt worden. Seine Nummer ist 1Ø und er befand sich an der Adresse ØØØ. Der zuletzt assemblierte Label hat die Label-Nummer 16 und befand sich an der absoluten Adresse Ø24C. Er war in der File der letzte Label vor dem EOF-Charakter und liegt deshalb ganz oben auf dem Symbol-Stack. Zu diesem Zeitpunkt zeigt der Symbol-Stack-Pointer TABLE+LABELS auf die Adresse Ø2EA.

PASSB

Hinter dem Label PASSB beginnt die zweite Phase des Assemblierens.

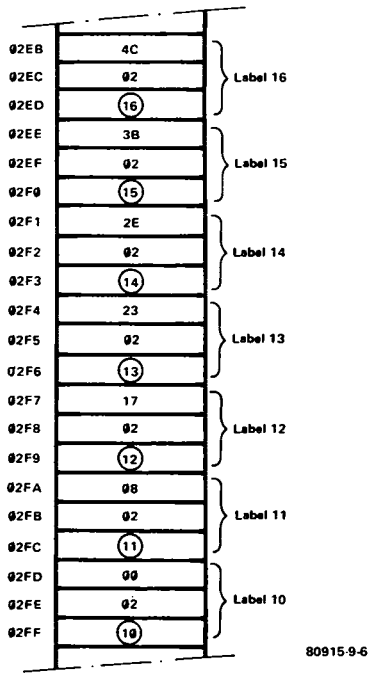


Bild 6. Aus Kapitel 5 ist bereits das Arbeiten mit dem Editor und Assembler bekannt. Dieses Bild zeigt, wie der Symbol-Stack nach der ersten Phase des Assemblierens bei diesem Beispiel aussieht.

Bild 7 zeigt das detaillierte Flußdiagramm dieser zweiten Phase. Zu Beginn ist der Display-Pointer CURAD gleich dem Adreß-Pointer BEGAD. Das Gleichsetzen dieser beiden Pointer war die letzte Aktion der ersten Phase der Assemblierung (Bild 5). Während der zweiten Phase des Assemblierens prüft der Junior-Computer wieder alle Instruktionen, die jetzt in der File enthalten sind. Während der erste Phase interessierte sich der Computer ausschließlich für die Labels, die durch den Pseudo-OP-Code FF gekennzeichnet waren. In der zweiten Phase des Assemblierens sind die Labels entfernt und der Computer konzentriert sich ausschließlich auf Instruktionen, die assembliert werden müssen. Das sind alle Instruktionen, hinter deren OP-Code eine Label-Nummer steht:

* JMP-Instruktionen
* JSR-Instruktionen und
* Branch-Instruktionen.

Hinter diesen Instruktionen können jedoch auch absolute Adressen beziehungsweise Offsets stehen. Allerdings müssen dann die bedingten und nichtbedingten Sprungbefehle bei der zweiten Phase der Assemblierung gesondert behandelt werden. Dabei gelten folgende Zusammenhänge:
Wie bei PASSA (Bild 4) beginnt PASSB mit dem Aufruf der Subroutine

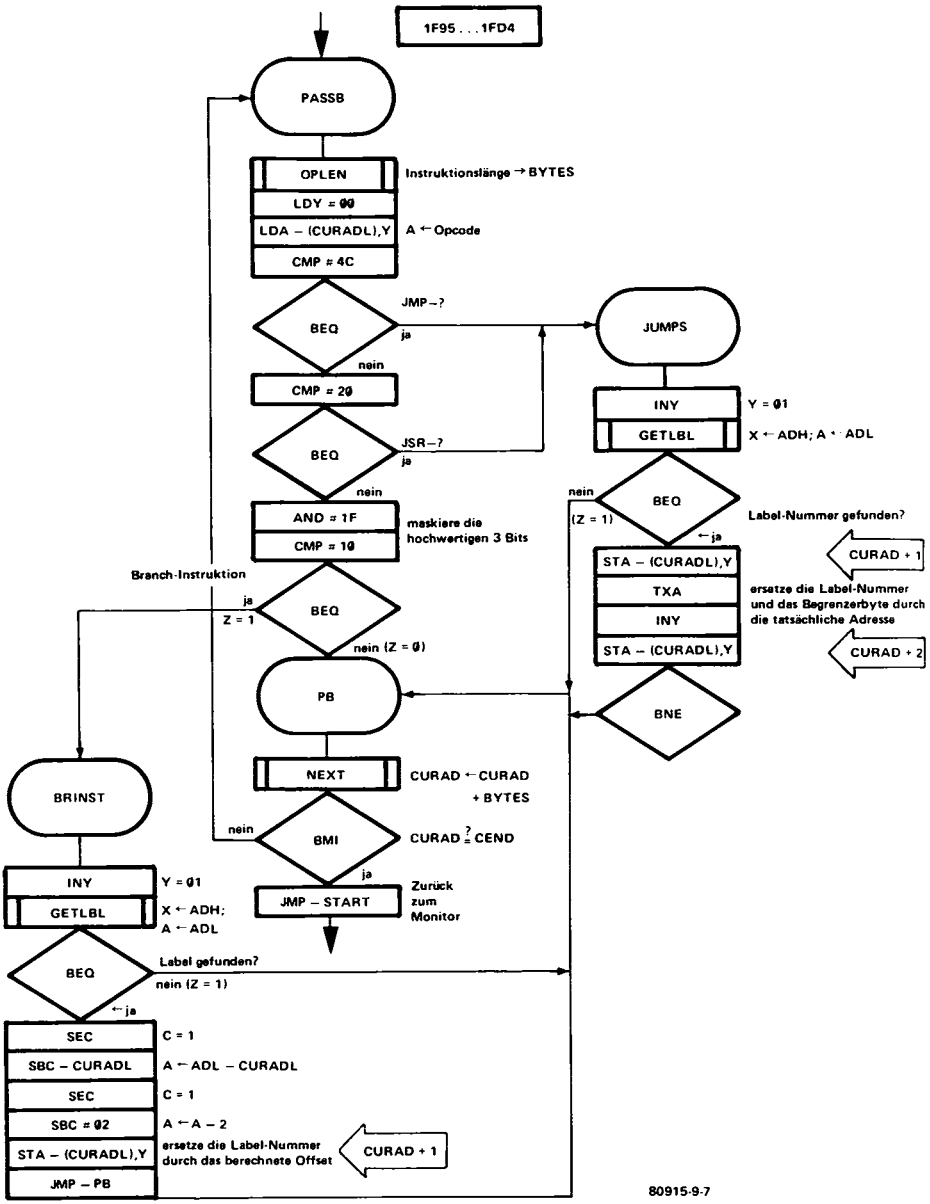


Bild 7. Das detaillierte Flußdiagramm der zweiten Phase des Assemblers.

OPLEN. Somit wird bekannt, um wieviele Speicherplätze CURAD nach unten zu verschieben ist, um auf den OP-Code der folgenden Instruktion zu zeigen. Die Lade-Instruktion LDA-(CURADL),Y, mit Y=ØØ transportiert den OP-Code der zu untersuchenden Instruktion in den Accu. Dann filtert der Computer die beiden Sprung-Instruktionen JMP und JSR sowie die Branch-Instruktionen aus:

1. CMP # 4C
   BEQ filtert die Sprung-Instruktion JMP mit dem OP-Code 4C aus.
2. CMP # 2Ø
   BEQ filtert die Sprung-Instruktion JSR mit dem OP-Code 2Ø aus.
3. AND 1F
   CMP # 1Ø
   BEQ filtert alle Branch-Instruktionen mit den OP-Codes 1Ø, 3Ø, FØ, DØ, 9Ø, BØ, 5Ø und 7Ø. Zu diesen OP-Codes gehören der Reihe nach die Mnmonics: BPL, BMI, BEQ, BNE, BCC, BCS, BVC und BVS. Durch das Maskieren mit 1F sind die niederwertigen fünf Bits des Accus unverändert, die hochwertigen drei Bits jedoch null. Betrachtet man die OP-Codes der bedingten Sprung-Instruktionen, fällt auf, daß das niederwertige OP-Code-Nibble immer null ist. Somit führt das Maskieren des OP-Codes einer beliebigen Branch-Instruktion mit 1F immer zum Ergebnis 1Ø. Durch den Vergleich CMP # 1Ø kann somit der Computer feststellen, ob der momentan untersuchte OP-Code zu einer Branch-Instruktion gehört oder nicht.

Handelt es sich um **keine Sprung- oder Branch-Instruktion**, muß der Computer sie nicht assemblieren. Somit kann sich der Computer nach dem Label PB um die folgende Instruktion in der File kümmern. Ein Aufruf der Subroutine NEXT setzt den Display-Pointer CURAD auf den OP-Code der folgenden Instruktion. Ist der Computer noch nicht am File-Ende angekommen, verzweigt das Assembler-Programm zum Label PASSB und kümmert sich nach der Berechnung der Instruktionslänge (OPLEN) um die folgende Instruktion. Der soeben beschriebene Vorgang läuft abermals in gewohnter Weise ab.

Handelt es sich um **eine Sprung- oder Branch-Instruktion**, müssen diese assembliert werden. Dazu muß der Junior-Computer die tatsächliche absoluten Adressen hinter die OP-Codes der Sprung-Instruktionen setzen. Für die Branch-Instruktionen müssen noch die Offsets berechnet und anschließend hinter die OP-Codes gesetzt werden. Wie der Computer diese Aufgabe mit Hilfe des Symbol-Stacks bewältigt, erklärt jetzt der folgende Abschnitt.

JMP- und JSR-Instruktionen

Alle nichtbedingten Sprung-Instruktionen werden ab dem Label JUMPS assembliert. Dabei wird die echte Sprung-Adresse auf dem Symbol-Stack aufgespürt und anschließend als Operanden-Byte hinter den OP-Code der Sprung-Instruktion gesetzt.

Nach dem Inkrementieren des Y-Registers (Y ist jetzt Ø1) folgt die Subroutine GETLBL. Diese Subroutine in **Bild 8** besprechen wir noch später im Detail. Zunächst genügt es, folgendes zu wissen:

1. Die Subroutine GETLBL sucht auf dem Symbol-Stack diejenige Label-

Nummer auf, die hinter dem OP-Code der momentan assemblierten Sprung-Instruktion steht. Ist die Label-Nummer hinter dem OP-Code der Sprung-Instruktion tatsächlich auf dem Symbol-Stack vorhanden, dann hat der Computer Zugriff zu der Adresse, die zur gesuchten Label-Nummer gehört.

2. Die zur gesuchten Label-Nummer gehörende Adresse holt der Computer vom Symbol-Stack. Diese Adresse ist eine 16-Bit-Adresse, also zwei Bytes lang. GETLBL packt diese beiden Adreß-Bytes vor der Rückkehr aus der Subroutine in zwei interne CPU-Register:
   * Das hochwertige Adreß-Byte des Labels steht nach der Rückkehr aus der Subroutine im X-Register.
   * Das niederwertige Adreß-Byte des Labels steht nach der Rückkehr aus der Subroutine im Accu der CPU.
3. Kann der Computer die hinter dem momentan assemblierten OP-Code stehende Label-Nummer auf dem Symbol-Stack finden, kehrt er aus der Subroutine GETLBL mit gesetzter Z-Flag zurück.
4. Kann der Computer die hinter dem momentan assemblierten OP-Code stehende Label-Nummer **nicht** auf dem Symbol-Stack finden, kehrt er aus der Subroutine GETLBL mit zurückgesetzter N-Flag ins Assembler-Programm zurück.

Nehmen wir einmal an, der Computer habe in der Subroutine GETLBL die Label-Nummer auf dem Symbol-Stack gefunden, die hinter dem OP-Code der momentan assemblierten Sprung-Instruktion steht. Er kehrt mit dem hoch- und niederwertigen Adreßbyte des Labels, gespeichert im X-Register und Accu der CPU, aus der Subroutine zurück. Die Z-Flag ist gesetzt und bei der folgenden BEQ-Instruktion findet keine Verzweigung zum Label PB statt. Das Y-Register ist noch immer Ø1. Die Instruktion STA-(CURAD),Y legt das niederwertige Adreß-Byte der Label-Adresse hinter dem OP-Code der momentan assemblierten Sprung-Instruktion ab. Dann wird das hochwertige Adreß-Byte der Label-Adresse vom X-Register in den Accu kopiert und das Y-Register abermals inkrementiert. Die folgenden Schreibeoperationen STA-(CURADL),Y schreiben somit die tatsächliche Sprungadresse in die File.

Vor dem Assemblieren war die Sprung-Instruktion so in der File abgelegt: OP-Code XX ØØ. Dabei ist der OP-Code entweder 2Ø oder 4C und XX eine beliebige Label-Nummer. Als Begrenzer-Byte verwendeten wir ØØ.

Nach dem Assemblieren ist die Sprung-Instruktion mit den beiden richtigen Operanden-Bytes in der File abgelegt: OP-Code ADL, ADH. Am OP-Code der Instruktion hat sich nichts geändert. Aber die beiden Operanden-Bytes ADL (niederwertiges Byte der Label-Adresse = niederwertiges Byte der absoluten Sprung-Adresse) und ADH (hochwertiges Byte der Label-Adresse = hochwertiges Byte der absoluten Sprung-Adresse) haben die Label-Nummern XX und das Begrenzer-Byte ØØ überschrieben. Die gesamte Sprung-Instruktion ist somit assembliert. Hinter dem OP-Code der assemblierten Sprung-Instruktion steht jetzt der richtige Operand: die absolute Sprung-Adresse ADL und ADH, die der Computer vom Symbol-Stack holte.

Branch-Instruktionen

Entdeckt der Computer während der zweiten Phase des Assemblierens eine

Branch-Instruktion in der File, muß er das zu dieser Branch-Instruktion gehörende Offset berechnen. Dazu muß die Label-Nummer, die hinter dem OP-Code der Branch-Instruktion steht, aus der File entfernt werden. Branch-Instruktionen werden ab dem Label BRINST assembliert. Nach der Inkrementierung des Y-Registers (Y ist jetzt Ø1) wird die Subroutine GETLBL aufgerufen. In dieser Subroutine holt der Computer die absolute Adresse des Labels, zu der die Branch-Instruktion verzweigen soll, vom Symbol-Stack. Kann der Computer auf dem Symbol-Stack die Label-Nummer finden, die hinter dem OP-Code der momentan assemblierten Branch-Instruktion steht, kehrt er aus der Subroutine GETLBL mit zurückgesetzter Z-Flag ins Assembler-Programm zurück. Selbstverständlich sind in diesem Fall das hoch- und das niederwertige Adreß-Byte der Label-Adresse im X-Register und Accu abgelegt. Wie kann jetzt der Computer das tatsächliche Offset der Branch-Instruktion berechnen und anschließend hinter dem OP-Code ablegen? Um diese Frage zu beantworten, muß bekannt sein, was der Computer zu diesem Zeitpunkt über die momentan assemblierte Branch-Instruktion weiß:

1. Nach der Rückkehr aus der Subroutine GETLBL kennt der Computer die Label-Adresse, zu der die bedingte Programmverzweigung (Branch-Instruktion) führen soll. Diese Label-Adresse ist im X-Register und Accu der CPU abgelegt. Die beiden Adreß-Bytes im Inneren der CPU sind die **Ziel-Adresse**, zu der die Programmverzweigung führen soll.
2. Durch den Display-Pointer CURAD kennt der Computer die Adresse, an der sich der OP-Code der momentan assemblierten Branch-Instruktion befindet. Diese Adresse ist die sogenannte **Quellen-Adresse**, von der aus die Verzweigung erfolgen soll. Die Quellen-Adresse ist der Inhalt des Display-Pointers CURAD, der in den Speicherzellen ØØE6 und ØØE7 in Page Zero abgelegt ist.
3. Aus der Ziel- und Quellen-Adresse kann der Computer das tatsächliche Offset berechnen:
   Offset = Ziel-Adresse minus Quellen-Adresse minus Ø2. Von der Differenz Ziel-Adresse minus Quellen-Adresse muß noch Ø2 subtrahiert werden, da der Programmzähler der CPU nach der Dekodierung einer Branch-Instruktion auf den folgenden OP-Code zeigt. Ist somit das Offset berechnet, muß dieses noch hinter dem OP-Code der momentan assemblierten Branch-Instruktion eingesetzt werden. Die Schreibeoperation STA-(CURADL),Y hinter dem Label BRINST überschreibt die Label-Nummer hinter dem OP-Code der Branch-Instruktion mit dem tatsächlichen Offset. Auch die Branch-Instruktion ist somit assembliert.

Nun ist das Haupt-Programm des Assemblers bekannt. Der Assembler bearbeitet eine mit dem Editor eingegebene File in zwei Phasen:

1. Phase: Die absoluten Adressen der Labels werden auf dem Symbol-Stack mit der dazugehörenden Label-Nummer abgelegt. Danach entfernt der Computer den Label aus der File, indem er den Datenblock, der dem Label folgt, um drei Speicherzellen nach oben schiebt. Der Label wird somit aus der File entfernt. Bei dieser Schiebeoperation des Datenblocks werden auch die folgenden Label in der File um drei Speicherzellen nach oben verschoben. Während der ersten Phase der Assemblierung ändern somit die OP-Codes der noch nicht aus der File entfernten Labels ständig

ihre Adressen. Informationen über Labels, die aus der File bereits entfernt sind, enthält der Symbol-Stack. Adressen von Labels, die auf dem Symbol-Stack abgelegt sind, ändern sich während der Assemblierung nicht mehr.
2. Phase: Nachdem alle Labels aus der File entfernt sind und alle Information über diese Labels auf dem Symbol-Stack abgelegt ist, beginnt die zweite Phase des Assemblierens. Während dieser Phase überprüft der Computer alle Instruktionen, hinter deren OP-Code sich eine Label-Nummer befindet. Dazu müssen die OP-Codes von JMP-, JSR- und Branch-Instruktionen aus der File herausgefiltert werden. Bei den Sprung-Instruktionen sucht der Computer die Label-Nummer, die sich hinter dem OP-Code der momentan assemblierten Instruktion befindet, auf dem Symbol-Stack auf. Somit hat er Zugriff auf die Label-Adresse. Die Label-Adresse ist gleich der absoluten Sprung-Adresse und wird hinter dem OP-Code der Sprung-Instruktion in die File eingesetzt.
Bei Branch-Instruktionen muß das Offset noch berechnet werden. Dieses Offset kann der Computer aus der Quellen- und der Ziel-Adresse berechnen. Ist das Offset berechnet, wird es hinter dem OP-Code der Branch-Instruktion in die File eingesetzt.

Die Subroutine GETLBL

GETLBL ist die zentrale Subroutine des Assemblers. In dieser Subroutine holt der Computer die Label-Adresse vom Symbol-Stack, wenn eine JMP-, JSR- oder Branch-Instruktion assembliert werden muß. Die Aufgabe dieser Subroutine läßt sich in folgende Punkte zusammenfassen:

1. Suche die Label-Nummer auf dem Symbol-Stack.
2. Ist die Label-Nummer gefunden, dann hole die Label-Adresse vom Symbol-Stack. Setze das hochwertige Adreßbyte der Label-Adresse in das X-Register der CPU und das niederwertige Adreßbyte in den Accu.
3. Ist die Label-Nummer nicht auf dem Symbol-Stack vorhanden, kehre mit gesetzter Z-Flag ins Assembler-Programm zurück.

Bild 8 beginnt mit dem Laden der Label-Nummer in den Accu, die sich hinter dem momentan assemblierten OP-Code befindet:
LDA-(CURADL),Y mit Y=Ø1.
Anschließend wird der Inhalt des Y-Registers mit FF geladen. Bevor wir die Besprechung der Subroutine GETLBL fortsetzen, sollte man sich folgendes ins Gedächtnis zurückrufen:

* Die Adresse, auf die der Symbol-Stack-Pointer TABLE+LABELS zeigt, ist die oberste, nichtbesetzte Speicherzelle des Symbol-Stacks. Während der zweiten Phase des Assemblierens ist der Symbol-Stack von der Adresse TABLE+FF=ENDAD auf die Adresse TABLE+LABEL angewachsen.
* TABLE+FF ist der "Boden" des Symbol-Stacks, während TABLE+LABELS die "Spitze" des Symbol-Stacks ist. Zwischen diesen beiden Adressen befinden sich alle Label-Informationen, die während PASSA auf dem Symbol-Stack abgelegt wurden.
* Die Suche einer Label-Nummer auf dem Symbol-Stack beginnt am "Boden" bei der Adresse TABLE+FF und endet am oberen Ende des Stacks bei der Adresse TABLE+LABELS.

Hinter dem Label SYMA prüft der Computer, ob der Symbol-Table-Pointer schon auf die Spitze des Symbol-Stacks zeigt. Diesen Sachverhalt prüft

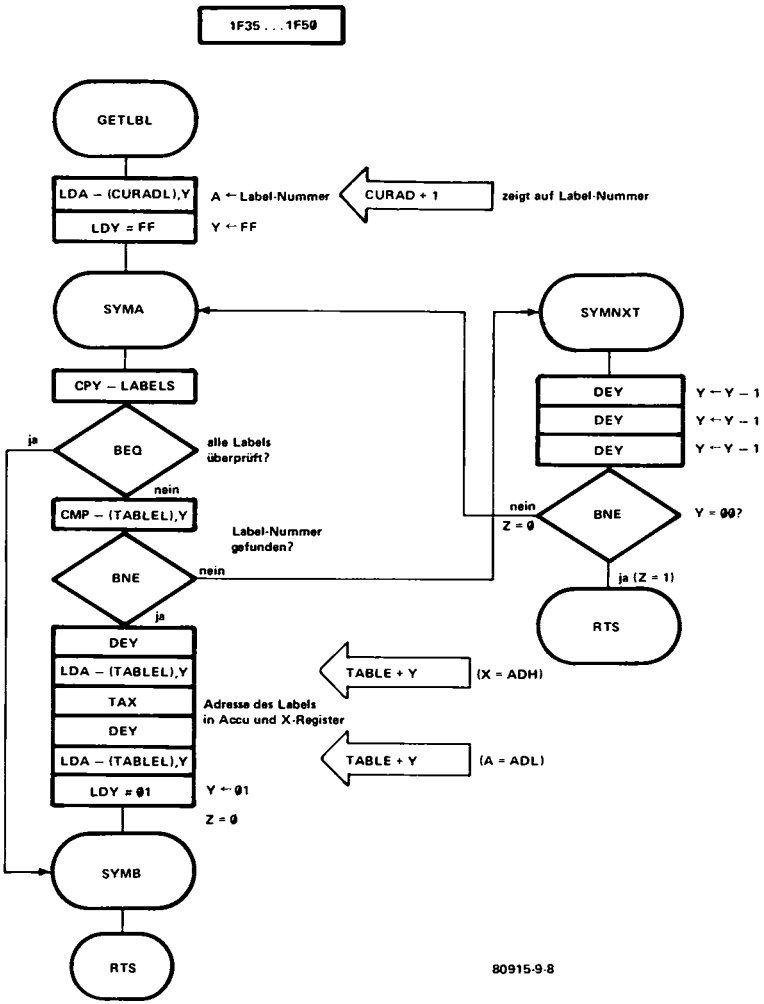


Bild 8. GETLBL ist die zentrale Subroutine des Assemblers während der zweiten Phase. In dieser Routine spürt der Computer die Label-Nummer der momentan zu assemblierenden Instruktion auf und setzt die zur Label-Nummer gehörende Adresse in zwei interne CPU-Register:
Der Accu enthält ADL und das X-Register ADH.

die Instruktion CPY-LABELS. Die folgende Vergleichsoperation CMP-(TABLEL),Y führt den Vergleich einer Label-Nummer auf dem Symbol-Stack mit einer Label-Nummer aus der File durch. Wie geht das vor sich? Bevor dieser Vergleich durchgeführt wird, haben sich im Computer folgende Bedingungen eingestellt:

1. Während PASSB ist der Computer in der File auf eine Instruktion gestoßen, hinter deren OP-Code sich eine Label-Nummer befindet. Diese Instruktion muß assembliert werden. Diese Aufgabe übernimmt die Subroutine GETLBL.
2. Die Label-Nummer, die sich hinter dem OP-Code der zu assemblierenden Instruktion befindet, wird zunächst in den Accu geladen.
3. Der Symbol-Table-Pointer zeigt jetzt auf die Adresse TABLE+FF. An dieser Adresse ist die erste Label-Nummer auf dem Symbol-Stack abgelegt worden. Zwischen der Label-Nummer im Accu und der Label-Nummer auf dem Symbol-Stack führt der Computer einen Vergleich durch. Die Instruktion, die diesen Vergleich ausführt, ist CMP-(TABLEL),Y. Sind beide Label-Nummern ungleich, verzweigt das Programm zum Label SYMNXT. Hinter diesem Label wird das Y-Register dreimal dekrementiert.
4. Da nach dieser dreimaligen Dekrementierung das Y-Register noch immer ungleich null ist, verzweigt das Programm zum Label SYMA zurück. Dort wird wieder geprüft, ob schon alle Label-Nummern auf dem Symbol-Stack mit der Label-Nummer im Accu verglichen wurden. Da zuvor das Y-Register dreimal dekrementiert wurde, zeigt der Symbol-Table-Pointer TABLE+Y auf eine weitere Label-Nummer, die auf dem Symbol-Stack abgelegt ist. Auch diese Label-Nummer wird mit der Label-Nummer im Accu verglichen.
5. Sind beide Label-Nummern abermals ungleich, wird das Y-Register wieder dekrementiert und der Symbol-Stack-Pointer zeigt auf eine Symbol-Stack-Adresse, an der eine weitere Label-Nummer abgelegt ist. Sind aber beide Label-Nummern bei der Vergleichsoperation CMP-(TABLEL),Y gleich, dann ist die gesuchte Label-Nummer auf dem Symbol-Stack gefunden.
6. Die Label-Nummer auf dem Symbol-Stack ist gefunden. Die zu dieser Label-Nummer gehörende Adresse kann der Computer jetzt vom Symbol-Stack holen. Zunächst wird das hochwertige Adreßbyte des Labels vom Symbol-Stack geholt. Die Instruktionen dafür sind:
   DEY
   LDA-(TABLEL),Y
   TAX
   Zunächst wird der Symbol-Table-Pointer um eins nach oben geschoben und zeigt somit auf das hochwertige Adreßbyte der Label-Adresse. Dieses Adreßbyte kommt in den Accu und gelangt dann ins X-Register der CPU. Ein weiteres Dekrementieren des Y-Registers hat zur Folge, daß der Symbol-Table-Pointer um eine Speicherzelle nach oben geschoben wird. Er zeigt somit auf die Speicherzelle, in der das niederwertige Adreßbyte der Label-Adresse abgelegt ist. Dieses niederwertige Adreßbyte der Label-Adresse wird in den Accu der CPU geladen. Die Instruktionen dafür waren:
   DEY
   LDA-(TABLEL),Y
   Somit steht das hochwertige Adreßbyte der Label-Adresse im X-Register und das niederwertige Adreßbyte im Accu. Diese Forderung muß erfüllt sein, wenn der Prozessor aus der Subroutine GETLBL in die Assembler-Routine PASSB zurückkehrte.

7. Eine weitere Forderung muß nach der Rückkehr in die Assembler-Routine die Subroutine GETLBL erfüllen:
   * Hat der Computer auf dem Symbol-Stack eine Label-Nummer gefunden, die der Label-Nummer der momentan assemblierten Instruktion gleich war, muß die Z-Flag zurückgesetzt sein.
   * Steht auf dem Symbol-Stack keine Label-Nummer mehr an, die der Label-Nummer der momentan assemblierten Instruktion gleicht, muß die Z-Flag gesetzt sein.

   Das Zurücksetzen der Z-Flag erledigt die Instruktion LDY # Ø1 vor dem Label SYMB. Diese Instruktion kann nur dann ausgeführt werden, wenn der Computer auf dem Symbol-Stack eine Label-Nummer gefunden hat.
   Hat aber der Computer den gesamten Symbol-Stack nach Label-Nummern abgesucht und kann dort die gesuchte Label-Nummer der momentan assemblierten Instruktion nicht finden, muß die Z-Flag gesetzt werden. Befindet sich die Label-Nummer nicht auf dem Symbol-Stack, dann wird die Suchaktion abgebrochen, sobald der Symbol-Stack-Pointer auf die Spitze des Symbol-Stacks zeigt. Nur in diesem Fall stellt sich die Situation ein, daß TABLE+Y und TABLE+LABELS auf dieselbe Adresse zeigen. Ein Vergleich der Indizes LABELS und Y gibt deshalb darüber Auskunft, ob die Label-Nummer auf dem Symbol-Stack vorhanden ist oder nicht. Hinter dem Label SYMA findet diese Vergleichsoperation statt:
   CPY-LABELS
   Ist die Label-Nummer nicht auf dem Symbol-Stack vorhanden, ist das Ergebnis dieser Vergleichsoperation null. Dadurch wird die Z-Flag in der gewünschten Weise gesetzt und der Computer kehrt unmittelbar über den Label SYMB in die Assembler-Routine zurück.

Nun ist auch der Assembler bekannt. Editor und Assembler sind eng miteinander verknüpft: Beide Programme verwenden dieselben Subroutinen. Nachdem diese Subroutinen bekannt sind, kann sie der Programmierer leicht in seine eigenen Programme einbauen. Am Ende dieses Buches ist das Source-Listing aller beschriebenen Programme abgedruckt. Daraus lassen sich wertvolle Kommentare entnehmen, wenn man selbst mit den Subroutinen des Monitors, Editors oder Assemblers arbeiten möchte. Auch für die Muster-Programme aus den Kapiteln 5 und 6 sind Source-Listings angefertigt worden.

Die EPROM-Routine BRANCH

Im Buch Junior-Computer 1 ist bei mehreren Gelegenheiten bereits darauf hingewiesen, daß ein im EPROM vorhandenes Programm die Offsets von Branch-Instruktionen berechnet. Dieses Programm heißt Branch und beginnt bei der Adresse 1FD5. Branch (Bild 9) ist eine unendliche Programmschleife, die nur über die Tasten RST, ST oder einen externen NMI oder IRQ verlassen werden kann.
Unmittelbar nach dem Start der Branch-Routine zeigt das Display ØØ ØØ ØØ. Jetzt gibt der Programmierer das niederwertige Adreßbyte der Adresse ein, an der sich der OP-Code der Branch-Instruktion befindet. Nach der Ein-

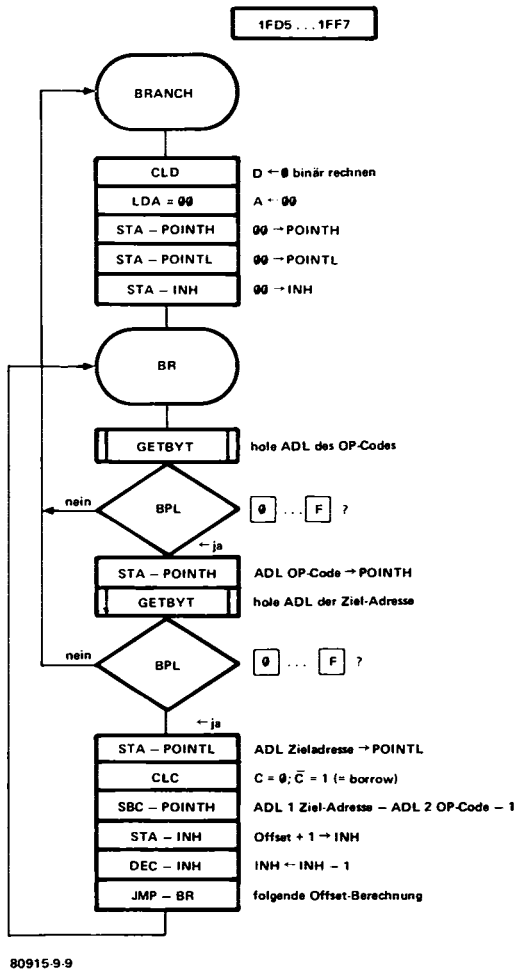


Bild 9. In Buch 1 ist bereits die Offset-Berechnung erläutert. Das erforderliche Programm Branch steht im EPROM des Junior-Computers zur Verfügung. Das detaillierte Flußdiagramm zeigt dieses Bild.

gabe des niederwertigen Adreßbytes wird dieses auf den linken beiden 7-Segment-Displays abgebildet.
Dann erfolgt die zweite Dateneingabe: Der Programmierer gibt das niederwertige Adreßbyte der Adresse ein, zu der die Branch-Instruktion führen soll. Nach der Eingabe dieses Adreßbytes wird es auf den beiden mittleren Anzeigen abgebildet. Die beiden rechten Displays des Junior-Computers zeigen das Offset an.

Bei der Eingabe der Adressen muß nur das niederwertige Adreßbyte von Quellen-Adresse und Ziel-Adresse eingegeben werden, da die CPU 6502 über einen Bereich von +127 Schritten vorwärts und −128 Schritten rückwärts verzweigt.
Nach der Berechnung eines Offsets läßt sich das Display durch Drücken einer beliebigen Kommando-Taste wieder auf ØØ ØØ ØØ setzen. Dasselbe gilt auch während der Eingabe der beiden niederwertigen Adreßbytes.
Bei der Besprechung der Branch-Routine können wir uns kurz fassen, da für die Berechnung des Offsets dasselbe gilt, wie beim Assembler in Kapitel 9, Bild 7.
Am Anfang werden die Displaybuffer INH, POINTL und POINTH mit ØØ geladen und der Computer auf binäre Rechenweise geschaltet. Hinter dem Label BR rufen wir die Subroutine GETBYT auf. Diese Subroutine legt zwei Daten-Tasten des Tastenfeldes im Accu ab und kehrt in diesem Fall mit gesetzter N-Flag in die Branch-Routine zurück. Wird dabei irgendeine Kommando-Taste bedient, ist die N-Flag zurückgesetzt und der Computer verzweigt zum Label BRANCH. Das Display wird dabei wieder zurückgesetzt. Nachdem die beiden Tasten in den Accu gelesen sind, wird der Accu-Inhalt in dem Displaybuffer POINTH abgelegt. POINTH enthält somit das niederwertige Adreßbyte der Adresse, an der sich der OP-Code der Branch-Instruktion befindet. Beim zweiten Aufruf der Subroutine GETBYT wird das niederwertige Adreßbyte der Adresse in den Accu gelesen, zu der die Programmverzweigung führen soll. Ist das niederwertige Adreßbyte der Zieladresse im Displaybuffer POINTL abgelegt, beginnt der Junior-Computer mit der Berechnung des Offsets.
Im Accu steht also das niederwertige Adreßbyte der Ziel-Adresse, zu der die Verzweigung führen soll. Davon zieht der Computer das niederwertige Adreßbyte der Adresse ab, an der der OP-Code der Branch-Instruktion steht. Da vor der Subtraktion die C-Flag zurückgesetzt ist (was an sich verboten ist!) ist das Ergebnis um eins zu klein. Das falsche Offset gelangt zum Displaybuffer INH und wird dort abermals dekrementiert. Erst jetzt steht im Displaybuffer INH das richtige Offset. Dann verzweigt das Programm zum Label BR zurück und verzweigt dort während der Subroutine GETBYT das niederwertige Adreßbyte der Ziel- und Quellen-Adresse sowie das berechnete Offset zwischen diesen Adressen an.

Für spitzfindige Leser:

Offset-Berechnung bei der Branch-Routine:
Offset = Ziel-Adresse minus Quellen-Adresse minus eins minus eins
Offset-Berechnung bei der Assembler-Routine:
Offset = Ziel-Adresse minus Quellen-Adresse minus zwei.
Buch zwei ist damit beendet.
Beide Bücher haben gezeigt, wie sich mit einem kleinen Computer sinnvolle Programme einfach anfertigen lassen. Die Beschreibung des Standart-Junior-Computers ist somit abgeschlossen. Das soll aber nicht heißen, daß auch das Projekt "Junior-Computer" abgeschlossen ist. Am Ende dieses Buches befindet sich ein Anhang. Ein Subroutinen-Summary und das Source-Listing mit ausführlichen Kommentaren sind darin enthalten.

Der Anhang dürfte fürs erste eine wahre Fundgrube für den "Software-Bastler" sein. Wer sich bis jetzt durch die beiden Junior-Computer Bücher gearbeitet hat, dem wird es nicht schwer fallen, beim folgenden Buch 3 wieder dabei zu sein: Durch eine umfangreiche, aber billige Hardware-Erweiterung entwickelt sich der Junior-Computer zu einem Personal-Computer!

Anhang 1

Subroutinen-Summary

ADCEND
Aufruf: JSR-ADCEND oder JSR-1ED8 oder 2Ø D8 1E
Die Subroutine ADCEND schiebt den Pointer CEND um ein, zwei oder drei Speicherplätze nach unten. Der Inhalt der Speicherzelle BYTES ØØF6 entscheidet, um wieviele Adressen CEND nach unten bewegt wird.

AK
Aufruf: JSR-AK oder JSR-1DB1 oder 2Ø B1 1D
Die Subroutine AK ist an die Subroutinen SCAND bzw. SCAN/SCANDS "angehängt". Beim Aufruf der Subroutinen SCAND/SCANDS wird automatisch auch die Subroutine AK durchlaufen. AK fragt das Keyboard des Junior-Computers ab. Ist irgendeine Taste des Tastenfeldes gedrückt, kehrt der Prozessor mit A ≠ ØØ aus dieser Subroutine zurück. Ist jedoch keine Taste gedrückt, hat der Accu den Inhalt A = ØØ.

BEGIN
Aufruf: JSR-BEGIN oder JSR-1ED3 oder 2Ø D3 1E
Die Subroutine BEGIN macht den CURAD-Pointer (Display-Pointer) gleich dem BEGAD-Pointer. Die symbolische Schreibweise ist CURAD:= BEGAD

CONVD
Aufruf: JSR-CONVD oder JSR-1DDF oder 2Ø DF 1D
Die Subroutine CONVD erledigt die zeitliche Ablaufsteuerung des Displays. Mit Hilfe einer Lookup Table wird der Inhalt der Displaybuffer in den 7-Segment-Code umgewandelt.

DOWN
Aufruf: JSR-DOWN oder JSR-1EA6 oder 2Ø A6 1E
Die Subroutine DOWN bewegt einen Datenblock zwischen den Pointern CURAD und CEND um ein, zwei oder drei Speicherplätze nach unten. Der Inhalt der Speicherzelle BYTES ØØF6 entscheidet, um wieviele Speicherplätze der Datenblock nach unten bewegt wird.

FILLWS
Aufruf: JSR-FILLWS oder JSR-1E47 oder 2Ø 47 1E
Die Subroutine FILLWS transportiert eine Instruktion vom Displaybuffer in den Arbeitsspeicher des Junior-Computers. Dabei wird im Arbeits-

speicher, je nach Instruktionslänge ein, zwei, oder drei Bytes Platz gemacht. Die N-Flag ist immer gesetzt, wenn der Prozessor aus dieser Subroutine zurückkehrt.

GETBYT
Aufruf: JSR-GETBYT oder JSR-1D6F oder 20 6F 1D
Die Subroutine GETBYT liest zwei Daten-Tasten in den Computer und setzt deren Tastenwert im Accu zusammen. Während dieser Subroutine bedient der Computer auch das Display. Die erste gedrückte Taste ist das hochwertige Accu-Nibble. Weitere Besonderheiten von GETBYT:
- Sind die beiden Tastenwerte der Daten-Tasten im Accu zusammengesetzt, kehrt der Computer aus GETBYT mit gesetzter N-Flag (N=1) zurück.
- Wurde eine der fünf Kommando-Tasten gedrückt, kehrt der Computer mit zurückgesetzter N-Flag (N=Ø) aus dieser Subroutine zurück.

GETKEY
Aufruf: JSR-GETKEY oder JSR-1DF9 oder 20 F9 1D
Die Subroutine GETKEY berechnet den Tastenwert einer gedrückten Taste. Nach der Rückkehr aus dieser Subroutine steht der berechnete Tastenwert im Accu der CPU.

LENACC
Aufruf: JSR-LENACC oder JSR-1E60 oder 20 60 1E
Die Subroutine LENACC berechnet die Instruktionslänge einer beliebigen Instruktion der CPU 6502. Dabei muß der OP-Code der Instruktion vor dem Aufruf dieser Subroutine im Accu stehen. Die Instruktionslänge wird in der Speicherzelle BYTES ØØF6 abgespeichert.

NEXT
Aufruf: JSR-NEXT oder JSR-1EF8 oder 20 F8 1E
Die Subroutine NEXT schiebt den Display-Pointer CURAD um ein, zwei oder drei Speicherplätze nach unten. Der Inhalt der Speicherzelle BYTES ØØF6 entscheidet, um wieviele Adressen CURAD nach unten bewegt wird.

OPLEN
Aufruf: JSR-OPLEN oder JSR-1E5C oder 20 5C 1E
Die Subroutine OPLEN berechnet die Instruktionslänge einer beliebigen Instruktion der CPU 6502, auf deren OP-Code der Display-Pointer CURAD zeigt. Die berechnete Instruktionslänge wird in der Speicherzelle BYTES ØØF6 abgelegt.

RDINST
Aufruf: JSR-RDINST oder JSR-1E20 oder 20 20 1E
Die Subroutine RDINST liest vom Tastenfeld eine CPU-Instruktion und transportiert diese in die Displaybuffer.
- POINTH enthält den OP-Code
- POINTL enthält den ersten Operanden (falls vorhanden)
- INH enthält den zweiten Operanden (falls vorhanden)

Während der Programmierer über das Tastenfeld eine Instruktion eingibt, steuert RDINST auch das Display. Ist die gesamte Instruktion eingegeben, kehrt der Prozessor mit gesetzter N-Flag (N=1) aus dieser Subroutine zurück. Wird aber während RDINST eine Kommando-Taste gedrückt, unterbricht der Computer das Einlesen der Instruktion und kehrt mit zurückgesetzter N-Flag (N=Ø) aus dieser Subroutine zurück.

RECEND
Aufruf: JSR-RECEND oder JSR-1EEA oder 2Ø EA 1E
Die Subroutine RECEND schiebt den Pointer CEND um ein, zwei oder drei Speicherplätze nach oben. Der Inhalt der Speicherzelle BYTES ØØF6 entscheidet, um wieviele Adressen CEND nach oben bewegt wird.

SCAN
Aufruf: JSR-SCAN oder JSR-1D4D oder 2Ø 4D 1D
Die Subroutine SCAN füllt die Displaybuffer POINTH, POINTL und INH mit einer Instruktion der CPU 6502 und zeigt diese anschließend auf dem Display an. Auch das Tastenfeld des Junior-Computers wird abgefragt. SCAN führt folgende Aufgaben aus:
- SCAN befördert zum Displaybuffer die Instruktion, auf deren OP-Code der Display-Pointer CURAD zeigt.
- SCAN berechnet automatisch die Instruktionslänge der auf dem Display angezeigten Instruktion.
- SCAN arbeitet mit variabler Display-Länge.
- Entdeckt SCAN eine gedrückte Taste im Tastenfeld, kehrt der Prozessor mit dem Tastenwert im Accu aus dieser Subroutine zurück.

SCAND
Aufruf: JSR-SCAND oder JSR-1D88 oder 2Ø 88 1D
SCAND ist eine Subroutine, die in den Displaybuffer INH die Daten transportiert, auf die der Adreß-Pointer POINT zeigt. Ansonsten ist sie mit SCANDS identisch.

SCANDS
Aufruf: JSR-SCANDS oder JSR-1D8E oder 2Ø 8E 1D
Die Subroutine SCANDS zeigt den Inhalt der drei Displaybuffer POINTH, POINTL und INH auf dem Display an. Dabei kann die Display-Länge variabel sein. Durch Laden von Ø1, Ø2 oder Ø3 in die Speicherzelle BYTES ØØF6 ergibt sich folgende Situation:
- BYTES enthält Ø1: nur den Inhalt von POINTH wird auf dem Display dargestellt.
- BYTES enthält Ø2: nur den Inhalt von POINTH und POINTL wird auf dem Display dargestellt.
- BYTES enthält Ø3: Der Inhalt von POINTH, POINTL und INH wird auf dem Display dargestellt.

SHOW
Aufruf: JSR-SHOW oder JSR-1DCC oder 2Ø CC 1D
Die Subroutine SHOW transportiert den Inhalt irgendeines Displaybuffers (POINTH, POINTL oder INH) zum Display. Dabei wird das X-Register

als Display-Zähler verwendet. Der Inhalt des X-Registers bestimmt, welches der sechs Displays angesteuert wird. SHOW kann nur hexadezimale Zahlen auf dem Display darstellen.

UP

Aufruf: JSR-UP oder JSR-1E83 oder 2Ø 83 1E

Die Subroutine UP bewegt einen Datenblock zwischen den Pointern CURAD und CEND um ein, zwei oder drei Speicherplätze nach oben. Der Inhalt der Speicherzelle BYTES ØØF6 entscheidet, um wieviele Speicherplätze der Datenblock nach oben bewegt wird.

Anhang 2

Source-Listing:

* Editor
* Assembler
* Monitor
* Branch-Routine

```
0000: 1C00            LOYS   ORG   $1C00  VERSION D
0010:
0020:
0030:
0040:
0050:                 SOURCE LISTING OF ELEKTOR'S JUNIOR COMPUTER
0060:
0070:                 WRITTEN BY A. NACHTMANN
0080:
0090:                 DATE: 7 FEB. 1980
0100:
0110:                 THE FEATURES OF JUNIOR'S MONITOR ARE:
0120:
0130:                 HEX ADDRESS DATA DISPLAY (ENTRY VIA RST)
0140:                 HEX EDITOR (START ADDRESS $1CB5)
0150:                 HEX ASSEMBLER (START ADDRESS $1F51)
0160:
0170:                 EDITOR'S POINTERS AND TEMPS IN PAGE ZERO
0180:
0190: 1C00            KEY    *      $00E1
0200: 1C00            BEGADL *      $00E2  BEGIN ADDRESS POINTER
0210: 1C00            BEGADH *      $00E3
0220: 1C00            ENDADL *      $00E4  END ADDRESS POINTER
0230: 1C00            ENDADH *      $00E5
0240: 1C00            CURADL *      $00E6  CURRENT ADDRESS POINTER
0250: 1C00            CURADH *      $00E7
0260: 1C00            CENDL  *      $00E8  CURRENT ADDRESS POINTER
0270: 1C00            CENDH  *      $00E9
0280: 1C00            MOVADL *      $00EA
0290: 1C00            MOVADH *      $00EB
0300: 1C00            TABLEL *      $00EC
0310: 1C00            TABLEH *      $00ED
0320: 1C00            LABELS *      $00EE
0330: 1C00            BYTES  *      $00F6  NUMBER OF BYTES TO BE DISPLAYED
0340: 1C00            COUNT  *      $00F7
0350:
0360:                 MPU REGISTERS IN PAGE ZERO
0370:
0380: 1C00            PCL    *      $00EF
0390: 1C00            PCH    *      $00F0
0400: 1C00            PREG   *      $00F1
0410: 1C00            SPUSER *      $00F2
0420: 1C00            ACC    *      $00F3
0430: 1C00            YREG   *      $00F4
0440: 1C00            XREG   *      $00F5
0450
0460:                 HEX DISPLAY BUFFERS IN PAGE ZERO
0470:
0480: 1C00            INL    *      $00F8
0490: 1C00            INH    *      $00F9
0500: 1C00            POINTL *      $00FA
0510: 1C00            POINTH *      $00FB
0520:
0530:                 TEMPORARY DATA BUFFERS IN PAGE ZERO
0540:
0550: 1C00            TEMP   *      $00FC
0560: 1C00            TEMPX  *      $00FD
0570: 1C00            NIBBLE *      $00FE
0580: 1C00            MODE   *      $00FF  (0 = DA MODE, ≠0 = AD MODE)
0590:
0600:                 MEMORY LOCATIONS IN THE 6532-IC
0610:
0620: 1C00            PAD    *      $1A80  DATA REGISTER OF PORT A
0630: 1C00            PADD   *      $1A81  DATA DIRECTION REGISTER OF PORT A
0640: 1C00            PBD    *      $1A82  DATA REGISTER OF PORT B
0650: 1C00            PBDD   *      $1A83  DATA DIRECTION REGISTER OF PORT B
0660:
0670:                 WRITE EDGE DETECT CONTROL
0680:
0690: 1C00            EDETA  *      $1AE4  NEG EDET DISABLE PA7-IRQ
0700: 1C00            EDETB  *      $1AE5  POS EDET DISABLE PA7-IRQ
0710: 1C00            WDETC  *      $1AE6  NEG EDET ENABLE PA7-IRQ
0720: 1C00            EDETD  *      $1AE7  POS EDET ENABLE PA7-IRQ
0730:
0740:                 READ FLAG REGISTER AND CLEAR TIMER & IRQ FLAG
0750:
0760: 1C00            RDFLAG *      $1AD5  BIT6=PA7-FLAG; BIT7=TIMER-FLAG
0770:
0780:                 WRITE COUNT INTO TIMER, DISABLE TIMER-IRQ
0790:
0800: 1C00            CNTA   *      $1AF4  CLK1T
0810: 1C00            CNTB   *      $1AF5  CLK8T
0820: 1C00            CNTC   *      $1AF6  CLK64T
0830: 1C00            CNTD   *      $1AF7  CLK1KT
0840:
0850:                 WRITE COUNT INTO TIMER, ENABLE TIMER-IRQ
0860:
0870: 1C00            CNTE   *      $1AFC  CLK1T
0880: 1C00            CNTF   *      $1AFD  CLK8T
0890: 1C00            CNTG   *      $1AFE  CLK64T
0900: 1C00            CNTH   *      $1AFF  CLK1KT
0910:
0920:                 INTERRUPT VECTORS: IRQ & NMI VECTORS SHOULD BE
0930:                 LOADED IN THE FOLLOWING MEMORY LOCATIONS FOR
0940:                 PROPER SYSTEM OPERATION.
```

```
0950:
0960: 1C00            NMIL    *      $1A7A  NMI LOWER BYTE
0970: 1C00            NMIH    *      $1A7B  NMI HIGHER BYTE
0980: 1C00            IRQL    *      $1A7E  IRQ LOWER BYTE
0990: 1C00            IRQH    *      $1A7F  IRQ HIGHER BYTE
1000:
1010:                 BEGINNERS MAY LOAD INTO THESE LOCATIONS
1020:                 $1C00 FOR STEP BY STEP MODUS AND BRK COMMAND
1030:
1040:
1050:
1060:                 JUNIOR'S MAINROUTINES
1070:
1080: 1C00 85 F3      SAVE    STAZ   ACC     SAVE ACCU
1090: 1C02 68                 PLA             GET CURRENT P-REGISTER
1100: 1C03 85 F1              STAZ   PREG    SAVE P-REGISTER
1110: 1C05 68         SAVEA   PLA            GET CURRENT PCL
1120: 1C06 85 EF              STAZ   PCL     SAVE CURRENT PCL
1130: 1C08 85 FA              STAZ   POINTL PCL TO DISPLAY BUFFER
1140: 1C0A 68                 PLA             GET CURRENT PCH
1150: 1C0B 85 F0              STAZ   PCH     SAVE CURRENT PCH
1160: 1C0D 85 FB              STAZ   POINTH PCH TO DISPLAY BUFFER
1170: 1C0F 84 F4      SAVEB   STYZ   YREG    SAVE CURRENT Y-REGISTER
1180: 1C11 86 F5              STXZ   XREG    SAVE CURRENT X-REGISTER
1190: 1C13 BA                 TSX             GET CURRENT SP
1200: 1C14 86 F2              STX    SPUSER SAVE CURRENT SP
1210: 1C16 A2 01              LDXIM $01      SET AD-MODE
1220: 1C18 86 FF              STXZ   MODE
1230: 1C1A 4C 33 1C           JMP    START
1240:
1250: 1C1D A9 1E      RESET   LDAIM $1E     PB1---PB4
1260: 1C1F 8D 83 1A           STA    PBDD     IS OUTPUT
1270: 1C22 A9 04              LDAIM $04      RESET P-REGISTER
1280: 1C24 85 F1              STAZ   PREG
1290: 1C26 A9 03              LDAIM $03
1300: 1C28 85 FF              STAZ   MODE    SET AD-MODE
1310: 1C2A 85 F6              STAZ   BYTES  DISPLAY POINTH,POINTL,INH
1320: 1C2C A2 FF              LDXIM $FF     ADJUST THE STACKPOINTER
1330: 1C2E 9A                 TXS
1340: 1C2F 86 F2              STXZ   SPUSER
1350: 1C31 D8                 CLD
1360: 1C32 78                 SEI
1370:
1380: 1C33 20 88 1D   START   JSR    SCAND  DISPLAY DATA SPECIFIED BY POINTH,POINTL
1390: 1C36 D0 FB              BNE    START  WAIT UNTIL KEY IS RELEASED
1400: 1C38 20 88 1D   STARA   JSR    SCAND  DISPLAY DATA SPECIFIED BY POINT
1410: 1C3B F0 FB              BEQ    STARA   ANY KEY DEPRESSED
1420: 1C3D 20 88 1D           JSR    SCAND   DEBOUNCE KEY
1430: 1C40 F0 F6              BEQ    STARA  ANY KEY STILL DEPRESSED
1440: 1C42 20 F9 1D           JSR    GETKEY IF YES , DECODE KEY,RETURN WITH KEY IN ACCU
1450:
1460: 1C45 C9 13      GOEXEC CMPIM $13     GO-KEY?
1470: 1C47 D0 13              BNE    ADMODE
1480: 1C49 A6 F2              LDXZ   SPUSER  GET CURRENT SP
1490: 1C4B 9A                 TXS
1500: 1C4C A5 FB              LDAZ   POINTH  START EXECUTION AT POINTH,POINTL
1510: 1C4E 48                 PHA
1520: 1C4F A5 FA              LDAZ   POINTL
1530: 1C51 48                 PHA
1540: 1C52 A5 F1              LDAZ   PREG   RESTORE CURRENT P REGISTER
1550: 1C54 48                 PHA
1560: 1C55 A6 F5              LDXZ   XREG
1570: 1C57 A4 F4              LDYZ   YREG
1580: 1C59 A5 F3              LDAZ   ACC
1590: 1C5B 40                 RTI             EXECUTE PROGRAM
1600: 1C5C C9 10      ADMODE CMPIM $10     AD-KEY?
1610: 1C5E D0 06              BNE    DAMODE
1620: 1C60 A9 03              LDAIM $03      SET AD-MODE
1630: 1C62 85 FF              STAZ   MODE
1640: 1C64 D0 14              BNE    STEPA
1650:
1660: 1C66 C9 11      DAMODE CMPIM $11     DA-KEY?
1670: 1C68 D0 06              BNE    STEP
1680: 1C6A A9 00              LDAIM $00      SET DA-MODE
1690: 1C6C 85 FF              STAZ   MODE
1700: 1C6E F0 0A              BEQ    STEPA
1710:
1720: 1C70 C9 12      STEP    CMPIM $12     PLUS-KEY?
1730: 1C72 D0 09              BNE    PCKEY
1740: 1C74 E6 FA              INCZ   POINTL
1750: 1C76 D0 02              BNE    STEPA
1760: 1C78 E6 FB              INCZ   POINTH
1770: 1C7A 4C 33 1C   STEPA   JMP    START
1780:
1790: 1C7D C9 14      PCKEY   CMPIM $14     PC-KEY?
1800: 1C7F D0 0B              BNE    ILLKEY
1810: 1C81 A5 EF              LDAZ   PCL
1820: 1C83 85 FA              STAZ   POINTL  LAST PC TO DISPLAY BUFFER
1830: 1C85 A5 F0              LDAZ   PCH
1840: 1C87 85 FB              STAZ   POINTH
1850: 1C89 4C 7A 1C           JMP    STEPA
1860:
1870: 1C8C C9 15      ILLKEY CMPIM $15     ILLEGAL KEY?
1880: 1C8E 10 EA              BPL    STEPA   IF YES,IGNORE IT
1890:
```

```
1900: 1C90 85 E1     DATA   STAZ  KEY      SAVE KEY
1910: 1C92 A4 FF            LDYZ  MODE     Y=0 IS DATA MODE,ELSE ADDRESS MODE
1920: 1C94 D0 0D            BNE   ADDRES
1930: 1C96 B1 FA            LDA1Y POINTL GET DATA SPECIFIED
1940: 1C98 0A               ASLA         BY POINT
1950: 1C99 0A               ASLA         SHIFT LOW ORDER
1960: 1C9A 0A               ASLA         NIBBLE INTO HIGH ORDER NIBBLE
1970: 1C9B 0A               ASLA
1980: 1C9C 05 E1            ORAZ  KEY      DATA WITH KEY
1990: 1C9E 91 FA            STA1Y POINTL RESTORE DATA
2000: 1CA0 4C 7A 1C         JMP   STEPA
2010:
2020: 1CA3 A2 04     ADDRES LDXIM $04      4 SHIFTS
2030: 1CA5 06 FA     ADLOOP ASLZ  POINTL   POINTH,POINTL 4 POSITIONS TO LEFT
2040: 1CA7 26 FB            ROLZ  POINTH
2050: 1CA9 CA               DEX
2060: 1CAA D0 F9            BNE   ADLOOP
2070: 1CAC A5 FA            LDAZ  POINTL
2080: 1CAE 05 E1            ORAZ  KEY      RESTORE ADDRESS
2090: 1CB0 85 FA            STAZ  POINTL
2100: 1CB2 4C 7A 1C         JMP   STEPA
2110:
2120:
2130:
2140:
2150:
2160:                JUNIOR'S HEX EDITOR
2170:
2180:                FOLLOWING COMMANDS ARE VALID:
2190:
2200:                "INSERT": INSERT A NEW LINE JUST BEFORE DISPLAYED LINE
2210:
2220:                "INPUT": INSERT A NEW LINE JUST BEHIND THE
2230:                      DISPLAYED LINE
2240:
2250:                "SEARCH": SEARCH IN WORKSPACE FOR A GIVEN 2BYTE PATTERN
2260:
2270:                "SKIP": SKIP TO NEXT INSTRUCTION
2280:
2290:                "DELETE": DELETE CURRENT DISPLAYED INSTRUCTION
2300:
2310:                AN ERROR IS INDICATED, IF THE INSTRUCTION POINTER
2320:                CURAD IS OUT OF RANGE
2330:
2340: 1CB5 20 D3 1E  EDITOR JSR   BEGIN  CURAD:=BEGAD
2350: 1CB8 A4 E3            LDYZ  BEGADH
2360: 1CBA A6 E2            LDXZ  BEGADL
2370: 1CBC E8               INX
2380: 1CBD D0 01            BNE   EDIT
2390: 1CBF C8               INY
2400: 1CC0 86 E8     EDIT   STXZ  CENDL  CEND:=BEGAD+1
2410: 1CC2 84 E9            STYZ  CENDH
2420: 1CC4 A9 77            LDAIM $77    DISPLAY "77"
2430: 1CC6 A0 00            LDYIM $00
2440: 1CC8 91 E6            STAIY CURADL
2450:
2460: 1CCA 20 4D 1D  CMND   JSR   SCAN   DISPLAY CURRENT INSTRUCTION,WAIT FOR A KEY
2470:
2480: 1CCD C9 14     SEARCH CMPIM $14     SEARCH COMMAND?
2490: 1CCF D0 2A            BNE   INSERT
2500: 1CD1 20 6F 1D         JSR   GETBYT READ 1ST BYTE
2510: 1CD4 10 F7            BPL   SEARCH COM. KEY?
2520: 1CD6 85 FB            STAZ  POINTH DISCARD DATA
2530: 1CD8 20 6F 1D         JSR   GETBYT READ 2ND BYTE
2540: 1CDB 10 F0            BPL   SEARCH COM. KEY?
2550: 1CDD 85 FA            STAZ  POINTL DISCARD DATA
2560: 1CDF 20 D3 1E         JSR   BEGIN  CURAD:=BEGAD
2570: 1CE2 A0 00     SELOOP LDYIM $00
2580: 1CE4 B1 E6            LDAIY CURADL COMPARE INSTRUCTION
2590: 1CE6 C5 FB            CMPZ  POINTH AGAINST DATA TO BE SEARCHED
2600: 1CE8 D0 07            BNE   SEARA  SKIP TO NEXT INSTRUCTION, IF NOT EQUAL
2610: 1CEA C8               INY
2620: 1CEB B1 E6            LDAIY CURADL
2630: 1CED C5 FA            CMPZ  POINTL
2640: 1CEF F0 D9            BEQ   CMND   RETURN, IF 2BYTE PATTERN IS FOUND
2650: 1CF1 20 5C 1E  SEARA  JSR   OPLEN  GET LENGTH OF THE CURRENT INSTRUCTION
2660: 1CF4 20 F8 1E         JSR   NEXT   SKIP TO NEXT INSTRUCTION
2670: 1CF7 30 E9            BMI   SELOOP SEARCH AGAIN, IF CURAD IS LESS THAN CEND
2680: 1CF9 10 3E            BPL   ERRA
2690:
2700: 1CFB C9 10     INSERT CMPIM $10     INSERT COMMAND?
2710: 1CFD D0 0A            BNE   INPUT
2720: 1CFF 20 20 1E         JSR   RDINST READ INSTRUCTION AND COMPUTE LENGTH
2730: 1D02 10 C9            BPL   SEARCH COM. KEY?
2740: 1D04 20 47 1E         JSR   FILLWS MOVE DATA IN WS DOWNWARD BY THE AM. IN BYTES
2750: 1D07 F0 C1            BEQ   CMND   RETURN TO DISPLAY THE INSERTED INSTR.
2760:
2770: 1D09 C9 13     INPUT  CMPIM $13     INPUT COMMAND?
2780: 1D0B D0 14            BNE   SKIP
2790: 1D0D 20 20 1E         JSR   RDINST READ INSTRUCTION AND COMPUTE LENGTH
2800: 1D10 10 BB            BPL   SEARCH COM. KEY?
2810: 1D12 20 5C 1E         JSR   OPLEN  LENGTH OF THE CURRENT INSTR.
2820: 1D15 20 F8 1E         JSR   NEXT   RETURN WITH N=1, IF CURAD IS LESS THAN CEND
2830: 1D18 A5 FD            LDAZ  TEMPX  LENGTH OF INSTR. TO BE INSERTED
2840: 1D1A 85 F6            STAZ  BYTES
```

```
1D1C 20 47 1E        JSR   FILLWS MOVE DATA IN WS DOWNWARD BY THE AM. IN BYTES
1D1F F0 A9           BEQ   CMND   RETURN TO DISPLAY THE INSERTED DATA

1D21 C9 12    SKIP   CMPIM $12    SKIP COMMAND?
1D23 D0 07           BNE   DELETE
1D25 20 F8 1E        JSR   NEXT   SKIP TO NEXT INSTRUCTION. CURAD LESS THAN CEND?
1D28 30 A0           BMI   CMND
1D2A 10 0D           BPL   ERRA

1D2C C9 11    DELETE CMPIM $11    DELETE COMMAND?
1D2E D0 09           BNE   ERRA
1D30 20 83 1E        JSR   UP     DELETE CURRENT INSTR. BY MOVING UP THE WS
1D33 20 EA 1E        JSR   RECEND ADJUST CURRENT END ADDRESS
1D36 4C CA 1C        JMP   CMND

1D39 A9 EE    ERRA   LDAIM $EE
1D3B 85 FB           STAZ  POINTH
1D3D 85 FA           STAZ  POINTL
1D3F 85 F9           STAZ  INH
1D41 A9 03           LDAIM $03
1D43 85 F6           STAZ  BYTES
1D45 20 8E 1D ERRB   JSR   SCANDS DISPLAY EEEEEE UNTIL KEY IS RELEASED
1D48 D0 FB           BNE   ERRB
1D4A 4C CA 1C        JMP   CMND


                     EDITOR'S SUBROUTINES

                     SCAN IS A SUBROUTINE, FILLING UP
                     THE DISPLAY BUFFER DETERMINED BY
                     CURAD. THEN THE DISPLAY IS SCANNED
                     DEPENDING OF THE LENGTH OF THE INSTRUCTION
                     POINTED BY CURAD
                     IF A DEPRESSED KEY IS DETECTED



                     SCAN RETURNS WITH VALUE IN ACCU

1D4D A2 02    SCAN   LDXIM $02    FILL UP THE DISPLAY BUFFER
1D4F A0 00           LDYIM $00
1D51 B1 E6    FILBUF LDAIY CURADL START FILLING AT OP CODE
1D53 95 F9           STAX  INH
1D55 C8              INY
1D56 CA              DEX
1D57 10 F8           BPL   FILBUF
1D59 20 5C 1E        JSR   OPLEN  STORE INSTRUCTION LENGTH IN BYTES
1D5C 20 8E 1D SCANA  JSR   SCANDS DISPLAY CURRENT INSTRUCTION
1D5F D0 FB           BNE   SCANA  KEY RELEASED?
1D61 20 8E 1D SCANB  JSR   SCANDS DISPLAY CURRENT INSTRUCTION
1D64 F0 FB           BEQ   SCANB  ANY KEY DEPRESSED?
1D66 20 8E 1D        JSR   SCANDS DISPLAY CURRENT INSTRUCTION
1D69 F0 F6           BEQ   SCANB  ANY KEY STILL DEPRESSED?
1D6B 20 F9 1D        JSR   GETKEY IF YES, RETURN WITH KEY IN ACCU
1D6E 60              RTS

                     GETBYT READS 2 HEXKEYS AND COMPOSES
                     THEIR VALUES IN THE A REGISTER. IF ONLY
                     HEXKEYS WERE DEPRESSED, IT RETURNS WITH
                     N=1. IF A COMMAND KEY WAS DEPRESSED, IT
                     RETURNES WITH N=0.

1D6F 20 5C 1D GETBYT JSR   SCANA  READ HIGH ORDER NIBBLE
1D72 C9 10           CMPIM $10
1D74 10 11           BPL   BYTEND COMMAND KEY?
1D76 0A              ASLA
1D77 0A              ASLA         IF NOT, SAVE HIGH ORDER NIBBLE
1D78 0A              ASLA
1D79 0A              ASLA
1D7A 85 FE           STA   NIBBLE
1D7C 20 5C 1D        JSR   SCANA  READ LOW ORDER NIBBLE
1D7F C9 10           CMPIM $10
1D81 10 04           BPL   BYTEND COMMAND KEY?
1D83 05 FE           ORA   NIBBLE IF  NOT, COMPOSE BYTE
1D85 A2 FF           LDXIM $FF    SET N=1
1D87 60       BYTEND RTS

                     SCAND IS A SUBROUTINE SHOWING DATA SPECIFIED BY
                     POINT.
                     SCANDS IS A SUBROUTINE SHOWING THE CONTENTS OF
                     DISPLAY BUFFER AS A FUNCTION OF BYTES.
                     THE FOLLOWING SUBROUTINE AK SCANS THE KEYBOARD.
                     IT RETURNS WITH A=0, IF NO KEY IS DEPRESSED AND
                     WITH A≠0 IF A KEY IS DEPRESSED.
                     WHEN SCAND OR SCANDS ARE LEFT, PA0...PA7 IS INPUT.



1D88 A0 00    SCAND  LDYIM $00
1D8A B1 FA           LDAIY POINTL GET DATA SPECIFIED BY POINT
1D8C 85 F9           STAZ  INH
```

```
3800: 1D8E A9 7F     SCANDS LDAIM $7F
3810: 1D90 8D 81 1A         STA   PADD    PA0...PA6 IS OUTPUT
3820: 1D93 A2 08            LDXIM $08     ENABLE DISPLAY
3830: 1D95 A4 F6            LDYZ  BYTES   FETCH LENGTH FROM BYTES
3840: 1D97 A5 FB     SCDSA  LDAZ  POINTH  OUTPUT 1ST BYTE
3850: 1D99 20 CC 1D         JSR   SHOW
3860: 1D9C 88               DEY
3870: 1D9D F0 0D            BEQ   SCDSB   MORE BYTES?
3880: 1D9F A5 FA            LDAZ  POINTL
3890: 1DA1 20 CC 1D         JSR   SHOW    IF YES, OUTPUT 2ND BYTE
3900: 1DA4 88               DEY
3910: 1DA5 F0 05            BEQ   SCDSB   MORE BYTES?
3920: 1DA7 A5 F9            LDAZ  INH
3930: 1DA9 20 CC 1D         JSR   SHOW    IF YES, OUTPUT 3RD BYTE
3940: 1DAC A9 00     SCDSB  LDAIM $00
3950: 1DAE 8D 81 1A         STA   PADD    PA0...PA7 IS INPUT
3960:
3970: 1DB1 A0 03     AK     LDYIM $03     SCAN 3 ROWS
3980: 1DB3 A2 00            LDXIM $00     RESET ROW COUNTER
3990:
4000: 1DB5 A9 FF     ONEKEY LDAIM $FF
4010: 1DB7 8E 82 1A  AKA    STX   PBD     OUTPUT ROW NUMBER
4020: 1DBA E8               INX           ENABLE FOLLOWING ROW
4030: 1DBB E8               INX
4040: 1DBC 2D 80 1A         AND   PAD     INPUT ROW PATTERN
4050: 1DBF 88               DEY           ALL ROWS SCANNED?
4060: 1DC0 D0 F5            BNE   AKA
4070: 1DC2 A0 06            LDYIM $06     TURN DISPLAY OFF
4080: 1DC4 8C 82 1A         STY   PBD
4090: 1DC7 09 80            ORAIM $80     SET BIT7=1
4100: 1DC9 49 FF            EORIM $FF     INVERT KEY PATTERN
4110: 1DCB 60               RTS
4120:
4130:                  THE SUBROUTINE SHOW TRANSPORTS THE
4140:                  CONTENTS OF ANY DISPLAY BUFFER TO THE
4150:                  DISPLAY. THE X REGISTER IS USED AS A
4160:                  SCAN COUNTER. IT DETERMINES, IF POINTH,
4170:                  POINTL OR INH IS TRANSPORTED TO THE
4180:                  DISPLAY.
4190: 1DCC 48        SHOW   PHA           SAVE DISPLAY
4200: 1DCD 84 FC            STYZ  TEMP    SAVE Y REGISTER
4210: 1DCF 4A               LSRA
4220: 1DD0 4A               LSRA          GET HIGH ORDER NIBBLE
4230: 1DD1 4A               LSRA
4240: 1DD2 4A               LSRA
4250: 1DD3 20 DF 1D         JSR   CONVD   OUTPUT HIGH ORDER NIBBLE
4260: 1DD6 68               PLA           GET DISPLAY AGAIN
4270: 1DD7 29 0F            ANDIM $0F     MASK OFF HIGH ORDER NIBBLE
4280: 1DD9 20 DF 1D         JSR   CONVD   OUTPUT LOW ORDER NIBBLE
4290: 1DDC A4 FC            LDYZ  TEMP    RESTORE Y REGISTER
4300: 1DDE 60               RTS
4310:
4320:
4330:
4340:
4350:                  THE SUBROUTINE CONVD CONTROLS THE DISPLAY SCAN.
4360:                  IT CONVERTS THE CONTENTS OF THE DISPLAY BUFFER
4370:                  TO BE DISPLAYED INTO A SEGMENT PATTERN.
4380:
4390: 1DDF A8        CONVD  TAY           USE NIBBLE AS INDEX
4400: 1DE0 B9 0F 1F         LDAY  LOOK    FETCH SEGMENT PATTERN
4410: 1DE3 8D 80 1A         STA   PAD     OUTPUT SEGMENT PATTERN
4420: 1DE6 8E 82 1A         STX   PBD     OUTPUT DIGIT ENABLE
4430: 1DE9 A0 7F            LDYIM $7F
4440: 1DEB 88        DELAY  DEY           DELAY 500 US APPROX.
4450: 1DEC 10 FD            BPL   DELAY
4460: 1DEE 8C 80 1A         STY   PAD     TURN SEGMENTS OFF
4470: 1DF1 A0 06            LDYIM $06
4480: 1DF3 8C 82 1A         STY   PBD     TURN DISPLAY OFF
4490: 1DF6 E8               INX           ENABLE NEXT DIGIT
4500: 1DF7 E8               INX
4510: 1DF8 60               RTS
4520:
4530:                  GETKEY CONVERTS A DEPRESSED KEY INTO A
4540:                  HEX NUMBER. IT RETURNS WITH THE KEY VALUE
4550:                  IN ACCU.
4560:
4570: 1DF9 A2 21     GETKEY LDXIM $21     START AT ROW 0
4580: 1DFB A0 01     GETKEA LDYIM $01     GET ONE ROW
4590: 1DFD 20 B5 1D         JSR   ONEKEY A=0, NO KEY DEPRESSED
4600: 1E00 D0 07            BNE   KEYIN
4610: 1E02 E0 27            CPXIM $27
4620: 1E04 D0 F5            BNE   GETKEA EACH ROW SCANNED?
4630: 1E06 A9 15            LDAIM $15     RETURN IF INVALID KEY
4640: 1E08 60               RTS
4650: 1E09 A0 FF     KEYIN  LDYIM $FF
4660: 1E0B 0A        KEYINA ASLA          SHIFT LEFT UNTIL Y=KEY NUMBER
4670: 1E0C B0 03            BCS   KEYINB
4680: 1E0E C8               INY
4690: 1E0F 10 FA            BPL   KEYINA
4700: 1E11 8A        KEYINB TXA
4710: 1E12 29 0F            ANDIM $0F     MASK MSD
4720: 1E14 4A               LSRA          DEVIDE BY 2
4730: 1E15 AA               TAX
4740: 1E16 98               TYA
```

```
4750: 1E17 10 03           BPL   KEYIND
4760: 1E19 18        KEYINC CLC
4770: 1E1A 69 07           ADCIM $07    ADD ROW OFFSET
4780: 1E1C CA        KEYIND DEX
4790: 1E1D D0 FA           BNE   KEYINC
4800: 1E1F 60              RTS
4810:
4820:                      RDINST TRANSFERS AN INSTRUCTION FROM KEYBOARD
4830:                      TO THE DISPLAY BUFFER. IT RETURNS WITH N=0 IF
4840:                      A COMMAND KEY WAS DEPRESSED. ONCE THE ENTIRE
4850:                      INSTRUCTION  IS READ, RDINST RETURNS WITH N=1.
4860:
4870:
4880: 1E20 20 6F 1D  RDINST JSR   GETBYT READ OP CODE
4890: 1E23 10 21           BPL   RDB    RETURN, IF COMMAND KEY
4900: 1E25 85 FB           STAZ  POINTH STORE OP CODE IN DISPLAY BUFFER
4910: 1E27 20 60 1E        JSR   LENACC COMPUTE INSTRUCTION LENGTH
4920: 1E2A 84 F7           STYZ  COUNT
4930: 1E2C 84 FD           STYZ  TEMPX
4940: 1E2E C6 F7           DECZ  COUNT
4950: 1E30 F0 12           BEQ   RDA    1 BYTE INSTRUCTION?
4960: 1E32 20 6F 1D        JSR   GETBYT IF NOT, READ FIRST OPERAND
4970: 1E35 10 0F           BPL   RDB    RETURN, IF COMMAND KEY
4980: 1E37 85 FA           STAZ  POINTL STORE 1ST OPERAND IN DISPLAY BUFFER
4990: 1E39 C6 F7           DECZ  COUNT
5000: 1E3B F0 07           BEQ   RDA    2 BYTE INSTRUCTION?
5010: 1E3D 20 6F 1D        JSR   GETBYT IF NOT, READ 2ND OPERAND
5020: 1E40 10 04           BPL   RDB    RETURN IF COMMAND KEY
5030: 1E42 85 F9           STAZ  INH    STORE 2ND OPERAND IN DISPLAY BUFFER
5040: 1E44 A2 FF     RDA   LDXIM $FF    N=1
5050: 1E46 60        RDB   RTS
5060:
5070:                      FILLWS TRANSFERS THE DATA FROM DISPLAY TO
5080:                      WORKSPACE. IT'S ALWAYS LEFT WITH Z=1
5090:
5100: 1E47 20 A6 1E  FILLWS JSR   DOWN   MOVE DATA DOWN BY THE AMOUNT IN BYTES
5110: 1E4A 20 DC 1E        JSR   ADCEND ADJUST CURRENT END ADDRESS
5120: 1E4D A2 02           LDXIM $02
5130: 1E4F A0 00           LDYIM $00
5140: 1E51 B5 F9     WS    LDAZX INH    FETCH DATA FROM DISPLAY BUFFER
5150: 1E53 91 E6           STAIY CURADL INSERT DATA INTO DATA FIELD
5160: 1E55 CA              DEX
5170: 1E56 C8              INY
5180: 1E57 C4 F6           CPYZ  BYTES  ALL INSERTED?
5190: 1E59 D0 F6           BNE   WS     IF NOT, CONTINUE
5200: 1E5B 60              RTS
5210:
5220:                      OPLEN COMPUTES THE LENGTH OF ANY 6502 INSTR.
5230:                      THE INSTR. LENGTH IS SAVED IN BYTES.
5240:
5250: 1E5C A0 00     OPLEN LDYIM $00
5260: 1E5E B1 E6           LDAIY CURADL FETCH OP CODE FROM WS
5270: 1E60 A0 01     LENACC LDYIM $01    LENGTH OF OP CODE IS 1 BYTE
5280: 1E62 C9 00           CMPIM $00
5290: 1E64 F0 1A           BEQ   LENEND BRK INSTRUCTION?
5300: 1E66 C9 40           CMPIM $40
5310: 1E68 F0 16           BEQ   LENEND RTI INSTRUCTION?
5320: 1E6A C9 60           CMPIM $60
5330: 1E6C F0 12           BEQ   LENEND RTS INSTRUCTION?
5340: 1E6E A0 03           LDYIM $03
5350: 1E70 C9 20           CMPIM $20
5360: 1E72 F0 0C           BEQ   LENEND JSR INSTRUCTION?
5370: 1E74 29 1F           ANDIM $1F    STRIP TO 5 BITS
5380: 1E76 C9 19           CMPIM $19
5390: 1E78 F0 06           BEQ   LENEND ANY ABS,Y INSTRUCTION?
5400: 1E7A 29 0F           ANDIM $0F    STRIP TO 4 BITS
5410: 1E7C AA              TAX          USE NIBBLE AS INDEX
5420: 1E7D BC 1F 1F        LDYX  LEN    FETCH LENGTH FROM LEN
5430: 1E80 84 F6     LENEND STYZ  BYTES  DISCARD LENGTH IN BYTES
5440: 1E82 60              RTS
5450:
5460:                      UP MOVES A DATA FIELD BETWEEN CURAD AND CEND
5470:                      UPWARD BY THE AMOUNT IN BYTES
5480:
5490: 1E83 A5 E6     UP    LDAZ  CURADL
5500: 1E85 85 EA           STAZ  MOVADL
5510: 1E87 A5 E7           LDAZ  CURADH MOVAD:=CURAD
5520: 1E89 85 EB           STAZ  MOVADH
5530: 1E8B A4 F6     UPLOOP LDYZ  BYTES
5540: 1E8D B1 EA           LDAIY MOVADL MOVE UPWARD BY THE AMOUNT IN BYTES
5550: 1E8F A0 00           LDYIM $00
5560: 1E91 91 EA           STAIY MOVADL
5570: 1E93 E6 EA           INCZ  MOVADL
5580: 1E95 D0 02           BNE   UPA
5590: 1E97 E6 EB           INCZ  MOVADH MOVADH:=MOVADH+1
5600: 1E99 A5 EA     UPA   LDAZ  MOVADL
5610: 1E9B C5 E8           CMPZ  CENDL
5620: 1E9D D0 EC           BNE   UPLOOP ALL DATA MOVED?
5630: 1E9F A5 EB           LDAZ  MOVADH IF NOT, CONTINUE
5640: 1EA1 C5 E9           CMPZ  CENDH
5650: 1EA3 D0 E6           BNE   UPLOOP
5660: 1EA5 60              RTS
5670:
5680:                      DOWN MOVES A DATA FIELD BETWEEN CURAD
5690:                      AND CEND DOWNWARD BY THE AMOUNT IN BYTES
```

```
5700:
5710: 1EA6 A5 E8     DOWN   LDAZ  CENDL
5720: 1EA8 85 EA            STAZ  MOVADL MOVAD:=CEND
5730: 1EAA A5 E9            LDAZ  CENDH
5740: 1EAC 85 EB            STAZ  MOVADH
5750: 1EAE A0 00     DNLOOP LDYIM $00
5760: 1EB0 B1 EA            LDAIY MOVADL MOVE DOWNWARD BY THE AMOUNT IN BYTES
5770: 1EB2 A4 F6            LDYZ  BYTES
5780: 1EB4 91 EA            STAIY MOVADL
5790: 1EB6 A5 EA            LDAZ  MOVADL
5800: 1EB8 C5 E6            CMPZ  CURADL
5810: 1EBA D0 06            BNE   DNA    ALL DATA MOVED?
5820: 1EBC A5 EB            LDAZ  MOVADH IF NOT, CONTINUE
5830: 1EBE C5 E7            CMPZ  CURADH
5840: 1EC0 F0 10            BEQ   DNEND
5850: 1EC2 38        DNA    SEC
5860: 1EC3 A5 EA            LDAZ  MOVADL
5870: 1EC5 E9 01            SBCIM $01
5880: 1EC7 85 EA            STAZ  MOVADL
5890: 1EC9 A5 EB            LDAZ  MOVADH MOVAD:=MOVAD-1
5900: 1ECB E9 00            SBCIM $00
5910: 1ECD 85 EB            STAZ  MOVADH
5920: 1ECF 4C AE 1E         JMP   DNLOOP
5930: 1ED2 60        DNEND  RTS
5940:
5950:
5960:                BEGIN SETS CURAD EQUAL TO BEGAD
5970:
5980: 1ED3 A5 E2     BEGIN  LDAZ  BEGADL
5990: 1ED5 85 E6            STAZ  CURADL
6000: 1ED7 A5 E3            LDAZ  BEGADH CURAD:=BEGAD
6010: 1ED9 85 E7            STAZ  CURADH
6020: 1EDB 60               RTS
6030:
6040:                ADCEND ADVANCES CURRENT END ADDRESS
6050:                DOWNWARD BY THE AMOUNT IN BYTES
6060:
6070: 1EDC 18        ADCEND CLC
6080: 1EDD A5 E8            LDAZ  CENDL
6090: 1EDF 65 F6            ADCZ  BYTES  CEND:=CEND+BYTES
6100: 1EE1 85 E8            STAZ  CENDL
6110: 1EE3 A5 E9            LDAZ  CENDH
6120: 1EE5 69 00            ADCIM $00
6130: 1EE7 85 E9            STAZ  CENDH
6140: 1EE9 60               RTS
6150:
6160:                RECEND REDUCES THE CURRENT END ADDRESS
6170:                BY THE AMOUNT IN BYTES
6180:
6190: 1EEA 38        RECEND SEC
6200: 1EEB A5 E8            LDAZ  CENDL
6210: 1EED E5 F6            SBCZ  BYTES  CEND:=CEND-BYTES
6220: 1EEF 85 E8            STAZ  CENDL
6230: 1EF1 A5 E9            LDAZ  CENDH
6240: 1EF3 E9 00            SBCIM $00
6250: 1EF5 85 E9            STAZ  CENDH
6260: 1EF7 60               RTS
6270:
6280:                NEXT ADVANCES THE CURRENT DISPLAYED ADDRES
6290:                DOWNWARD BY THE AMOUNT IN BYTES
6300:
6310: 1EF8 18        NEXT   CLC
6320: 1EF9 A5 E6            LDAZ  CURADL
6330: 1EFB 65 F6            ADCZ  BYTES  CURAD:=CURAD+BYTES
6340: 1EFD 85 E6            STAZ  CURADL
6350: 1EFF A5 E7            LDAZ  CURADH
6360: 1F01 69 00            ADCIM $00
6370: 1F03 85 E7            STAZ  CURADH
6380: 1F05 38               SEC
6390: 1F06 A5 E6            LDAZ  CURADL
6400: 1F08 E5 E8            SBCZ  CENDL
6410: 1F0A A5 E7            LDAZ  CURADH
6420: 1F0C E5 E9            SBCZ  CENDH
6430: 1F0E 60               RTS
6440:
6450:                THE LOOKUP TABLE "LOOK" IS USED, TO CONVERT
6460:                A HEX NUMBER INTO A 7 SEGMENT PATTERN.
6470:                THE LOOKUP TABLE "LEN" IS USED, TO CONVERT AN
6480:                INSTRUCTION INTO AN INSTRUCTION LENGTH.
6490:
6500:
6510: 1F0F 40        LOOK   =     $40          "0"
6520: 1F10 79               =     $79          "1"
6530: 1F11 24               =     $24          "2"
6540: 1F12 30               =     $30          "3"
6550: 1F13 19               =     $19          "4"
6560: 1F14 12               =     $12          "5"
6570: 1F15 02               =     $02          "6"
6580: 1F16 78               =     $78          "7"
6590: 1F17 00               =     $00          "8"
6600: 1F18 10               =     $10          "9"
6610: 1F19 08               =     $08          "A"
6620: 1F1A 03               =     $03          "B"
6630: 1F1B 46               =     $46          "C"
6640: 1F1C 21               =     $21          "D"
```

```
6650: 1F1D 06               =     $06         "E"
6660: 1F1E 0E               =     $0E         "F"
6670:
6680: 1F1F 02        LEN    =     $02
6690: 1F20 02               =     $02
6700: 1F21 02               =     $02
6710: 1F22 01               =     $01
6720: 1F23 02               =     $02
6730: 1F24 02               =     $02
6740: 1F25 02               =     $02
6750: 1F26 01               =     $01
6760: 1F27 01               =     $01
6770: 1F28 02               =     $02
6780: 1F29 01               =     $01
6790: 1F2A 01               =     $01
6800: 1F2B 03               =     $03
6810: 1F2C 03               =     $03
6820: 1F2D 03               =     $03
6830: 1F2E 03               =     $03
6840:
6850: 1F2F 6C 7A 1A         JMI   NMIL   JUMP TO A  USER SELECTABLE NMI VECTOR
6860: 1F32 6C 7E 1A         JMI   IRQL   JUMP TO A  USER SELECTABLE IRQ VECTOR
6870:
6880:                 GETLBL IS AN ASSEMBLER SUBROUTINE. IT SEARCHES FOR
6890:                 LABELS ON THE SYMBOL PSEUDO STACK. IF THIS STACK
6900:                 CONTAINS A VALID LABEL, IT RETURNS WITH THE
6910:                 HIGH ORDER LABEL ADDRESS IN X AND THE LOW ORDER LABEL
6920:                 ADDRESS IN A. IF NO VALID LABEL IS FOUND, IT RE-
6930:                 TURNES WITH Z=1.
6940:
6950: 1F35 B1 E6      GETLBL LDAIY CURADL FETCH CURRENT LABEL NUMBER FROM WS
6960: 1F37 A0 FF             LDYIM $FF    RESET PSEUDO STACK
6970: 1F39 C4 EE      SYMA   CPYZ  LABELS UPPER MOST SYMBOL TABLE ADDRESS?
6980: 1F3B F0 0D             BEQ   SYMB   IF YES, RETURN, NO LABEL ON PSEUDO STACK
6990: 1F3D D1 EC             CMPIY TABLEL LABEL NR. IN WS = LABEL NR. ON PSEUDO STACK?
7000: 1F3F D0 0A             BNE   SYMNXT
7010: 1F41 88                DEY          IF YES, GET HIGH ORDER ADD
7020: 1F42 B1 EC             LDAIY TABLEL
7030: 1F44 AA                TAX          DISCARD HIGH ORDER ADD IN X
7040: 1F45 88                DEY
7050: 1F46 B1 EC             LDAIY TABLEL GET LOW ORDER ADD
7060: 1F48 A0 01             LDYIM $01     PREPARE Y REGISTER
7070: 1F4A 60         SYMB   RTS
7080:
7090: 1F4B 88         SYMNXT DEY                  *********   *********
7100: 1F4C 88                DEY                  * X=ADH *   * A=ADL *
7110: 1F4D 88                DEY                  *********   *********
7120: 1F4E D0 E9             BNE   SYMA
7130: 1F50 60                RTS
7140:
7150:
7160:
7170:
7180:
7190:
7200:
7210:                 ASSEMBLER MAIN ROUTINE
7220:
7230:                 FOLLOWING INSTRUCTIONS ARE ASSEMBLED:
7240:
7250:                 JSR INSTRUCTION
7260:                 JMP INSTRUCTION
7270:                 BRANCH INSTRUCTIONS
7280:
7290:
7300:
7310:
7320:
7330:
7340: 1F51 38         ASSEMB SEC
7350: 1F52 A5 E4             LDAZ  ENDADL
7360: 1F54 E9 FF             SBCIM $FF
7370: 1F56 85 EC             STAZ  TABLEL TABLE:=ENDAD-$FF
7380: 1F58 A5 E5             LDAZ  ENDADH
7390: 1F5A E9 00             SBCIM $00
7400: 1F5C 85 ED             STAZ  TABLEH
7410: 1F5E A9 FF             LDAIM $FF
7420: 1F60 85 EE             STAZ  LABELS
7430: 1F62 20 D3 1E          JSR   BEGIN  CURAD:=BEGAD
7440:
7450: 1F65 20 5C 1E   PASSA  JSR   OPLEN  START PASS ONE, GET CURR. INSTR.
7460: 1F68 A0 00             LDYIM $00
7470: 1F6A B1 E6             LDAIY CURADL FETCH CURRENT INSTRUCTION
7480: 1F6C C9 FF             CMPIM $FF    IS THE CURRENT INSTR. A LABEL?
7490: 1F6E D0 1D             BNE   NXTINS
7500: 1F70 C8                INY
7510: 1F71 B1 E6             LDAIY CURADL IF YES, FETCH LABEL NR.
7520: 1F73 A4 EE             LDYZ  LABELS
7530: 1F75 91 EC             STAIY TABLEL DEPOSIT LABEL NR. ON SYMBOL STACK
7540: 1F77 88                DEY
7550: 1F78 A5 E7             LDAZ  CURADH GET HIGH ORDER ADD
7560: 1F7A 91 EC             STAIY TABLEL DEPOSIT ON SYMBOL STACK
7570: 1F7C 88                DEY
7580: 1F7D A5 E6             LDAZ  CURADL GET LOW ORDER ADD
7590: 1F7F 91 EC             STAIY TABLEL DEPOSIT ON SYMBOL STACK
```

```
7600: 1F81 88                 DEY
7610: 1F82 84 EE              STYZ  LABELS ADJUST PSEUDO STACK POINTER
7620: 1F84 20 83 1E           JSR   UP     DELETE CURRENT LABEL IN WS
7630: 1F87 20 EA 1E           JSR   RECEND ADJUST CURRENT END ADD
7640: 1F8A 4C 65 1F           JMP   PASSA  LOOK FOR MORE LABELS
7650:
7660: 1F8D 20 F8 1E  NXTINS JSR   NEXT   IF NO LABEL, SKIP TO NEXT INSTR.
7670: 1F90 30 D3              BMI   PASSA  ALL LABELS IN WS COLLECTED?
7680: 1F92 20 D3 1E           JSR   BEGIN  START PASS 2
7690: 1F95 20 5C 1E  PASSB   JSR   OPLEN  GET LENGTH OF THE CURRENT INSTR.
7700: 1F98 A0 00              LDYIM $00
7710: 1F9A B1 E6              LDAIY CURADL FETCH CURRENT INSTR.
7720: 1F9C C9 4C              CMPIM $4C    JMP INSTR.?
7730: 1F9E F0 16              BEQ   JUMPS
7740: 1FA0 C9 20              CMPIM $20    JSR INSTR.?
7750: 1FA2 F0 12              BEQ   JUMPS
7760: 1FA4 29 1F              ANDIM $1F    STRIP TO 5 BITS
7770: 1FA6 C9 10              CMPIM $10    ANY BRANCH INSTRUCTION?
7780: 1FA8 F0 1A              BEQ   BRINST
7790: 1FAA 20 F8 1E  PB      JSR   NEXT   IF NOT, RETURN
7800: 1FAD 30 E6              BMI   PASSB  ALL LABELS BETWEEN CURAD AND ENDAD ASSEMBLED?
7810: 1FAF A9 03              LDAIM $03    ENABLE 3 DISPLAY BUFFERS
7820: 1FB1 85 F6              STAZ  BYTES
7830: 1FB3 4C 33 1C           JMP   START  EXIT HERE  ********
7840:
7850:
7860: 1FB6 C8         JUMPS  INY          SET POINTER TO LABEL NR.
7870: 1FB7 20 35 1F           JSR   GETLBL GET LABEL ADD
7880: 1FBA F0 EE              BEQ   PB     RETURN, IF NOT FOUND
7890: 1FBC 91 E6              STAIY CURADL STORE LOW ORDER ADD
7900: 1FBE 8A                 TXA
7910: 1FBF C8                 INY
7920: 1FC0 91 E6              STAIY CURADL STORE HIGH ORDER ADD
7930: 1FC2 D0 E6              BNE   PB
7940:
7950: 1FC4 C8         BRINST INY          SET POINTER TO LABEL NR.
7960: 1FC5 20 35 1F           JSR   GETLBL GET LABEL ADD.
7970: 1FC8 F0 E0              BEQ   PB     RETURN, IF LABEL NOT FOUND
7980: 1FCA 38                 SEC
7990: 1FCB E5 E6              SBCZ  CURADL COMPUTE BRANCH OFFSET
8000: 1FCD 38                 SEC
8010: 1FCE E9 02              SBCIM $02    DESTINATION-SOURCE-2=OFFSET
8020: 1FD0 91 E6              STAIY CURADL INSERT BRANCH OFFSET IN WS
8030: 1FD2 4C AA 1F           JMP   PB
8040:
8050:
8060
8070
8080
8090:                  THE SUBROUTINE BRANCH COMPUTES THE OFFSET OF BRANCH
8100:                  INSTRUCTIONS. THE 2 RIGHT HAND DISPLAYS SHOW THE
8110:                  COMPUTED OFFSET DEFINED BY THE 4 LEFT HAND DISPLAYS.
8120:                  THE PROGRAM MUST BE STOPPED WITH THE RESET KEY.
8130:
8140: 1FD5 D8          BRANCH CLD
8150: 1FD6 A9 00              LDAIM $00    RESET DISPLAY BUFFER
8160: 1FD8 85 FB              STAZ  POINTH
8170: 1FDA 85 FA              STAZ  POINTL
8180: 1FDC 85 F9              STAZ  INH
8190: 1FDE 20 6F 1D  BR      JSR   GETBYT READ SOURCE
8200: 1FE1 10 F2              BPL   BRANCH COMMAND KEY?
8210: 1FE3 85 FB              STAZ  POINTH SAVE SOURCE IN BUFFER
8220: 1FE5 20 6F 1D           JSR   GETBYT READ DESTINATION
8230: 1FE8 10 EB              BPL   BRANCH COMMAND KEY
8240: 1FEA 85 FA              STAZ  POINTL SAVE DESTINATION IN BUFFER
8250: 1FEC 18                 CLC
8260: 1FED A5 FA              LDAZ  POINTL FETCH DESTINATION
8270: 1FEF E5 FB              SBCZ  POINTH SUBTRACT SOURCE
8280: 1FF1 85 F9              STAZ  INH
8290: 1FF3 C6 F9              DECZ  INH    EQUALIZE AND SAVE OFFSET IN BUFFER
8300: 1FF5 4C DE 1F           JMP   BR
8310:
8320:                  VECTORS AT THE END OF THE MEMORY:
8330:
8340:                  1FFA $2F   NMI VECTOR
8350:                  1FFB $1F
8360:                  1FFC $1D   RESET VECTOR
8370:                  1FFD $1C
8380:                  1FFE $32   IRQ OR BRK VECTOR
8390:                  1FFF $1F
8400:
8410:                  END OF JUNIOR'S MONITOR
8420:
8430:
8440:
8450:
8460:
8470:
8480:
8490:
8500:
8510:
8520:
8530:
```

```
8540:  ACC     00F3    ADCEND 1EDC    ADDRES 1CA3    ADLOOP 1CA5
8550:  ADMODE  1C5C    AK     1DB1    AKA    1DB7    ASSEMB 1F51
8560:  BEGADH  00E3    BEGADL 00E2    BEGIN  1ED3    BR     1FDE
8570:  BRANCH  1FD5    BRINST 1FC4    BYTEND 1D87    BYTES  00F6
8580:  CENDH   00E9    CENDL  00E8    CMND   1CCA    CNTA   1AF4
8590:  CNTB    1AF5    CNTC   1AF6    CNTD   1AF7    CNTE   1AFC
8600:  CNTF    1AFD    CNTG   1AFE    CNTH   1AFF    CONVD  1DDF
8610:  COUNT   00F7    CURADH 00E7    CURADL 00E6    DAMODE 1C66
8620:  DATA    1C90    DELAY  1DEB    DELETE 1D2C    DNA    1EC2
8630:  DNEND   1ED2    DNLOOP 1EAE    DOWN   1EA6    EDETA  1AE4
8640:  EDETB   1AE5    EDETD  1AE7    EDIT   1CC0    EDITOR 1CB5
8650:  ENDADH  00E5    ENDADL 00E4    ERRA   1D39    ERRB   1D45
8660:  FILBUF  1D51    FILLWS 1E47    GETBYT 1D6F    GETKEA 1DFB
8670:  GETKEY  1DF9    GETLBL 1F35    GOEXEC 1C45    ILLKEY 1C8C
8680:  INH     00F9    INL    00F8    INPUT  1D09    INSERT 1CFB
8690:  IRQH    1A7F    IRQL   1A7E    JUMPS  1FB6    KEYIN  1E09
8700:  KEYINA  1E0B    KEYINB 1E11    KEYINC 1E19    KEYIND 1E1C
8710:  KEY     00E1    LABELS 00EE    LENACC 1E60    LENEND 1E80
8720:  LEN     1F1F    LOOK   1F0F    LOYS   1C00    MODE   00FF
8730:  MOVADH  00EB    MOVADL 00EA    NEXT   1EF8    NIBBLE 00FE
8740:  NMIH    1A7B    NMIL   1A7A    NXTINS 1F8D    ONEKEY 1DB5
8750:  OPLEN   1E5C    PADD   1A81    PAD    1A80    PASSA  1F65
8760:  PASSB   1F95    PB     1FAA    PBDD   1A83    PBD    1A82
8770:  PCH     00F0    PCKEY  1C7D    PCL    00EF    POINTH 00FB
8780:  POINTL  00FA    PREG   00F1    RDA    1E44    RDB    1E46
8790:  RDFLAG  1AD5    RDINST 1E20    RECEND 1EEA    RESET  1C1D
8800:  SAVE    1C00    SAVEA  1C05    SAVEB  1C0F    SCAN   1D4D
8810:  SCANA   1D5C    SCANB  1D61    SCAND  1D88    SCANDS 1D8E
8820:  SCDSA   1D97    SCDSB  1DAC    SEARA  1CF1    SEARCH 1CCD
8830:  SELOOP  1CE2    SHOW   1DCC    SKIP   1D21    SPUSER 00F2
8840:  STARA   1C38    START  1C33    STEP   1C70    STEPA  1C7A
8850:  SYMA    1F39    SYMB   1F4A    SYMNXT 1F4B    TABLEH 00ED
8860:  TABLEL  00EC    TEMP   00FC    TEMPX  00FD    UP     1E83
8870:  UPA     1E99    UPLOOP 1E8B    WDETC  1AE6    WS     1E51
8880:  XREG    00F5    YREG   00F4
```

Source-Listing:

* Binary Decimal Conversion
* Demo Routine
* Play Routine
* Input Routine
* Repeat Routine
* diverse Subroutinen
* diverse Interruptroutinen

```
0010:                BINARY DECIMAL CONVERSION
0020:
0030:
0040: 0200                   ORG   $0200
0050:
0060:                MEMORY CELLS
0070:
0080: 0200           INH     *     $00F9  DISPLAY BUFFERS
0090: 0200           POINTL  *     $00FA
0100: 0200           POINTH  *     $00FB
0110: 0200           HEXL    *     $00D7  DATA BUFFERS
0120: 0200           HEXH    *     $00D8
0130:
0140:                MONITOR SUBROUTINE GETBYT
0150:
0160: 0200           GETBYT  *     $1D6F  KEYBOARD & DISPLAY SCAN
0170:
0180:                START OF DISPL
0190:
0200: 0200 A9 00     DISPL   LDAIM $00    DISPLAY 000000
0210: 0202 85 F9             STAZ  INH
0220: 0204 85 FA             STAZ  POINTL
0230: 0206 85 FB             STAZ  POINTH
0240: 0208 20 6F 1D  DA      JSR   GETBYT READ KEYBOARD, SCAN DISPLAY
0250: 020B 10 F3             BPL   DISPL  RETURN IF COMMAND KEY
0260: 020D 85 F9             STAZ  INH    BYTE TO DISPLAY BUFFER
0270: 020F 85 D7             STAZ  HEXL   BYTE TO DATA BUFFER
0280: 0211 20 17 02          JSR   HEXDEC BINARY DECIMAL CONVERSION
0290: 0214 4C 08 02          JMP   DA     WAIT FOR A NEW BYTE
0300:
0310:
0320:                SUBROUTINES OF THE CONVERSION PROGRAM
0330:
0340: 0217 20 2E 02  HEXDEC JSR   COMNUM COMPUTE ONES
0350: 021A 85 FA             STAZ  POINTL DISCARD ONES
0360: 021C 84 D7             STYZ  HEXL   GET CONTENTS OF THE SUBTRACTION COUNTER
0370: 021E 20 2E 02          JSR   COMNUM COMPUTE TENS
0380: 0221 A2 04             LDXIM $04    SET SHIFT COUNTER
0390: 0223 0A        HD      ASLA         SHIFT LEFT
0400: 0224 CA                DEX          ALL SHIFTS DONE
0410: 0225 D0 FC             BNE   HD      IF NOT, CONTINUE
0420: 0227 05 FA             ORAZ  POINTL TENS & ONES INTO 1 BYTE
0430: 0229 85 FA             STAZ  POINTL
0440: 022B 84 FB             STYZ  POINTH HUNDREDS TO DISPLAY BUFFER
0450: 022D 60                RTS
0460:
0470: 022E A0 00     COMNUM LDYIM $00    RESET HIGH ORDER HEX BUFFER
0480: 0230 84 D8             STYZ  HEXH
0490: 0232 20 3B 02          JSR   SUBTRA SUBTRACT Y*$0A
0500: 0235 18                CLC
0510: 0236 A5 D7             LDAZ  HEXL   CORRECT SUBTRACTION ERROR
0520: 0238 69 0A             ADCIM $0A
0530: 023A 60                RTS
0540:
0550: 023B 38        SUBTRA SEC
0560: 023C A5 D7             LDAZ  HEXL   16 BIT SUBTRACTION
0570: 023E E9 0A             SBCIM $0A
0580: 0240 85 D7             STAZ  HEXL
0590: 0242 A5 D8             LDAZ  HEXH
0600: 0244 E9 00             SBCIM $00
0610: 0246 30 04             BMI   SUB    COMPLETE SUBTRACTION, IF RESULT NEGATIVE
0620: 0248 C8                INY          SUBTRACTION COUNTER = Y+1
0630: 0249 4C 3B 02          JMP   SUBTRA CONTINUE SUBTRACTION
0640: 024C 60        SUB     RTS
0650:
0660:                END OF DISPL

    SYMBOL TABLE
    COMNUM 022E    DA     0208    DISPL  0200    GETBYT 1D6F
    HD     0223    HEXDEC 0217    HEXH   00D8    HEXL   00D7
    INH    00F9    POINTH 00FB    POINTL 00FA    SUBTRA 023B
    SUB    024C


    SYMBOL TABLE
    HEXL   00D7    HEXH   00D8    INH    00F9    POINTL 00FA
    POINTH 00FB    DISPL  0200    DA     0208    HEXDEC 0217
    HD     0223    COMNUM 022E    SUBTRA 023B    SUB    024C
    GETBYT 1D6F

0010:                DEMO ROUTINE
0020:
0030: 0000                   ORG   $0000
0040:
0050:                I/O DEFINITION
0060:
0070: 0000           PAD     *     $1A00  DATA REGISTER
0080: 0000           PADD    *     $1A01  DATA DIRECTION REGISTER
0090: 0000           PBD     *     $1A02  DATA REGISTER
0100: 0000           PBDD    *     $1A03  DATA DIRECTION REGISTER
0110:
0120: 0000 A2 00     DEMO    LDXIM $00
0130: 0002 8E 01 1A          STX   PADD   PA0...PA7 IS INPUT
0140: 0005 E8                INX
0150: 0006 8E 03 1A          STX   PBDD   PB0 IS OUTPUT
```

```
0160:
0170: 0009 AD 80 1A  FREQ   LDA   PAD     READ SWITCH PATTERN
0180: 000C 49 FF            EORIM $FF     INVERT PATTERN
0190: 000E A0 00            LDYIM $00     B0 IS ZERO
0200: 0010 8C 82 1A         STY   PBD     TOGGLE SPEAKER ON
0210: 0013 20 20 00         JSR   DELAY   DELAY = SWITCHES * LOOP TIME
0220: 0016 C8               INY
0230: 0017 8C 82 1A         STY   PBD     TOGGLE SPEAKER OFF
0240: 001A 20 20 00         JSR   DELAY   DELAY = SWITCHES * LOOP TIME
0250: 001D 4C 09 00         JMP   FREQ    RETURN
0260:
0270:                  SUBROUTINE DELAY
0280:
0290: 0020 AA        DELAY  TAX           X-REGISTER IS THE DELAY COUNTER
0300: 0021 CA        DEL    DEX           DELAY LOOP
0310: 0022 D0 FD            BNE   DEL
0320: 0024 60               RTS
0330:

    SYMBOL TABLE
    DELAY  0020    DEL    0021    DEMO   0000    FREQ   0009
    PADD   1A81    PAD    1A80    PBDD   1A83    PBD    1A82


    SYMBOL TABLE
    DEMO   0000    FREQ   0009    DELAY  0020    DEL    0021
    PAD    1A80    PADD   1A81    PBD    1A82    PBDD   1A83

0010:                  PLAY ROUTINE
0020:
0030: 0000                 ORG   $0000
0040:
0050:                  TEMPORARY DATA BUFFERS IN PAGE ZERO
0060:
0070: 0000           ROW    *     $00D9  ROW BUFFER
0080: 0000           KEY    *     $00DA  KEY VALUE BUFFER
0090: 0000           TEMPX  *     $00DB  ROW NUMBER BUFFER
0100:
0110:                  I/O DEFINITION
0120:
0130: 0000           PAD    *     $1A80
0140: 0000           PADD   *     $1A81
0150: 0000           PBD    *     $1A82
0160: 0000           PBDD   *     $1A83
0170:
0180:
0190: 0000 A9 F0     PLAY   LDAIM $F0    PA7...PA4 IS OUTPUT
0200: 0002 8D 81 1A         STA   PADD    PA3...PA0 IS INPUT
0210: 0005 A9 01            LDAIM $01
0220: 0007 8D 83 1A         STA   PBDD    PB0 IS OUTPUT
0230: 000A 8D 82 1A         STA   PBD     TOGGLE SPEAKER OFF
0240: 000D A9 00            LDAIM $00
0250: 000F 8D 80 1A         STA   PAD     ALL MATRIX ROWS ARE ZERO
0260:
0270: 0012 20 9C 00  PA     JSR   KEYIN   ANY KEY DEPRESSED?
0280: 0015 F0 FB            BEQ   PA      BRANCH, IF NO
0290: 0017 20 A4 00         JSR   DELAY   DEBOUNCE THE KEY
0300: 001A 20 9C 00         JSR   KEYIN   KEY STILL DEPRESSED
0310: 001D F0 F3            BEQ   PA      IF YES, CONTINUE
0320: 001F 20 44 00         JSR   KEYVAL  COMPUTE THE KEY VALUE
0330: 0022 A4 DA            LDYZ  KEY     GET KEY VALUE
0340:
0350: 0024 A9 00     TONE   LDAIM $00    B0 = 0
0360: 0026 8D 82 1A         STA   PBD     TOGGLE SPEAKER ON
0370: 0029 BE 00 1A         LDXY  DEL     FETCH THE FREQUENCY
0380: 002C 20 AA 00  TA     JSR   EQUAL   EQUALIZE 22 MICRO SEC.
0390: 002F CA               DEX           HALF PERIODE PASSED?
0400: 0030 D0 FA            BNE   TA      WAIT, IF NOT
0410: 0032 A9 01            LDAIM $01     B0 = 1
0420: 0034 8D 82 1A         STA   PBD     TOGGLE SPEAKER OFF
0430: 0037 BE 00 1A         LDXY  DEL     FETCH THE FREQUENCY AGAIN
0440: 003A 20 9C 00  TB     JSR   KEYIN   ANY KEY STILL DEPRESSED?
0450: 003D F0 D3            BEQ   PA      IF YES, GENERATE A TONE
0460: 003F CA               DEX           HALF PERIODE PASSED?
0470: 0040 D0 F8            BNE   TB      WAIT, IF NOT
0480: 0042 F0 E0            BEQ   TONE    CONTINUE, IF YES
0490:
0500:

    SYMBOL TABLE
    DELA   00A6    DELAY  00A4    DEL    1A00    EQUAL  00AA
    KEYA   004A    KEYB   0063    KEYC   0094    KEYIN  009C
    KEYVAL 0044    KEY    00DA    PA     0012    PADD   1A81
    PAD    1A80    PBDD   1A83    PBD    1A82    PLAY   0000
    ROWA   006E    ROWB   0078    ROWC   0082    ROWD   008C
    ROW    00D9    TA     002C    TB     003A    TEMPX  00DB
    TONE   0024


    SYMBOL TABLE
    PLAY   0000    PA     0012    TONE   0024    TA     002C
    TB     003A    KEYVAL 0044    KEYA   004A    KEYB   0063
    ROWA   006E    ROWB   0078    ROWC   0082    ROWD   008C
    KEYC   0094    KEYIN  009C    DELAY  00A4    DELA   00A6
    EQUAL  00AA    ROW    00D9    KEY    00DA    TEMPX  00DB
    DEL    1A00    PAD    1A80    PADD   1A81    PBD    1A82
    PBDD   1A83
```

```
0010:                 SUBROUTINES OF THE PLAY PROGRAM
0020:
0030: 0044 A9 F7      KEYVAL LDAIM $F7     ALL ROWS ARE ONE
0040: 0046 85 D9             STAZ  ROW
0050: 0048 A2 04             LDXIM $04     SET UP ROW COUNTER
0060: 004A CA         KEYA   DEX           RETURN IF INVALID ROW
0070: 004B 30 F7             BMI   KEYVAL IS FOUND
0080: 004D 06 D9             ASLZ  ROW     SPECIFIED ROW IS ZERO
0090: 004F A5 D9             LDAZ  ROW
0100: 0051 8D 80 1A          STA   PAD     OUTPUT ROW NUMBER
0110: 0054 AD 80 1A          LDA   PAD     IF NO KEY IS DEPRESSED IN THE
0120: 0057 29 0F             ANDIM $0F     SPECIFIED ROW, OUTPUT
0130: 0059 C9 0F             CMPIM $0F     NEXT ROW NUMBER
0140: 005B F0 ED             BEQ   KEYA
0150: 005D 86 DB             STXZ  TEMPX   SAVE ROW NUMBER
0160: 005F 85 DA             STAZ  KEY     SAVE COLUMN NUMBER
0170: 0061 A2 00             LDXIM $00
0180: 0063 46 DA      KEYB   LSRZ  KEY     SHIFT UNTIL CARRY CLEAR
0190: 0065 90 07             BCC   ROWA    BRANCH TO COMPUTE KEY VALUE
0200: 0067 E8                INX
0210: 0068 E0 04             CPXIM $04     ALL ROWS SCANNED?
0220: 006A D0 F7             BNE   KEYB    IF NOT CONTINUE
0230: 006C F0 D6             BEQ   KEYVAL RETURN IF INVALID ROW NUMBER
0240: 006E A5 DB      ROWA   LDAZ  TEMPX   GET ROW NUMBER AGAIN
0250: 0070 C9 03             CMPIM $03     ROW 0?
0260: 0072 D0 04             BNE   ROWB
0270: 0074 8A                TXA
0280: 0075 4C 94 00          JMP   KEYC
0290:
0300: 0078 C9 02      ROWB   CMPIM $02     ROW 1?
0310: 007A D0 06             BNE   ROWC
0320: 007C 8A                TXA
0330: 007D 18                CLC
0340: 007E 69 04             ADCIM $04
0350: 0080 D0 12             BNE   KEYC
0360:
0370: 0082 C9 01      ROWC   CMPIM $01     ROW 2?
0380: 0084 D0 06             BNE   ROWD
0390: 0086 8A                TXA
0400: 0087 18                CLC
0410: 0088 69 08             ADCIM $08
0420: 008A D0 08             BNE   KEYC
0430:
0440: 008C C9 00      ROWD   CMPIM $00     ROW 3?
0450: 008E D0 B4             BNE   KEYVAL RETURN, IF ROW IS INVALID
0460: 0090 8A                TXA
0470: 0091 18                CLC
0480: 0092 69 0C             ADCIM $0C
0490:
0500: 0094 85 DA      KEYC   STAZ  KEY     SAVE KEY VALUE
0510: 0096 A9 00             LDAIM $00     RESET PORT A
0520: 0098 8D 80 1A          STA   PAD
0530: 009B 60                RTS
0540:
0550: 009C AD 80 1A   KEYIN  LDA   PAD
0560: 009F 29 0F             ANDIM $0F     MASK OFF HIGH ORDER NIBBLE
0570: 00A1 49 0F             EORIM $0F     IF NO KEY: ACCU = $00
0580: 00A3 60                RTS
0590:
0600: 00A4 A0 FF      DELAY  LDYIM $FF     SET DELAY COUNTER
0610: 00A6 88         DELA   DEY
0620: 00A7 D0 FD             BNE   DELA    TIME OUT ?
0630: 00A9 60                RTS
0640:
0650: 00AA EA         EQUAL  NOP           EQUALIZE 20 MICRO SEC
0660: 00AB EA                NOP
0670: 00AC EA                NOP
0680: 00AD EA                NOP
0690: 00AE EA                NOP
0700: 00AF 60                RTS
0710:
0720: 1A00                   ORG   $1A00
0730:
0740:                 FREQUENCY LOOKUP TABLE
0750:
0760: 1A00 8E         DEL    =     $8E
0770: 1A01 86                =     $86
0780: 1A02 7E                =     $7E
0790: 1A03 77                =     $77
0800: 1A04 70                =     $70
0810: 1A05 6A                =     $6A
0820: 1A06 64                =     $64
0830: 1A07 5E                =     $5E
0840: 1A08 59                =     $59
0850: 1A09 54                =     $54
0860: 1A0A 4E                =     $4E
0870: 1A0B 4A                =     $4A
0880: 1A0C 47                =     $47
0890: 1A0D 43                =     $43
0900: 1A0E 3E                =     $3E
0910: 1A0F 3C                =     $3C
0920:
0930:
0940:                 END OF PLAY
```

```
                    INPUT ROUTINE

                          ORG   $0200

                    TEMPORARY DATA BUFFERS IN PAGE ZERO

ROW       *       $00D9
KEY       *       $00DA
TEMPX     *       $00DB
NOTEL     *       $00DC   NOTE POINTER
NOTEH     *       $00DD
LENGTH    *       $00DE   TIME OF A DEPRESSED KEY
ENDL      *       $00DF   END OF THE NOTE BUFFER

                    INTERVAL TIMER

CNTA      *       $1AF4   DISABLE TIMER IRQ
CNTG      *       $1AFE   ENABLE TIMER IRQ, CLK64T

                    GOTO MONITOR

RESET     *       $1C1D   NEW I/O DEFINITION

                    I/O DEFINITION

PAD       *       $1A80
PADD      *       $1A81
PBD       *       $1A82
PBDD      *       $1A83

                    IRQ VECTOR

IRQL      *       $1A7E
IRQH      *       $1A7F


                    START OF THE INPUT PROGRAM

0200 78       INPUT SEI             DISABLE IRQ LINE
0201 D8             CLD
0202 A9 20          LDAIM IRQIN     SET UP IRQ VECTOR
0204 8D 7E 1A       STA   IRQL
0207 A9 1A          LDAIM IRQIN     /256
0209 8D 7F 1A       STA   IRQH
020C A9 F0          LDAIM $F0       PA7...PA4 IS OUTPUT, PA3...PA0 IS INPUT
020E 8D 81 1A       STA   PADD
0211 A9 01          LDAIM $01
0213 8D 83 1A       STA   PBDD      PB0 IS OUTPUT FOR SPEAKER
0216 8D 82 1A       STA   PBD       TOGGLE SPEAKER OFF
0219 85 DD          STAZ  NOTEH     HIGH ORDER BYTE OF NOTE POINTER
021B A0 00          LDYIM $00
021D 8C 80 1A       STY   PAD       SET ALL ROWS ZERO
0220 84 DC          STYZ  NOTEL     LOW ORDER BYTE OF NOTE POINTER
0222 A0 D8          LDYIM $D8       DEFINE ENDADDRESS OF INPUT BUFFER
0224 84 DF          STYZ  ENDL
0226 A9 77          LDAIM $77       LOAD EOF CHARACTER
0228 91 DC    INA   STAIY NOTEL     FILLUP WORKSPACE WITH EOFS
022A 88             DEY
022B C0 FF          CPYIM $FF       WS FILLED UP?
022D D0 F9          BNE   INA       IF NOT CONTINUE
022F 20 E0 02 KEYSCN JSR  KEYIN     ANY KEY DEPRESSED?
0232 F0 FB          BEQ   KEYSCN    WAIT IF NO KEY IS DEPRESSED
0234 20 E8 02       JSR   DELAY     DEBOUNCE KEYBOARD
0237 20 E0 02       JSR   KEYIN     STILL ANY KEY DEPRESSED
023A F0 F3          BEQ   KEYSCN    IF YES, CONTINUE
023C 20 88 02       JSR   KEYVAL    COMPUTE KEY NUMBER
023F A9 00          LDAIM $00
0241 85 DE          STAZ  LENGTH    RESET TIME COUNTER
0243 A9 FF          LDAIM $FF       START TIMER, ENABLE TIMER IRQ,
0245 8D FE 1A       STA   CNTG      RESET IRQ LINE
0248 58             CLI             ENABLE IRQ LINE
0249 A4 DA          LDYZ  KEY       LOOKUP CONVERSION BY KEY VALUE
024B A9 00    TONE  LDAIM $00       TOGGLE SPEAKER ON
024D 8D 82 1A       STA   PBD       PB0 IS LOG 0
0250 BE 00 1A       LDXY  DEL       FETCH DELAY
0253 20 EE 02 TA    JSR   EQUAL     EQUALIZE 20 MIKRO SEC
0256 CA             DEX
0257 D0 FA          BNE   TA        DELAY
0259 A9 01          LDAIM $01       TOGGLE SPEAKER OFF
025B 8D 82 1A       STA   PBD
025E BE 00 1A       LDXY  DEL       FETCH DELAY AGAIN
0261 20 E0 02 TB    JSR   KEYIN     ANY KEY STILL DEPRESSED?
0264 F0 05          BEQ   STORE     BRANCH IF KEY IS RELEASED
0266 CA             DEX
0267 D0 F8          BNE   TB
0269 F0 E0          BEQ   TONE      CONTINUE AS LONG AS A KEY IS DEPRESSED
026B 8D F4 1A STORE STA   CNTA      RESET IRQ LINE,DISABLE TIMER IRQ
026E A5 DC          LDAZ  NOTEL
0270 C5 DF          CMPZ  ENDL      IS WORKSPACE FULL?
0272 F0 11          BEQ   ST        IF YES, EXIT HERE
0274 98             TYA
0275 A0 00          LDYIM $00
0277 91 DC          STAIY NOTEL     STORE KEY VALUE IN WS
0279 C8             INY
027A A5 DE          LDAZ  LENGTH    GET TIME OF THE DEPRESSED KEY
```

```
0960: 027C 91 DC            STAIY NOTEL  STORE KEY TIME IN WS
0970: 027E E6 DC            INCZ  NOTEL
0980: 0280 E6 DC            INCZ  NOTEL  ADJUST NOTE POINTER
0990: 0282 4C 2F 02         JMP   KEYSCN
1000: 0285 4C 1D 1C  ST     JMP   RESET  BACK TO MONITOR
1010:
0010:                SUBROUTINES OF THE INPUT PROGRAM
0020:
0030: 0288 A9 F7     KEYVAL LDAIM $F7    ALL ROWS ARE ONE
0040: 028A 85 D9            STAZ  ROW
0050: 028C A2 04            LDXIM $04    SET UP ROW COUNTER
0060: 028E CA        KEYA   DEX          RETURN IF INVALID ROW
0070: 028F 30 F7            BMI   KEYVAL IS FOUND
0080: 0291 06 D9            ASLZ  ROW    SPECIFIED ROW IS ZERO
0090: 0293 A5 D9            LDAZ  ROW
0100: 0295 8D 80 1A         STA   PAD    OUTPUT ROW NUMBER
0110: 0298 AD 80 1A         LDA   PAD    IF NO KEY IS DEPRESSED IN THE
0120: 029B 29 0F            ANDIM $0F    SPECIFIED ROW, OUTPUT
0130: 029D C9 0F            CMPIM $0F    NEXT ROW NUMBER
0140: 029F F0 ED            BEQ   KEYA
0150: 02A1 86 DB            STXZ  TEMPX  SAVE ROW NUMBER
0160: 02A3 85 DA            STAZ  KEY    SAVE COLUMN NUMBER
0170: 02A5 A2 00            LDXIM $00
0180: 02A7 46 DA     KEYB   LSRZ  KEY    SHIFT UNTIL CARRY CLEAR
0190: 02A9 90 07            BCC   ROWA   BRANCH TO COMPUTE KEY VALUE
0200: 02AB E8               INX
0210: 02AC E0 04            CPXIM $04    ALL ROWS SCANNED?
0220: 02AE D0 F7            BNE   KEYB   IF NOT CONTINUE
0230: 02B0 F0 D6            BEQ   KEYVAL RETURN IF INVALID ROW NUMBER
0240: 02B2 A5 DB     ROWA   LDAZ  TEMPX  GET ROW NUMBER AGAIN
0250: 02B4 C9 03            CMPIM $03    ROW 0?
0260: 02B6 D0 04            BNE   ROWB
0270: 02B8 8A               TXA
0280: 02B9 4C D8 02         JMP   KEYC
0290:
0300: 02BC C9 02     ROWB   CMPIM $02    ROW 1?
0310: 02BE D0 06            BNE   ROWC
0320: 02C0 8A               TXA
0330: 02C1 18               CLC
0340: 02C2 69 04            ADCIM $04
0350: 02C4 D0 12            BNE   KEYC
0360:
0370: 02C6 C9 01     ROWC   CMPIM $01    ROW 2?
0380: 02C8 D0 06            BNE   ROWD
0390: 02CA 8A               TXA
0400: 02CB 18               CLC
0410: 02CC 69 08            ADCIM $08
0420: 02CE D0 08            BNE   KEYC
0430:
0440: 02D0 C9 00     ROWD   CMPIM $00    ROW 3?
0450: 02D2 D0 B4            BNE   KEYVAL RETURN, IF ROW IS INVALID
0460: 02D4 8A               TXA
0470: 02D5 18               CLC
0480: 02D6 69 0C            ADCIM $0C
0490:
0500: 02D8 85 DA     KEYC   STAZ  KEY    SAVE KEY VALUE
0510: 02DA A9 00            LDAIM $00    RESET PORT A
0520: 02DC 8D 80 1A         STA   PAD
0530: 02DF 60               RTS
0540:
0550: 02E0 AD 80 1A  KEYIN  LDA   PAD
0560: 02E3 29 0F            ANDIM $0F    MASK OFF HIGH ORDER NIBBLE
0570: 02E5 49 0F            EORIM $0F    IF NO KEY: ACCU = $00
0580: 02E7 60               RTS
0590:
0600: 02E8 A0 FF     DELAY  LDYIM $FF    SET DELAY COUNTER
0610: 02EA 88        DELA   DEY
0620: 02EB D0 FD            BNE   DELA   TIME OUT ?
0630: 02ED 60               RTS
0640:
0650: 02EE EA        EQUAL  NOP          EQUALIZE 20 MICRO SEC
0660: 02EF EA               NOP
0670: 02F0 EA               NOP
0680: 02F1 EA               NOP
0690: 02F2 EA               NOP
0700: 02F3 60               RTS
0710:
```

```
0720: 1A00                ORG   $1A00
0730:
0740:               FREQUENCY LOOKUP TABLE
0750:
0760: 1A00 8E       DEL     =     $8E
0770: 1A01 86               =     $86
0780: 1A02 7E               =     $7E
0790: 1A03 77               =     $77
0800: 1A04 70               =     $70
0810: 1A05 6A               =     $6A
0820: 1A06 64               =     $64
0830: 1A07 5E               =     $5E
0840: 1A08 59               =     $59
0850: 1A09 54               =     $54
0860: 1A0A 4E               =     $4E
0870: 1A0B 4A               =     $4A
0880: 1A0C 47               =     $47
0890: 1A0D 43               =     $43
0900: 1A0E 3E               =     $3E
0910: 1A0F 3C               =     $3C
0920:
0930:
0940: 1A20                ORG   $1A20
0950:
0960:               TIMER INTERRUPT PROGRAM
0970:
0980: 1A20 48       IRQIN   PHA           SAVE ACCU
0990: 1A21 E6 DE            INCZ  LENGTH  INCREMENT TIME
1000: 1A23 A9 FF            LDAIM $FF     TIMER OFFSET IS $FF
1010: 1A25 8D FE 1A         STA   CNTG    START TIMER AGAIN
1020: 1A28 68               PLA           RESTORE ACCU
1030: 1A29 40               RTI
1040:
1050:               END OF INPUT
```

```
SYMBOL TABLE
CNTA    1AF4    CNTG    1AFE    DELA    02EA    DELAY   02E8
DEL     1A00    ENDL    00DF    EQUAL   02EE    INA     0228
INPUT   020C    IRQH    1A7F    IRQIN   1A20    IRQL    1A7E
KEYA    028E    KEYB    02A7    KEYC    02D8    KEYIN   02E0
KEYSCN  022F    KEYVAL  0288    KEY     00DA    LENGTH  00DE
NOTEH   00DD    NOTEL   00DC    PADD    1A81    PAD     1A80
PBDD    1A83    PBD     1A82    RESET   1C1D    ROWA    02B2
ROWB    02BC    ROWC    02C6    ROWD    02D0    ROW     00D9
ST      0285    STORE   026B    TA      0253    TB      0261
TEMPX   00DB    TONE    024B
```

```
SYMBOL TABLE
ROW     00D9    KEY     00DA    TEMPX   00DB    NOTEL   00DC
NOTEH   00DD    LENGTH  00DE    ENDL    00DF    INPUT   020C
INA     0228    KEYSCN  022F    TONE    024B    TA      0253
TB      0261    STORE   026B    ST      0285    KEYVAL  0288
KEYA    028E    KEYB    02A7    ROWA    02B2    ROWB    02BC
ROWC    02C6    ROWD    02D0    KEYC    02D8    KEYIN   02E0
DELAY   02E8    DELA    02EA    EQUAL   02EE    DEL     1A00
IRQIN   1A20    IRQL    1A7E    IRQH    1A7F    PAD     1A80
PADD    1A81    PBD     1A82    PBDD    1A83    CNTA    1AF4
CNTG    1AFE    RESET   1C1D
```

```
0010:                   REPEAT ROUTINE
0020:
0030: 0000                    ORG   $0000
0040:
0050:                   TEMPORARY DATABUFFERS IN PAGE ZERO
0060:
0070: 0000              KEY     *     $00DA
0080: 0000              NOTEL   *     $00DC
0090: 0000              NOTEH   *     $00DD
0100: 0000              LENGTH  *     $00DE
0110:
0120:                   INTERVAL TIMER
0130:
0140: 0000              CNTA    *     $1AF4  DISABLE TIMER IRQ
0150: 0000              CNTD    *     $1AF7  DISABLE TIMER IRQ, CLK1KT
0160: 0000              CNTG    *     $1AFE  ENABLE TIMER IRQ, CLK64T
0170: 0000              RDFLAG  *     $1AD5  B7 IS TIMER FLAG
0180:
0190:                   GOTO MONITOR
0200:
0210: 0000              RESET   *     $1C1D  NEW I/O DEFINITION
0220:
0230:                   I/O DEFINITION
0240:
0250: 0000              PBD     *     $1A82
0260: 0000              PBDD    *     $1A83
0270:
0280:                   IRQ VECTOR
0290:
0300: 0000              IRQL    *     $1A7E
0310: 0000              IRQH    *     $1A7F
0320:
0330:
0340:                   START OF THE REPEAT PROGRAM
0350:
0360: 0000 78           REPEAT SEI          DISABLE IRQ LINE
0370: 0001 D8                  CLD
0380: 0002 A9 30               LDAIM IRQRE  SET UP IRQ VECTOR
0390: 0004 8D 7E 1A            STA   IRQL
0400: 0007 A9 1A               LDAIM IRQRE  /256
0410: 0009 8D 7F 1A            STA   IRQH
0420: 000C A9 01               LDAIM $01    PB0 IS OUTPUT
0430: 000E 8D 83 1A            STA   PBDD
0440: 0011 8D 82 1A            STA   PBD    TOGGLE SPEAKER OFF
0450: 0014 85 DD               STAZ  NOTEH  SET NOTE POINTER
0460: 0016 A9 00               LDAIM $00
0470: 0018 85 DC               STAZ  NOTEL  SET NOTE POINTER
0480: 001A 8D F4 1A            STA   CNTA   RESET IRQ LINE, DISABLE TIMER IRQ
0490: 001D 58                  CLI          ENABLE CPU IRQ
0500:
0510: 001E A9 FF        FETCH  LDAIM $FF    SET TIMER ENABLE TIMER IRQ
0520: 0020 8D FE 1A            STA   CNTG
0530: 0023 A0 00               LDYIM $00    FETCH NOTE
0540: 0025 B1 DC               LDAIY NOTEL
0550: 0027 85 DA               STAZ  KEY
0560: 0029 C8                  INY          FETCH LENGTH
0570: 002A B1 DC               LDAIY NOTEL
0580: 002C 85 DE               STAZ  LENGTH
0590: 002E A4 DA               LDYZ  KEY    LOOKUP CONVERSION
0600:
0610: 0030 A9 00        TONE   LDAIM $00    TOGGLE SPEAKER ON
0620: 0032 8D 82 1A            STA   PBD
0630: 0035 BE 00 1A            LDXY  DEL    GET FREQUENCY
0640: 0038 20 70 00     TONEA  JSR   EQUALA DELAY 22 MICRO SEC
0650: 003B CA                  DEX
0660: 003C D0 FA               BNE   TONEA  LOOP TIME IS 27 MIKRO SEC*X
0670: 003E A9 01               LDAIM $01    TOGGLE SPEAKER OFF
0680: 0040 8D 82 1A            STA   PBD
0690: 0043 BE 00 1A            LDXY  DEL    GET FREQUENCY AGAIN
0700: 0046 A5 DE        TONEB  LDAZ  LENGTH GET LENGTH
0710: 0048 30 08               BMI   TONEC  TIME OUT?
0720: 004A 20 74 00            JSR   EQUALB EQUALIZE 17 MICRO SEC
0730: 004D CA                  DEX
0740: 004E D0 F6               BNE   TONEB  LOOP TIME IS 27 MICRO SEC*X AGAIN
0750: 0050 F0 DE               BEQ   TONE   RETURN AFTER ONE PERIODE
0760: 0052 A2 04        TONEC  LDXIM $04    LOOP TIME = 4*CNTD*PRESET
0770: 0054 A9 30        TONED  LDAIM $30    PRESET = $30
0780: 0056 8D F7 1A            STA   CNTD   DISABLE TIMER IRQ
0790:
0800: 0059 2C D5 1A     POLL   BIT   RDFLAG READ FLAG REGISTER, TIME OUT?
0810: 005C 10 FB               BPL   POLL   IS TIMER FLAG STILL ZERO?
0820: 005E CA                  DEX
0830: 005F D0 F3               BNE   TONED  LOOP COUNTER ZERO?
0840: 0061 E6 DC               INCZ  NOTEL  ADJUST NOTE POINTER
0850: 0063 E6 DC               INCZ  NOTEL
0860: 0065 A0 00               LDYIM $00
0870: 0067 B1 DC               LDAIY NOTEL  END OF NOTE BUFFER?
0880: 0069 C9 77               CMPIM $77    EOF CHARACTER
0890: 006B D0 B1               BNE   FETCH  IF NOT EOF, CONTINUE
0900: 006D 4C 1D 1C            JMP   RESET  ELSE BACK TO MONITOR
0910:
```

```
0010:                   SUBROUTINES OF THE REPEAT PROGRAM
0020:
0030:                   17/22 MICRO SEC SUBROUTINE
0040:
0050: 0070 EA           EQUALA NOP
0060: 0071 4C 74 00             JMP   EQUALB
0070: 0074 EA           EQUALB NOP
0080: 0075 4C 78 00             JMP   EEND
0090: 0078 60           EEND   RTS
0100:
0110: 1A00                      ORG   $1A00
0120:
0130:                   FREQUENCY LOOKUP TABLE
0140:
0150: 1A00 8E           DEL    =     $8E
0160: 1A01 86                  =     $86
0170: 1A02 7E                  =     $7E
0180: 1A03 77                  =     $77
0190: 1A04 70                  =     $70
0200: 1A05 6A                  =     $6A
0210: 1A06 64                  =     $64
0220: 1A07 5E                  =     $5E
0230: 1A08 59                  =     $59
0240: 1A09 54                  =     $54
0250: 1A0A 4E                  =     $4E
0260: 1A0B 4A                  =     $4A
0270: 1A0C 47                  =     $47
0280: 1A0D 43                  =     $43
0290: 1A0E 3E                  =     $3E
0300: 1A0F 3C                  =     $3C
0310:
0320:
0330: 1A30                      ORG   $1A30
0340:
0350:                   TIMER INTERRUPT PROGRAM
0360:
0370: 1A30 48           IRQRE  PHA           SAVE ACCU
0380: 1A31 C6 DE               DECZ  LENGTH DECREMENT TIME
0390: 1A33 A9 FF               LDAIM $FF    TIMER OFFSET IS $FF
0400: 1A35 8D FE 1A            STA   CNTG   START TIMER AGAIN
0410: 1A38 68                  PLA          RESTORE ACCU
0420: 1A39 40                  RTI
0430:
0440:                   END OF REPEAT
```

```
SYMBOL TABLE
CNTA   1AF4   CNTD   1AF7   CNTG   1AFE   DEL    1A00
EEND   0078   EQUALA 0070   EQUALB 0074   FETCH  001E
IRQH   1A7F   IRQL   1A7E   IRQRE  1A30   KEY    00DA
LENGTH 00DE   NOTEH  00DD   NOTEL  00DC   PBDD   1A83
PBD    1A82   POLL   0059   RDFLAG 1AD5   REPEAT 0000
RESET  1C1D   TONE   0030   TONEA  0038   TONEB  0046
TONEC  0052   TONED  0054
```

```
SYMBOL TABLE
REPEAT 0000   FETCH  001E   TONE   0030   TONEA  0038
TONEB  0046   TONEC  0052   TONED  0054   POLL   0059
EQUALA 0070   EQUALB 0074   EEND   0078   KEY    00DA
NOTEL  00DC   NOTEH  00DD   LENGTH 00DE   DEL    1A00
IRQRE  1A30   IRQL   1A7E   IRQH   1A7F   PBD    1A82
PBDD   1A83   RDFLAG 1AD5   CNTA   1AF4   CNTD   1AF7
CNTG   1AFE   RESET  1C1D
```

Sachwortverzeichnis

Adreßbus 35
Adreß-Pointer 147
BEGAD
ENDAD
CURAD
CEND
Adressen, Modifizieren von — 121
Adressen, symbolische — 16, 183
AD-Taste 8
ASCII-Signal 72
Assembler 8, 27, 183
Assembler-Flußdiagramm 190
Assembler-Phase, erste — 184
Assembler-Phase, zweite — 189

Binär-Dezimalumwandlung . . 9, 10, 11
BINDEC 9
BRANCH 8
BRANCH, EPROM-Routine 205
BRANCH-Instruktion 189, 200
BRK-Kommando 109
— Ablauf — 106

Cold Start Entry 24, 156
COMNUM 14
Controlbus 35
CURAD 23

DA-Taste 8
Datenbus 35
Daten, Modifizieren von — 120
Daten-Register 38
Datenrichtungs-Register 38
DELETE-Taste 146
DEMO 43
DISPL 13, 18

Editor 8, 17, 145
Editor-Kommando-Tasten 160
SEARCH
INSERT
INPUT
SKIP
DELETE
End of File 146
EOF 146
Errormeldung 22

Flag
— Timer 60
— PA7 60
Flag-Register 34, 60
Flankendetektor 24, 71
Fehlermeldung 160

GETBYT 12

HEXDEC 13

INPUT 77
Input Mode "I" 39

Interrupts, Ablauf eines — 106
Interrupt-Mode 66
Intervall-Timer 58
IRQ, Auslösen eines — 108
IRQ-Leitung 62

JMP-Instruktion 198
JSR-Instruktion 198

Keyboard-Matrix 131
Kommando-Tasten 22

Label 17, 184
Label-Begrenzer 17
– Indikator 17
– Nummer 17
Latch-Funktion 38
Lookup-Table 27, 53

Monitor 100, 113
– Ausgang 113
– Das Verlassen des – 117
– Eingang 102, 103, 104
– Flußdiagramm, Globales 102
– Springen aus dem – 101
– Springen in den – 100

NMI, Auslösen eines – 106
NMI-Vektor 31

OPLEN 21
Output Mode "O" 40

PA7-Flag 60
PASSA 193
PASSB 196
Peripherer Interface Adapter 33
PIA 33
PIA-Hardware 37
PIA-Programmier-Modell 35
PIA-Software 38
Polling-Mode 67
Port A 35
Port B 35
Port-Register 38
PREG 113
Programm Input 79
Programmier-Modell, PIA – 35
Programmier-Modell
  Flankendetektor 74
Prozessor Status Register 113
Pseudo-OP-Code 184

SAVE 115
SCAND 124
SCANDS 124
SEARCH-Taste 146
SKIP-Taste 146
Springen aus dem Monitor 101
Springen in den Monitor 100
Step by Step 105
SUBTRA 14
Subroutinen-Nummer 18
Subroutinen
  ADCEND 169
  AK 133
  CONVD 136
  DELAY 51
  DOWN 175
  EQUAL 54
  FILLWS 169
  GETBYT 161
  GETKEY 139
  GETLBL 202
  KEYIN 50
  KEYVAL 47
  NEXT 165
  OPLEN/LENACC 179
  RDINST 166
  SHOW 132, 135
  UP 177
symbolische Adressen 16
Symbol-Stack 27, 184
7-Segment-Display 128

Taste
– AD 8
– DA 8
– RST 106

Tasten
  INSERT 22
  INPUT 23
  DELETE 23
  SEARCH 23
  SKIP 24
Tastenkennung 119
Tastenprellen 127
Teiler-Register 59
Time Out 61
Timer 34
Timer-Flag 60
Timerzelle 59
Two Pass Assembler 27

Quellen-Adresse 201

REPEAT 77, 90
RESET 113
resident 100

Routinen
DELETE 172
INSERT 166
INPUT 171
SEARCH 160
SKIP 172
Warm Start Entry 24

Ziel-Adresse 201

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-Computern, 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 feststellen, 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 als Lernsystem angeboten.

Nachdem das erste Buch den Weg in's Computerland ebnete und den Umgang mit dem Junior-Computer beschrieb, enthält Buch II die Programmier-Anleitung: Da ist die Rede von "Monitor", "Editor", "Assemblieren", und "Disassemblieren". Und – das alles wird mittels zahlreicher Tabellen, Flußdiagramme und Zeichnungen ausführlich erklärt. In Buch 3 wird der Junior-Computer zum ausgewachsenen Personal-Computer. Dazu gehören u.a. ein Cassetten- und TTY-Interface, ein 5 KByte-Speicher uvm.

Das System ist also soweit ausbaufähig, wie es die spezielle oder allgemeine Anwendung fordert.
Also dann: Frisch an's Werk!

Elektor Verlag GmbH, D-5133 Gangelt 1

ISBN 3-921608-23-6