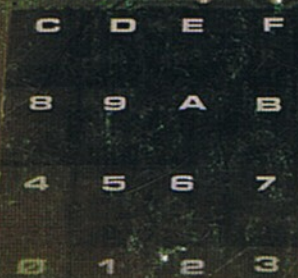


the practical introduction  
to a powerful system

# JUNIOR COMPUTER



Book 2

Elektor

# The Elektor Junior Computer

the practical introduction  
to a powerful system

A.Nachtmann  
G.H.Nachbar

Elektor Publishers Ltd.

**Copyright © 1981 Elektor Publishers Ltd. — Canterbury.**

The contents of this book are copyright and may not be reproduced or imitated in whole or in part without prior written permission of the publishers. This copyright protection also extends to all drawings, photographs and the printed circuits boards.

The circuits published are for domestic use only. Patent protection may exist with respect to circuits, devices, components etc. described in this publication. The publishers do not accept responsibility for failing to identify such patent or other protection.

Printed in the Netherlands  
ISBN 0 905705 076

## Feeling peckish?

How are you enjoying the Junior Computer so far? Everything to satisfaction? Fine. In that case, you must be eager to digest Book 2, a complete menu of computer facts and programming material.

Dig in, folks!

Now that we've got the Junior Computer up and running (Book 1) it is time to construct and construe a number of 'tools' with which to operate the microprocessor as efficiently as possible.

This is where chapter 5 comes in. It presents the programmer with such essential tools as the 'editor' and the 'assembler'. These are highly effective as they enable typing errors to be corrected and additional instructions to be inserted at any point without having to re-enter the entire program. The programmer now no longer has to calculate displacements (during conditional branch instructions and absolute addresses (during jump instructions) himself: all these tiresome chores are dealt with automatically inside the Junior Computer's EPROM.

Chapter 6 teaches the Junior Computer to 'sing'. It includes several circuits which, together with the peripheral interface adapter (PIA), turn the JC into a keyboard instrument. Via the PIA the computer controls a loud-speaker and in Book 3 it will be seen to use the same method to operate a printer.

The remaining chapters (7, 8 and 9) provide a detailed description of the monitor, the editor and the assembler, respectively, all of which combine to form the EPROM brain. Once the user knows exactly how the Junior Computer 'ticks', he/she will be able to write suitable programs personally. Various subroutines introduced in the book will then be of great assistance.

Finally, the appendix gives a clear and coherent summary of all the available subroutines and listings.

After reading Book 2 and mastering the contents, the programmer will be ready to add a printer and a cassette interface to the standard Junior Computer. These peripheral devices, to be 'served up' in Book 3, will transform

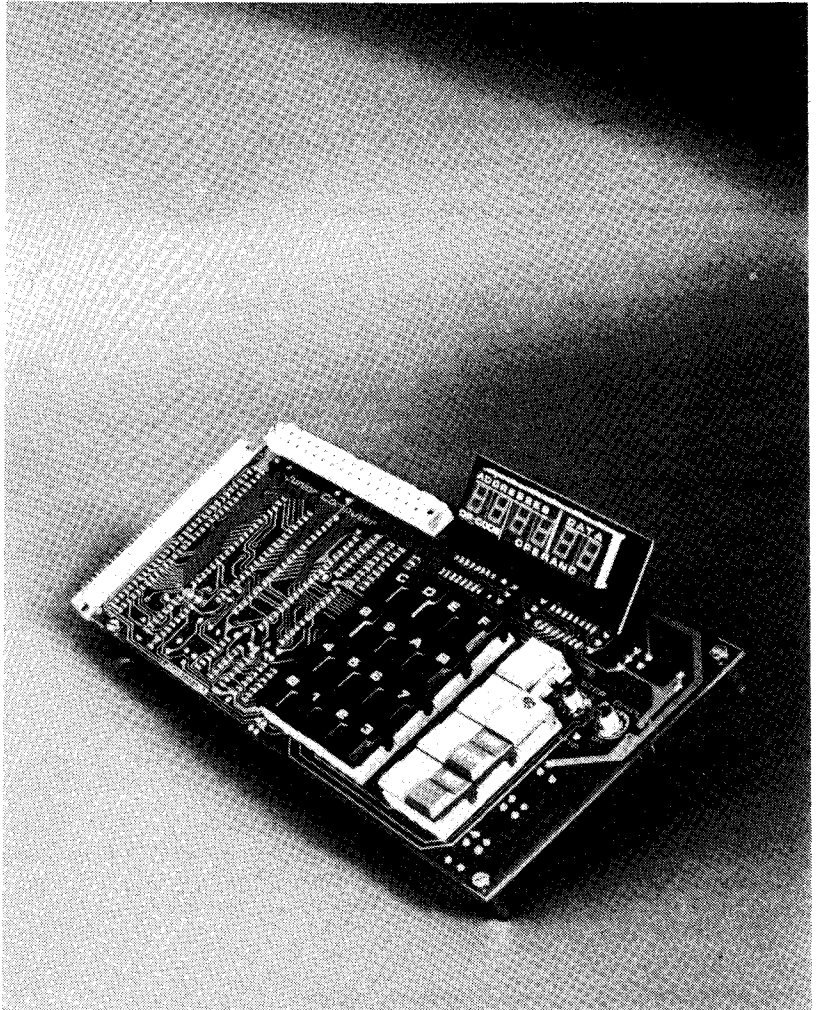
the machine into a full-fledged personal computer. Then the Junior Computer will have well and truly 'outgrown' its name.

The authors.

P.S. Since nothing in this world is perfect – including the Junior Computer! – we will lend a willing ear to any suggestions or comments that readers might care to make.

# Contents

<b>Chapter 5</b> . . . . .	<b>7</b>
<b>The Editor and the Assembler</b>	
<b>Chapter 6</b> . . . . .	<b>31</b>
<b>The Peripheral Interface Adapter or PIA</b>	
<b>Chapter 7</b> . . . . .	<b>91</b>
<b>The Monitor program – <i>Basic ‘housekeeping’ software</i></b>	
<b>Chapter 8</b> . . . . .	<b>133</b>
<b>The Editor program – <i>The ‘intelligence’ behind simple program entry</i></b>	
<b>Chapter 9</b> . . . . .	<b>169</b>
<b>The Assembler program – <i>Tidying up the edited program</i></b>	
<b>Appendix 1</b> . . . . .	<b>193</b>
<b>The program listing of the EPROM</b>	
<b>Appendix 2</b> . . . . .	<b>204</b>
<b>Listings of the programs used in chapters 5 and 6</b>	



# The Editor and the Assembler

Typing in a program is a dull, irksome occupation. As was shown in Book 1, each individual instruction has to be entered, one byte at a time. But before this can be done, a fair amount of paper work is involved:

- subroutine start addresses,
- displacements inherent to conditional branch instructions and
- absolute addresses inherent to unconditional branch instructions (jump instructions) all have to be calculated first.

When the entire program has finally been entered into the computer, it has to be checked for typing errors. Supposing one byte happened to be left out somewhere in the middle of the program. Does this mean retyping the whole lot? No – thanks to the editor and the assembler – it doesn't!

The editor and the assembler are both stored inside the Junior Computer's EPROM. The former allows new instructions to be inserted at any given point in the program, even after entry; in addition, it tracks down instructions and, if necessary, deletes them. Thus, the keyboard and the editor act as a 'pencil and rubber', so to speak, while data is being entered.

The editor also permits the programmer to type in symbolic addresses, called 'labels'. After this, the assembler steps in to calculate subroutine start addresses, displacements in the event of conditional branch instructions and absolute addresses for jump instructions, and all without any human help! As a result, program errors are reduced to a minimum and the user is saved a great deal of unnecessary labour.



## The Editor

Having studied the Junior Computer Book I, you should be familiar with most of the instructions and address modes appertaining to the 6502 microprocessor. Various program examples were given which showed how easy it is to program the JC. Up to now the data entered consisted entirely of hexadecimal numbers. When the AD key was depressed the JC 'knew' it had to interpret the following key information as addresses. Pressing the DA key, on the other hand, told it to store any entered data in the actual address location displayed.

As far as short programs are concerned, such as those given in Book I, this type of data entry is quite adequate. When more extensive programs are involved however (with a length of, say, several hundred bytes), typing errors can occur frequently and correcting them can become a very tedious procedure. What do you do, for instance, if a few instructions have been omitted from the middle of the program? Normally, this would mean having to re-enter a large part of the program from the correction onwards. What a waste of time and effort! Redundant instructions can be removed by replacing them with NOP instructions (op-code EA), which is making very poor use of the computer's memory space. Think how irritating it would be, if, when entering a long program, you run out of memory!

Fortunately, this can now be avoided with the aid of one of the JC's greatest assets: the EDITOR. This enables bytes to be inserted or deleted anywhere within the memory area. When bytes are to be inserted, the computer makes room for them by moving a data block further down in the program. When bytes are to be deleted, the data block is moved up to 'close the gap'. Block transfers and their respective programs were mentioned in Book I during the description of indexed and indirect addressing.

In short, the editor can save a lot of time and trouble where program correction is concerned. Exactly how simple it is to enter and correct programs will be seen later on in this chapter.

## The Assembler

Another indispensable aid towards simple, error-free, high-speed program entry is the assembler. As you will remember from Book I, branch instructions are often used. Before the displacement value of a branch instruction can be calculated, it is important to know the start and destination addresses. It is only then that the monitor routine BRANCH (start address 1FD5) can calculate the displacement. The same applies to the instructions JSR and JMP – the true jump address must be known. Thus, before a program can be keyed in, you must be familiar with the absolute addresses of the branch and jump instructions. Obtaining the correct addresses numerically involves a great deal of writing. A more efficient method is to let the assembler in the monitor program take care of it.

It fulfils the task quickly and simply and, above all, accurately. All that is required is to depress the GO key and the computer will calculate all the displacement values, absolute addresses of jump instructions and the subroutine start addresses by itself. Incredible as it may seem, the JC takes this painstaking job off the programmer's hands and completes it in a

matter of seconds. What is more, the assembler, the editor and several other routines only occupy 1 k of EPROM! This is possible as the 6502 microprocessor happens to have highly effective and powerful instructions and numerous address possibilities.

The time has now come to take a closer look at the editor and assembler and find out exactly how to use them.

### Editing and assembly from start to finish

Before the editor and assembler can be put into operation, we need to develop a program to be run on the Junior Computer. The program should perform the following:

- convert an 8-bit binary number into decimal
- display the binary number in hexadecimal form next to the decimal figure
- use the keyboard to enter the hexadecimal number
- use monitor routines to scan the six digit display.

The object of the exercise is to write a program as quickly as possible, free of errors and which meets the above requirements. It is best to start by constructing an algorithm which outlines the conversion of an 8-bit binary number ( $00 \dots FF$ ) into decimal ( $010 \dots 25510$ ). This is shown in figure 1. As can be seen, the algorithm is expressed in words. It should be mentioned at this stage that it is always a good idea to write down a complicated program in words first! Then a rough flow chart can be produced leading to the actual program.

During the binary to decimal conversion program, BINDEC, a counter is reset to zero and the hexadecimal number to be converted is stored inside a buffer. The actual procedure is as follows:

1. The value  $0A$  ( $=1010$ ) is repeatedly subtracted from the hexadecimal number in the buffer until the buffer contents become negative. After each subtraction the counter is incremented by one. This means that in the end the counter will show how many subtractions were carried out before the result became negative.
2. In order to represent a negative 8-bit number in an 8-bit processor, a 16-bit buffer is required consisting of two memory locations.
3. If the most significant digit of the 16-bit number is a 'one', the result of the subtraction was negative and the program branches to the section labelled UNITS. Here the processor will add  $0A$  to the negative buffer contents. The result is a figure which can assume a value between  $0 \dots 9$  and is the number of units of the decimal answer.
4. The contents of the counter are stored in the 16-bit buffer, after which the counter is once again reset. The processor will now continue to subtract  $0A$  from the buffer contents until a negative result is obtained. The counter will again indicate the number of subtractions that were carried out.
5. When  $0A$  is then added to the negative buffer contents, a figure is obtained which can assume any value between  $0 \dots 9$  and which represents the tens of the decimal answer.
6. The counter will now contain a figure with a value from  $0 \dots 2$ , representing the hundreds of the decimal answer.

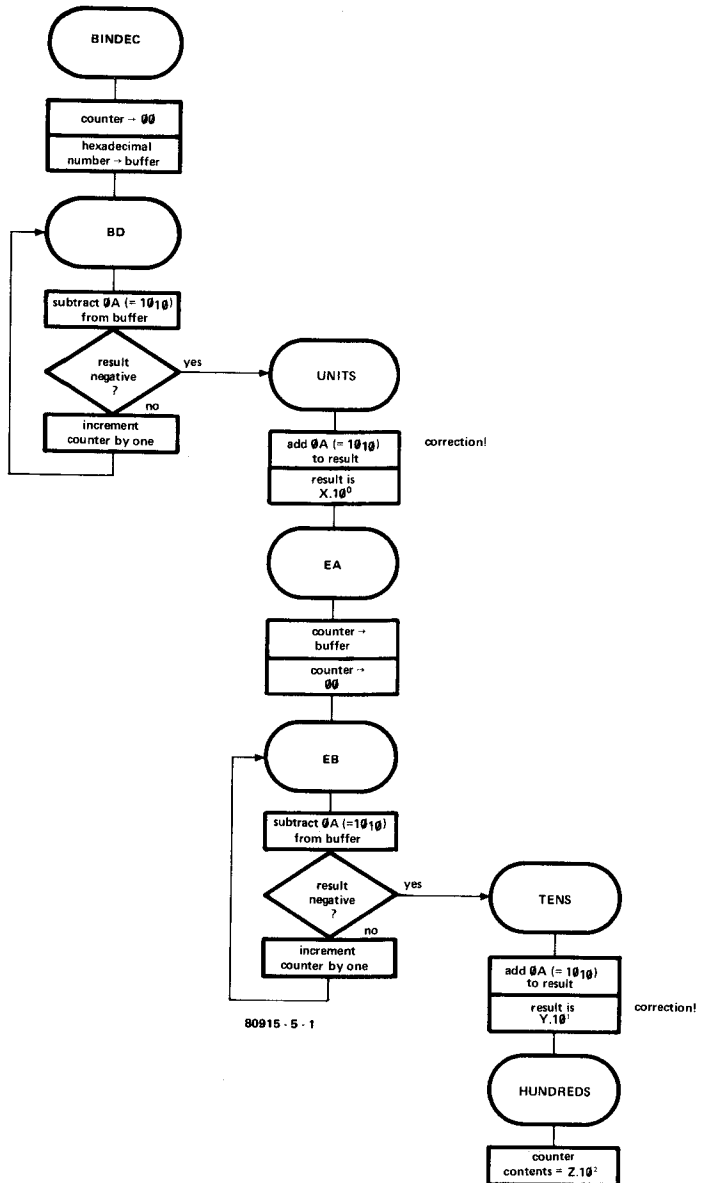


Figure 1. Before we can work with the editor and the assembler, let us develop a program that converts a hexadecimal value into a decimal. The editor and the assembler will be seen to intervene as soon as the program is written in memory. The flow chart contains a conversion algorithm.

This is all very well in theory, but of course only a practical example can prove that the algorithm really works. Let us therefore convert the hexadecimal value 0091 into decimal in the same way as the program given in figure 1.

number buffer:	0091	counter:	00
	<u>  0A</u>		
number buffer:	0087	counter:	01
	<u>  0A</u>		
number buffer:	007D	counter:	02
	<u>  0A</u>		
number buffer:	0073	counter:	03
	<u>  0A</u>		
number buffer:	0069	counter:	04
	<u>  0A</u>		
number buffer:	005F	counter:	05
	<u>  0A</u>		
number buffer:	0055	counter:	06
	<u>  0A</u>		
number buffer:	004B	counter:	07
	<u>  0A</u>		
number buffer:	0041	counter:	08
	<u>  0A</u>		
number buffer:	0037	counter:	09
	<u>  0A</u>		
number buffer:	002D	counter:	0A
	<u>  0A</u>		
number buffer:	0023	counter:	0B
	<u>  0A</u>		
number buffer:	0019	counter:	0C
	<u>  0A</u>		
number buffer:	000F	counter:	0D
	<u>  0A</u>		
number buffer:	0005	counter:	0E
	<u>  0A</u>		
negative result	(- 0005)	counter unchanged	

When 0A is added to the negative number contained in the buffer the result will be 5 once more. This will be the number of units contained in the decimal answer. Thus,  $91_{16} = ZY 51_{10}$ .

The contents of the counter (0E) are now transferred to the number

buffer and the counter is once more reset. The new buffer contents are then worked on to provide the next number:

number buffer:  $000E$  counter:  $00$   
 $- \quad 0A$

number buffer:  $0004$  counter:  $01$   
 $- \quad 0A$

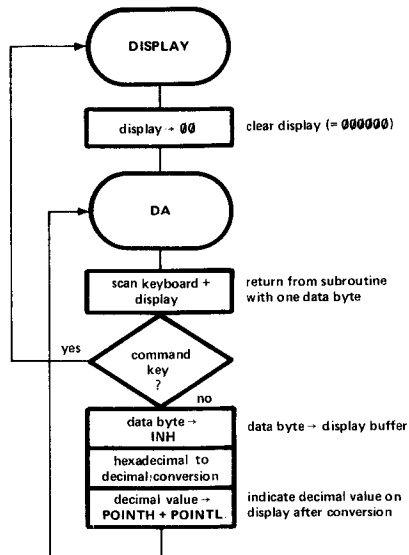
negative result ( $- 0006$ ) counter unchanged

When  $0A$  is added to the negative number contained in the buffer the result will be 4 once more. This will be the number of tens contained in the decimal result. Thus,  $9116 = Z 4510$ .

The counter contains 1, which represents the number of hundreds in the decimal result. The complete answer is therefore:  $9116 = 14510$ . The methods used to present this figure to the computer's display were described in Book I.

Now the algorithm for the binary to decimal conversion program is ready and the rough flow chart can be drawn up, as shown in figure 2.

Firstly, the display buffer is cleared. The computer then scans the keyboard and the display. The subroutine which the main program requires was also dealt with in Book I, subroutine GETBYT (start address 1D6F). If a command key is depressed, the processor will reset the N-flag before returning to the main program. In this way, the display can be cleared whenever a command key (it doesn't matter which) is depressed.

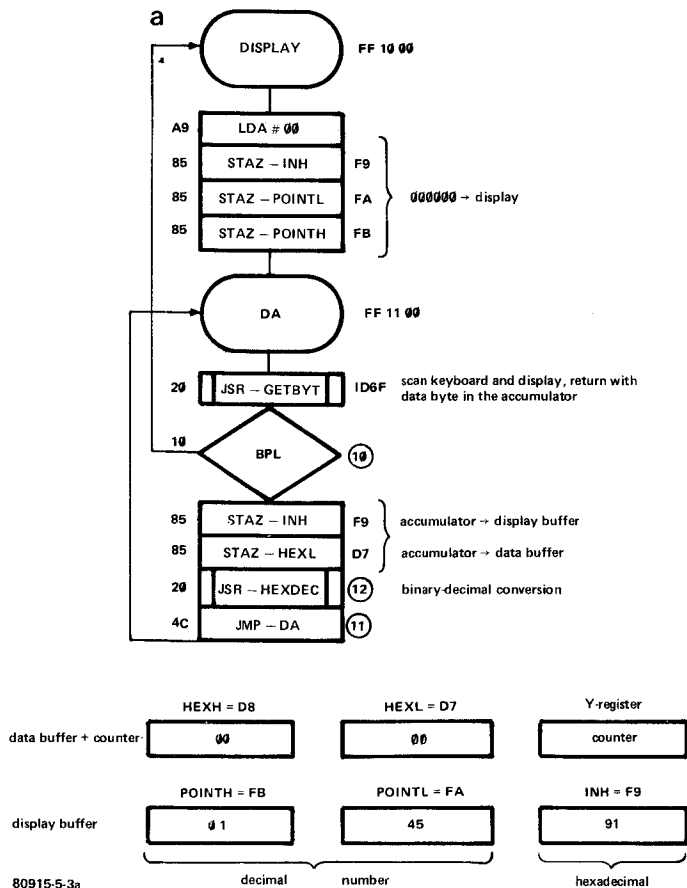


80915 - 5 - 2

**Figure 2. The general flow chart required to control the display and scan the keyboard during the hexadecimal to decimal conversion.**

Once the two data keys have been depressed, the microprocessor returns from the GETBYT subroutine with the entered data byte in the accumulator. This data byte is then transferred to the display buffer INH. It is at this point that the binary-to-decimal conversion takes place according to the algorithm drawn up previously. The decimal number obtained is also transferred to the display buffers POINTH and POINTL.

Once again, the computer will scan the keyboard and display the hexadecimal and the decimal numbers via the GETBYT subroutine. The full details of the flow chart are given in figure 3. The display buffer consists



**Figure 3a.** The detailed version of the flow chart in figure 2. The display buffer INH contains the hexadecimal value that is to be converted. After the conversion, the decimal number will be stored in display buffers POINTH and POINTL. Memory locations HEXH and HEXL and the Y index register act as temporary memory banks.

of addresses 00F9...00FB. The data or number buffer occupies addresses 00D7 and 00D8. The memory locations reserved for the program can be labelled as follows:

```
INH      * $ 00F9
POINTL  * $ 00FA   display buffers
POINTH  * $ 00FB
HEXL    * $ 00D7
HEXH    * $ 00D8   data buffers
GETBYT  * $ 1D6F   monitor subroutine
```

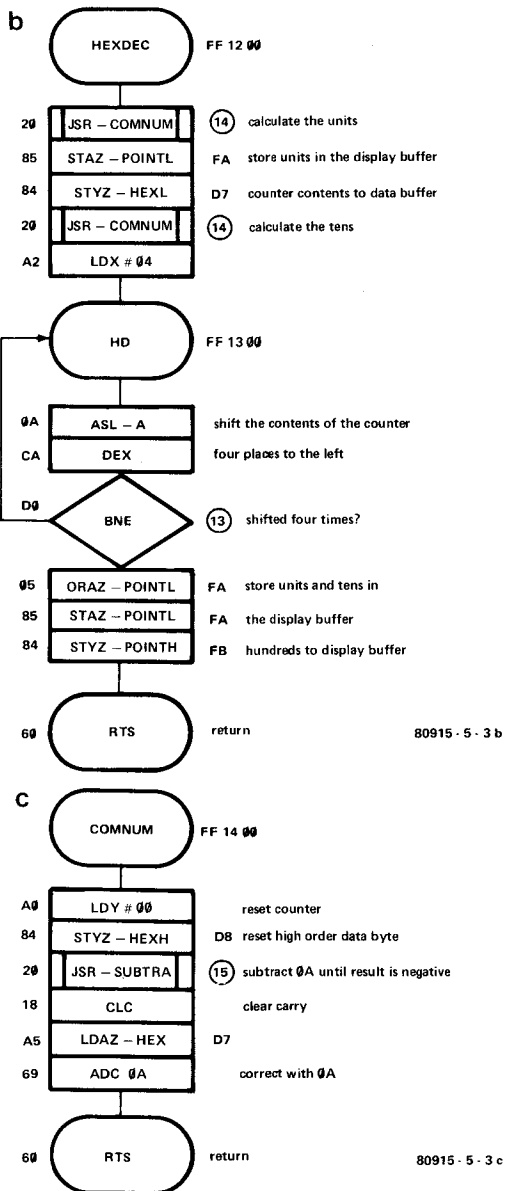
During the program the Y register will be used as a counter instead of a memory location. As can be gathered from figure 3, the main program DISPLAY contains the two subroutines GETBYT and HEXDEC. Both are different in that the start address of GETBYT is known, whereas that of HEXDEC is not. This is no problem, as from now on the computer will be able to calculate start addresses by itself. Thus, there is no need to assign a start address to HEXDEC. In addition, the main program includes a jump and a branch instruction. The destination address for the jump instruction and the displacement value for the branch instruction are also calculated by the computer, so that there is no need for the programmer to worry about them either.

The subroutine HEXDEC, which is the practical version of the algorithm we developed before, incorporates two subroutines, COMMUN and SUBTRA, as well as jump and branch instructions. Here again, we needn't worry about which addresses belong to which subroutines and what displacement values are required for the branch instructions. These tiresome, time-consuming calculations can all be left to the computer to deal with, which, after all, cannot go wrong!

The section of the monitor program which calculates the displacements of branch instructions is called the assembler. It is started in the usual manner, at address 1F51, by depressing the GO key.

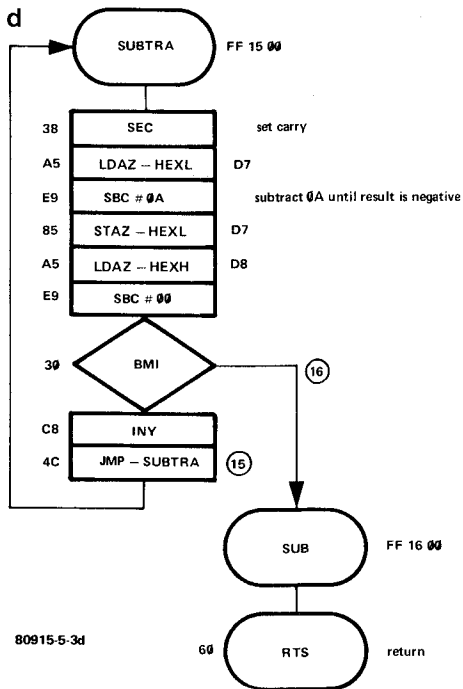
For the assembler to function properly, the computer requires a tool to prepare the programs so that the assembler can 'digest' them. This tool is in fact the editor.

As we mentioned before, the computer calculates addresses and displacements by itself. The question is: how does it know where to jump or branch, if the programmer does not enter the start or destination addresses? Well, the programmer simply enters *symbolic addresses* into the computer. From now on it is enough to enter JSR-SUBTRA, instead of JSR-023B, for they both mean the same thing as far as the JC is concerned! Thus, the assembler's task consists of allocating the absolute address 023B to the symbolic address SUBTRA. However, the word SUBTRA cannot be entered into the JC, as the only means of entering data is the hexadecimal keyboard — there is no alphanumeric one at the present time. This means that a label may only consist of hexadecimal characters which could, if we are not careful, be interpreted as op-codes or operands. The computer must therefore be able to distinguish between op-codes and labels. What in fact is the difference between them? To find out, look at the appendix to Book I where the op-codes of the 6502 microprocessor have been compiled in hexadecimal sequence. As the table shows, not all the hexadecimal



Figures 3b and 3c. The actual hexadecimal to decimal conversion takes place in the course of the HEXDEC subroutine. The COMNUM routine first calculates the units, then the tens and finally the hundreds.





**Figure 3d.** During the subroutine SUBTRA the processor carries out a 16 bit subtraction. The hexadecimal number 0A is subtracted several times until it gives a negative result. Here the Y index register acts as a subtraction counter.

numbers from 00 . . . FF have been allocated op-codes. There are several gaps. Even the hexadecimal number FF has no op-code. Let us give it a pseudo op-code, in other words, an op-code unknown to the microprocessor. Now the hexadecimal number FF can be used as a label. We could have chosen any other unused number from the table, such as 04, D3 or F7, but FF happens to be the easiest to enter.

The label SUBTRA still has a few other aspects worth considering. SUBTRA is the program departure point, or to be more exact, the symbolic start address of a subroutine. The pass-word to call up the subroutine is JSR-SUBTRA (jump to subroutine SUBTRA). Since only hexadecimal figures are used in the editor, subroutine SUBTRA will have to have a number. This can be any number from 00 . . . FF. Up to 256 such labels can be used for identification purposes in a program.

It is now high time to put the above into practice with the aid of a few examples, for from now on program entry will always involve the editor and the assembler contained in the monitor program.

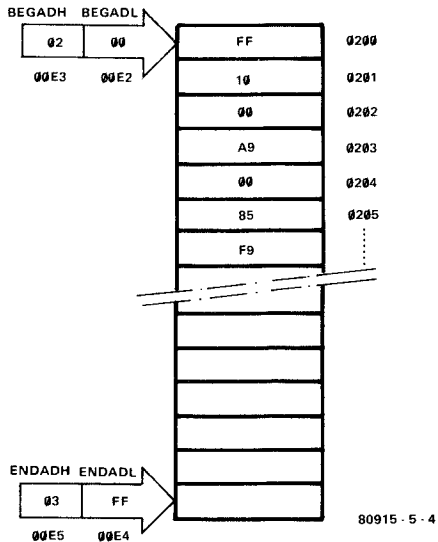


Figure 4. This is how the editor stores the initial instructions pertaining to the DISPLAY program (figure 3a) in the Junior Computer's memory. BEGAD and ENDAD are pointers and define the work space.

## The Editor

### 1. Structure of a label:

FF	15	00	example
FF	XX	00	general pattern

label indicator    label number    label limiter

As can be seen a label is three bytes long. In order to enter a label into the JC, six keys will have to be depressed. Figure 3 shows that the label SUBTRA is replaced by FF 15 00. Once the editor has been activated, labels like this can be entered straight into the computer. An ASCII keyboard (typewriter) will not be necessary.

Other *valid* entries include:

keyboard	display	
FF 37 00	FF 37 00	label number 37
FF FA 00	FF FA 00	label number FA
FF 00 00	FF 00 00	label number 00

*Non-valid* label entries include:

keyboard	display	
FF 21	FF 21 XX	limiter missing
FF 56 FF	FF 56 FF	wrong limiter
FF	FF 41 00	although the correct limiter will appear in the display after the label entry, it must be over-written with 00.

Once the entire label entry has been completed, the display buffer is copied into the memory.

### 2. Structure of a JSR instruction:

20	14	00	example
20	XX	00	general pattern

op-code of JSR      subroutine number      limiter

As usual, the jump to subroutine instruction is three bytes long. Now let us return to figure 3. During the HEXDEC subroutine, COMNUM is called twice. Usually, the instruction would be written as JSR-COMNUM. Subroutine COMNUM has been given the label number 14, but any other number would have been equally suitable. To the computer, 20 14 00 means: jump to subroutine number 14, or to put it more precisely: jump to the subroutine which starts with the label number 14. As you will remember, FF 14 00 and COMNUM are the same in the flow chart.

Other *valid* entries for a JSR instruction include:

keyboard	display		
20 39 00	20 39 00	jump to subroutine 39	
20 9A 00	20 9A 00	jump to subroutine 9A	
20 20 00	20 20 00	jump to subroutine 20	

Non-valid JSR entries include:

keyboard	display	
20 11	20 11 XX	limiter missing
20 71 9F	20 71 9F	wrong limiter
20	20 28 00	although the required subroutine number and the correct limiter appear in the display, this is a <i>coincidence</i> and the entire instruction must be entered. Only then will the display buffer be copied into the memory of the computer.

### 3. Structure of a jump instruction:

4C	11	00	example
4C	XX	00	general pattern

op-code of JMP instruction      label number      limiter

The jump instruction is also three bytes long. Towards the end of the main routine DISPLAY, a jump instruction can be seen, JMP-DA, or jump to label DA. The JC interprets 4C 11 00 as: jump to label number 11. Again, labels DA and FF 11 00 are the same.

Other *valid* entries for a jump instruction include:

keyboard	display	
4C 77 00	4C 77 00	jump to label number 77
4C 29 00	4C 29 00	jump to label number 29
4C 00 00	4C 00 00	jump to label number 00

*Non-valid* jump entries include:

keyboard	display	
4C 32	4C 32 XX	limiter missing
4C 08 AA	4C 08 AA	wrong limiter
4C	4C 24 00	again, the complete instruction must be entered before the computer will copy it into the memory

The hexadecimal numbers following the JSR and JMP instructions do not therefore constitute absolute addresses, but label numbers which the



6F. The label number 6F does not appear in the program (figure 3) and so the assembler will not assign an address to the JSR-GETBYT instruction but will ignore it.

7. Entry of the remaining instructions:

Taking the load instruction (LDA) as an example:

keyboard	display	mnemonic	address 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	pre-indexed indirect
B 1 F A	B1 FA	LDA-(FA),Y	post-indexed indirect
B D 0 0 0 2	BD 00 02	LDA-0200,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

The validity of the above also holds for the remainder of the 6502 instructions. The computer will calculate the length of each instruction by itself. For this it will use the monitor subroutine OPLEN which is almost identical to LENACC described in Book I. It enables the computer to control the display irrespective of the instruction length. As you know, instructions that use implied addressing are only one byte long. For this reason only the op-code field of the display will be lit when these instructions are entered. The rest of the display will be 'blanked'.

Instructions belonging to the above category have *no symbolic addresses* allocated to them. That is to say, the programmer must define the address locations from which data is to be read out or to which data is to be written, before the program can be entered.

8. Definition of program memory space:

Before the editor can be activated, the memory space required for storing the program must be defined. The boundaries are determined by two address pointers located on page zero. The pointer BEGADH, BEGADL (= BEGin ADDRESS High, BEGin ADDRESS Low) is stored in locations:

```
BEGADL * $ 00 E2
BEGADH * $ 00 E3
```

The pointer ENDADH, ENDADL (= END ADDRESS High, END ADDRESS Low) is stored in locations:

```
ENDADL * $ 00 E4
ENDADH * $ 00 E5
```

BEGAD therefore always refers to the initial address and ENDAD to the final address of the memory area in which the program is to be stored.

Two consecutive pages of memory, page 2 and page 3, are available in the standard version of the Junior Computer. This amounts to 512 bytes, which should be ample for most programs. It is therefore a good idea to define this large area of memory space by means of the address pointers before entering a program:

```
BEGADH, BEGADL = 02 00 and
ENDADH, ENDADL = 03 FF
```

The relevant data can be entered into these pointers via the keyboard as follows:

AD 00E2

DA 00      ADL of the BEGAD pointer  
+ 02      ADH of the BEGAD pointer  
+ FF      ADL of the ENDAD pointer  
+ 03      ADH of the ENDAD pointer

The pointers are located on page zero and the memory area in which the program is to be run can be defined as 0200 . . . 03FF. Other areas of memory can also be 'fenced off' in the same manner.

Page 0 \$ 0000 . . . 00E0

Page 1 \$ 0100 . . . 01F2

Page 1A \$ 1A00 . . . 1A79

There is only a limited amount of memory space available on these pages as they are partly occupied by the monitor program and the stack. If the limits of these areas are exceeded, the JC will report this as an error: EEEEE.

#### 9. How to start the editor:

Once the required memory space has been defined, the editor can be put into operation. The start address is 1CB5 and it can be initiated by depressing the following keys:

AD 1CB5

GO

The op-code portion of the display will now show 77 and the operand area of the display will remain blank. This is the computer's way of telling you that the editor is functioning and is ready to store any instructions entered inside the pre-determined operational memory area (work space).

#### 10. Editor command keys:

As soon as the editor is activated, the functions DA, AD, +, PC and GO are no longer valid. From now on, the second heading associated with these keys (SEARCH, INPUT, INSERT, DELETE and SKIP) will become effective. These keys enable instructions to be entered into the computer (INSERT, INPUT), to be deleted (DELETE) or even to be searched for (SEARCH). In addition, it is possible to jump from instruction to instruction via the SKIP command. Let us now examine each command in turn.

#### **INSERT**

*The INSERT command allows a new instruction to be inserted before the one indicated on the display.*

Whenever the pseudo-instruction 77 appears on the display the INSERT key must be depressed *without fail* for the data to be entered as instructions. The INSERT key is therefore the first key to depress if a label or an instruction is to be entered into the computer, once the editor has been activated.

#### **INPUT**

*The INPUT command enables a new instruction to be entered after the one indicated on the display.*

The programmer can select between the INPUT and INSERT commands at all times. Whichever key was the last to be operated will then be valid. As far as both are concerned, it should be noted that when they are used to

enter labels or instructions, the following data will be shifted into the display buffer one byte at a time. Only when the entire instruction has been entered into the computer will it be copied from the display buffer into the pre-determined memory area. This means that if an error occurs when entering an instruction, either the INSERT key of the INPUT key must be depressed several times before the error can be corrected. INSERT will place the fresh instruction in front of the one previously displayed, whereas INPUT will place it behind that instruction. Thus, each key will retain its proper function. The result is a simple, yet highly effective procedure to enter and edit programs.

### **DELETE**

*The DELETE command enables the displayed instruction to be erased from memory.*

The computer will fill up the memory locations which have now become available. This is done by transferring (shifting) the data block immediately following the erased instruction upwards by the number of bytes deleted. It ensures that no 'gaps' occur in the program once an instruction is deleted. In other words, the instruction following the one just deleted will appear on the display as soon as the DELETE key is pressed. Again, by repeatedly depressing the INSERT or INPUT keys, new instructions can be entered before or after the instruction shown.

### **SEARCH**

*The SEARCH command enables the computer to look for (and find) any double-byte pattern in the memory area.*

If, for example, we depress the keys SEARCH FF 11, the computer will search for label number 11 and show it in the display. The SEARCH command also manipulates the display pointer CURAD, also located on page zero, which always points to the op-code currently on display. This pointer is held in the following memory locations:

CURADL \* \$ 00 E6

CURADH \* \$ 00 E7

Initially, the SEARCH command leads the display pointer CURAD to the start address of the previously defined memory area. Since each instruction has a certain, specific length, calculated by the computer, CURAD is moved from op-code to op-code (and not from memory location to memory location). Through a set of simple comparisons, the computer can establish whether or not the required bit-pattern has been found. Labels are not the only things that can be searched for in this manner, any other type of instruction can also be located; for example:

SEARCH A9 00,

SEARCH 4C 11 (figure 3) or

SEARCH 69 0A (figure 3).

Note that if the same instruction occurs more than once in a program, the SEARCH command will only locate the first of them (the instruction nearest the start of the memory area). In other words, it is unable to discover two identical instructions in a row!

**SKIP**

The SKIP command enables the computer to jump from instruction to instruction.

This command is used to run through a program step by step, so that it can be checked for any errors in a matter of seconds.

Note: The INPUT command is essentially a combination of the SKIP and INSERT commands. This creates the following analogies:

SKIP INSERT X X = INPUT X X (1 byte)  
 SKIP INSERT X X X X = INPUT X X X X (2 bytes)  
 SKIP INSERT X X X X X X = INPUT X X X X X X (3 bytes)

11. Cold start entry/Warm start entry:

There are two start addresses for the editor:

Cold start entry = 1CB5 and

Warm start entry = 1CCA.

If the editor is started at the former address, it will add (and initialise) a few pointers to page zero outside the memory area chosen by the programmer. Thus, before a program can be entered for the first time, the editor must be started at address 1CB5.

Warm start entry, on the other hand, makes it possible for the programmer, to return to the editor after depressing the RST key. In this case, the pointers on page zero will not be initialised. After starting the editor at address 1CCA, the command keys INSERT, INPUT, DELETE, SEARCH and SKIP will operate in their usual manner.

12. How to enter the program shown in figure 3:

press key:	display:	comments:
RST	XX XX XX	} initialisation
AD 0 0 E 2	00 E2 XX	
DA 0 0	00 E2 00	} set the BEGAD pointer
+ 0 2	00 E3 02	
+ F F	00 E4 FF	
+ 0 3	00 E5 03	
AD 1 C B 5	1C B5 20	} set the ENDDAD pointer
GO	77	
INSERT F F 1 0 0 0	FF 10 00	start the editor (Cold start entry)
INPUT A 9 0 0	A9 00	enter label number 10 (DISPLAY)
INPUT 8 5 F 9	85 F9	LDA # 00
INPUT 8 5 F A	85 FA	STAZ-INH
INPUT 8 5 F B	85 FB	STAZ-POINTL
INPUT F F 1 1 0 0	FF 11 00	STAZ-POINTH
INPUT 2 0 6 F 1 D	20 6F 1D	label number 11 (DA)
INPUT 1 0 1 0	10 10	JSR-GETBYT (6F is not a label number!)
INPUT 8 5 E	85 XX	BPL to label number 10
		error! Instruction is still to be entered into memory
INPUT 8 5 F 9	85 F9	STAZ-INH
INPUT 8 5 D 7	85 D7	STAZ-HEXL
INPUT 2 0 1 2 0 0	20 12 00	JSR-HEXDEC (label number 12 is HEXDEC)
INPUT 4 C 1 1 0 0	4C 11 00	JMP-DA (label number 11 is DA)
INPUT F F 1 2 0 0	FF 12 00	label number 12 (HEXDEC)
INSERT		error! INSERT instead of INPUT



INPUT	2 0 1 4 0 0	20 14 00	JSR-COMNUM (label number 14 is COMNUM)
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 4 D 7	84 D7	STYZ-HEXL
INPUT	2 0 1 4 0 0	20 14 00	JSR-COMNUM (label number 14 is COMNUM)
INPUT	A 2 0 4	A2 04	LDX # 04
INPUT	F F 1 3 0 0	FF 13 00	label number 13 (HD)
INPUT	0 A C A	0A EEEEE	press command key, otherwise error! ASL-A
INPUT	C A	CA	DEX
INPUT	D 0 1 3	D0 13	BNE to label number 13
INPUT	0 5 F A	05 FA	ORAZ-POINTL
INPUT	8 5 F A	85 FA	STAZ-POINTL
INPUT	8 4 F B	84 FB	STYZ-POINTH
INPUT	6 0	60	RTS
SEARCH	F F 1 3	FF 13 00	search for label number 13
SKIP		0A	
SKIP		CA	
SKIP		D0 13	
SKIP		05 FA	
SKIP		85 FA	
SKIP		84 FB	
SKIP		60	
SKIP		77	77 means: depress the INSERT key!!!!
INSERT	F F 1 4 0 0	FF 14 00	label number 14 (COMNUM)
INPUT	8 4 D 8	84 D8	STYZ-HEXH

Note: The instruction LDY # 00 was inadvertently forgotten. It can be inserted quite easily with the aid of the INSERT command. Thus:

INSERT	A 0 0 0	A0 00	LDY # 00
SKIP		84 D8	STYZ-HEXH

The instruction LDY # 00 has now been inserted in front of STYZ-HEXH. This involved a fair amount of work for the computer, while the programmer could sit back and watch. Now the INPUT command will enable us to continue, for a new instruction is to be entered after 84 D8. Thus:

INPUT	2 0 1 5 0 0	20 15 00	JSR-SUBTRA (label number 15 is SUBTRA)
INPUT	1 8	18	CLC
INPUT	A 5 D 7	A5 D7	LDAZ-HEXL
INPUT	6 9 0 A	69 0A	ADC # 0A
INPUT	6 0	60	RTS
INPUT	F F 1 5 0 0	FF 15 00	label number 15 (SUBTRA)
INPUT	3 8	38	SEC
INPUT	A 5 D 7	A5 D7	LDAZ-HEXL
INPUT	E 9 0 A	E9 0A	SBC # 0A
INPUT	8 5 D 7	85 D7	STAZ-HEXL
INPUT	A 5 D 8	A5 D8	LDAZ-HEXH
INPUT	E 9 0 0	E9 00	SBC # 00
INPUT	3 0 1 6	30 16	BMI to label number 16
INPUT	C 8	C8	INY
INPUT	4 C 1 5 0 0	4C 15 00	JMP-SUBTRA (label number 15 is SUBTRA)
INPUT	F F 1 6 0 0	FF 16 00	label number 16 (SUB)
INPUT	6 0	60	RTS

The complete program has now been entered into the Junior Computer. It is advisable to check that everything has been entered correctly. This can be done as follows:

SEARCH F F 1 0

SKIP

SKIP

.

.

.

SKIP

Only when the entered program corresponds to the flow chart can the assembler be started.

## The Assembler

We have now entered a program into the computer with the aid of the editor, a program containing symbolic addresses. These consist of label numbers, subroutine numbers or labels to which the computer is to jump or branch. Such a program is not able to run on JC, as in its present form the microprocessor is unable to interpret the instructions. It is up to the assembler to 'shape' the program in such a way that the microprocessor can understand it. The assembler's task consists of:

- \* removing all the labels from the program
- \* assigning the true absolute addresses to the JSR instructions so that they are unambiguous
- \* treating the jump instructions likewise
- \* calculating the correct displacement values for branch instructions
- \* enabling the program to be stored in the programmable memory, after assembly, in an acceptable form for the 6502 microprocessor.

### How is the program stored in memory before assembly?

The start address of the program was chosen to be 0200. Figure 4 shows how the editor stored the program in the computer's memory. Since three memory locations were reserved for each label, this program is considerably longer than the assembled version will be. The start address of the assembler is 1F51:

RST           call the monitor program

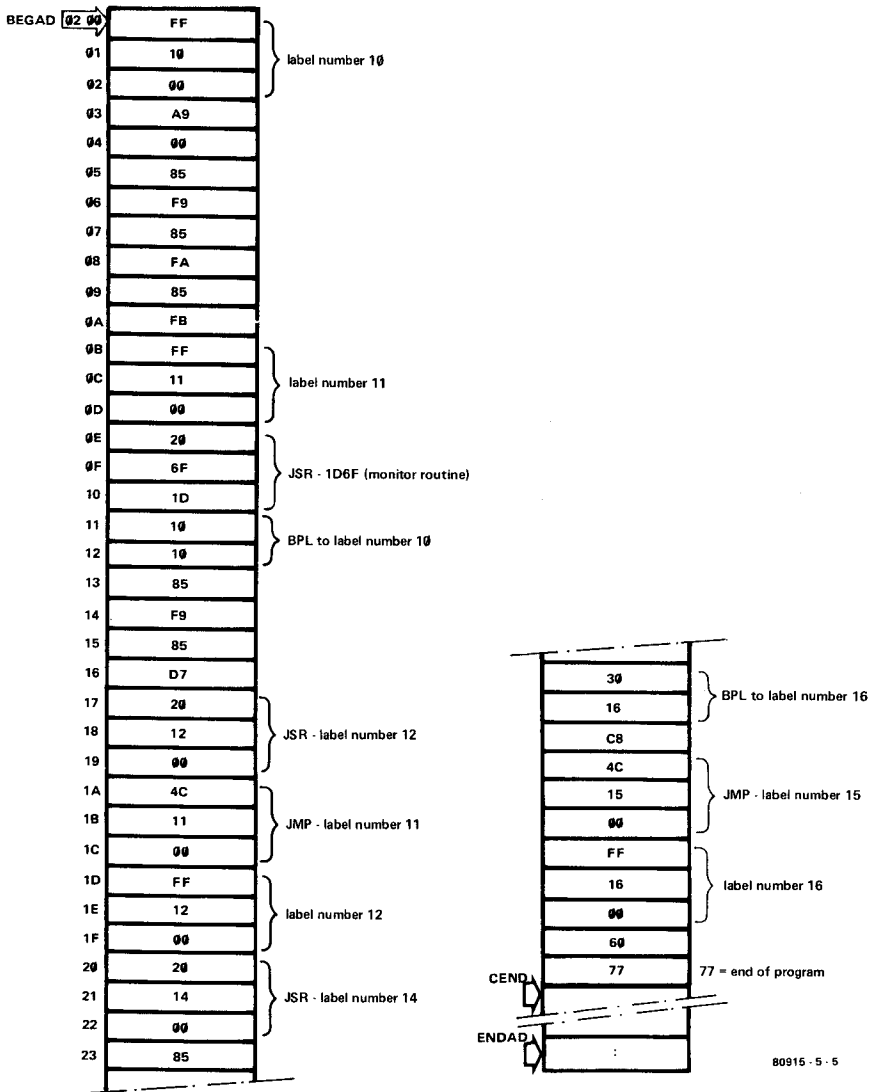
AD 1 F 5 1    enter start address

GO           start the assembler

After the GO key is depressed, the display will be blanked for a fraction of a second . . . all that is needed for the computer to assemble the program. If the program was much longer, the display may remain blank for several seconds. As soon as the program has been assembled, the computer will return to the monitor program. The keys AD, DA, +, PC and GO will now be valid once more.

### How is the program stored in memory after assembly?

The Junior Computer features a Two Pass Assembler. This means that the program is not assembled in one go, but in two stages, or 'passes'.



**Figure 5.** The addressable work space starts at BEGAD and ends at ENDAD. The memory space between the two pointers contains the program starting in this particular example from label FF 10 00 and ending with the EOF character 77. This is called a file. It contains all the instructions that are used in the program, together with their addresses and labels which have the symbolic op-code FF. The Junior Computer cannot 'digest' the program in this form, as both the absolute addresses for jump instructions and the displacements for branch instructions are as yet unknown. The 6502 CPU will not be able to handle the edited data until the program has been assembled.

## Pass one

The computer reads the entered program. At this stage it is only interested in the labels written by the programmer and so will ignore all other instructions. If a label is encountered (featuring the characters FF), the label number and its address will be stored in a symbol stack. The latter acts simply as a look-up table and is prepared by the computer without any help from the programmer.

Once the label number and its address have been 'saved' in the symbol stack, the computer will delete the label from the program. This is done by shifting the entire program up by three bytes — the length of the label. The label is overwritten by the instruction(s) immediately following and therefore disappears from the program. The same procedure is repeated until the processor has worked through the whole program from start to finish.

In the example used, the start address was 0200, so the computer will start here. The search continues until another label is found. This will then be added to the symbol stack like its predecessor, along with its address. Eventually, all the labels will have been removed from the program. Everything the computer needs to know about the labels is now being stored inside the symbol stack. This is a 256 byte software register capable of storing up to a maximum of  $256/3 = 85$  labels. This should be more than enough room to assemble programs in sections.

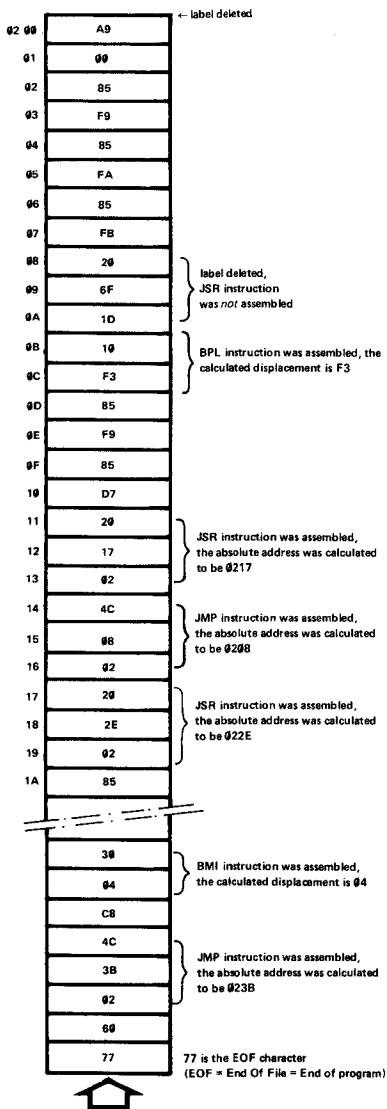
## Pass two

Now the assembler will run through the entire program once more, starting at address 0200. As all the labels have already been deleted, the assembler will concentrate on the following instructions *only*:

- \* JMP-
- \* JSR-
- \* branch instructions such as BPL, BMI, BEQ etc.

These instructions are as yet succeeded by label numbers. When the JC encounters one of the jump instructions, it will look to see which label number follows. Every label number has its own absolute address, which the computer can fetch from the symbol stack. How does it find the correct address in the symbol stack? During the first stage of assembly, remember, the computer saved the label number and its absolute address in the stack. Now the computer will compare the label number behind the jump instruction being assembled with those in the symbol stack. The corresponding absolute address is then inserted after the instruction thereby overwriting the label number and the limiter. The whole procedure is repeated until all the JMP and JSR instructions have been assembled.

This leaves the displacement values for the branch instructions. The next step for the computer, therefore, is to trace all the branch instructions contained in the program. These are still followed by the label numbers to which the branch is supposed to lead. Since the computer knows the address of each branch instruction and since the destination address is stored in the symbol stack, the displacement value can be calculated



During assembly the program was shifted up from below. This caused the symbolic addresses and the label numbers to be deleted.

80915-5-6

**Figure 6. After assembly, all the labels have disappeared from the file, which is why the program looks shorter. The instruction op-codes are now followed by their corresponding absolute addresses. Similarly, displacements have been assigned to the branch instructions. Now the program is acceptable to the 6502 CPU.**

without any difficulty:

destination — source + 2 = displacement,

which can be translated as: the address location to be branched to, minus the address location where the branch instruction is situated, plus two equals the displacement value. The program given in figure 3 is shown in its assembled form in figure 5.

We have now covered the operation of the assembler. It should be added that the editor can also be used to run through an assembled program. The following should then be taken into account:

\* the editor should *only* be started by way of a warm start entry (address 1CCA)

\* the pointers on page zero must be set manually:

BEGAD = CURAD

ENDAD = CEND (CEND = current end address)

Based on the example given in figure 3, the following keys are to be depressed for the editor to be used on an *already assembled program*.

```
AD 0 0 E 6
DA 0 0 }      set CURAD pointer to 0200
+ 0 2 }
+ FF }      set CEND pointer to 03FF
+ 0 3 }
AD 1 C C A    warm start entry
GO           start the editor
```

This procedure enables the SKIP key to be used to run through an assembled program. The SEARCH command gives the programmer the additional possibility to search for certain instructions inside the assembled program. The INSERT, INPUT and DELETE keys should only be used when entry or deletion of instructions will leave any jump or branch instruction absolute addresses and length — as calculated by the computer — unimpaired.

### Important addresses for the editor and assembler:

#### 1. Editor

Cold start entry : \$ 1CB5

Warm start entry : \$ 1CCA

Pointers belonging to the editor:

BEGADL \* \$ 00E2 } begin address pointer

BEGADH \* \$ 00E3 }

ENDADL \* \$ 00E4 } end address pointer

ENDADH \* \$ 00E5 }

CURADL \* \$ 00E6 } current address pointer

CURADH \* \$ 00E7 }

CENDL \* \$ 00E8 } current end address pointer

CENDH \* \$ 00E9 }

#### 2. Assembler:

Start address: \$ 1F51

#### 3. Further possibilities for the editor and assembler:

The ST key on the computer's keyboard is linked to the NMI circuit. The NMI-vector can be programmed so that either the editor or the

assembler can be started by depressing the ST key. The NMI-vector is stored in address locations:

NMIL \* \$ 1A7A low-order address byte

NMIH \* \$ 1A7B high-order address byte

By loading the start addresses into these locations the programmer can call up either the editor or the assembler with the ST key:

AD 1 A 7 A cold start entry via ST key

DA B 5

+ 1 C

or

AD 1 A 7 A warm start entry via ST key

DA C A

+ 1 C

or

AD 1 A 7 A start the assembler via ST key

DA 5 1

+ 1 F

We have now reached the point where we are able to operate the editor and assembler contained in the monitor program. These special features make it possible to enter and alter programs quickly and efficiently. This saves the programmer a lot of work — which is what computers are designed for! The editor and the assembler enable programs to be stored in memory in one 'lump', without gaps. This is very important when only a limited amount of memory space is available.

# The Peripheral Interface Adapter or PIA

The Junior Computer possesses a peripheral interface adapter which, like the CPU, is contained in a 40-pin IC. This multi-function device, called PIA for short, takes care of the entire data exchange between the outside world and the Junior Computer. The outside world can assume various forms: it can be a hexadecimal keyboard, a seven-segment display, an ASCII keyboard (similar to that of a typewriter), a printer or even a servo from a model aircraft. This chapter will describe the way in which the computer controls peripheral devices connected to it, how it receives and transmits data via the PIA and how this data is processed.

A computer without any inputs or outputs would be a virtually useless machine, for then it would be unable to communicate with the operator, who plays an essential part in the execution of a program. The situation would be rather like that of a university professor who, having reached a high level in scientific research, is unable to impart any knowledge to his students due to a total lack of vocabulary. Similarly, if a computer did not react to the inaptitude and inconsistency of the (human) programmer, the latter would never learn how to operate the computer correctly.

How can a computer communicate its 'thoughts' to the outside world? The data exchange that takes place between the operator and the computer is just like a conversation between two people. To be able to pass on thoughts and ideas stored in his brain, man uses language. The computer also has thoughts and ideas, stored in the memory in the form of processed data. When such data is to be imparted to the outside world the computer makes use of a printer or, as in the case of the Junior Computer, a six digit (alpha) numeric display.

To be able to follow a conversation it is just as important to listen as it is to talk, for this is the only way to receive information from the person



you are talking to and in fact the only way to find out whether you are "getting through" to him. The computer "listens" by means of a keyboard on which the incoming data is typed. In large(r) computers this will be an ASCII keyboard, but the basic Junior Computer is quite content with a hexadecimal version.

Just as a person uses his mouth to speak and his ears to listen, the Junior Computer transmits and receives information via the PIA. Therefore, the PIA must have suitable inputs and outputs to receive and transmit data, respectively. The term "data" refers to electrical signals which are either logic 0 or logic 1, and which change from one level to the other. The PIA in the Junior Computer is a 6532 type. This LSI (large scale integration) device contains 128 memory locations, a timer which can be programmed in a number of ways, a flag register and an edge detector that can respond to either positive or negative going pulses.

The PIA can be thought of as being a micro-microprocessor which can be programmed to carry out various instructions by means of 19 special 'registers'. Their operation and the numerous ways in which they can be programmed will be fully explained in the course of this chapter.

An overall summary of the 19 registers is given in the following table:

1. PORT A and PORT B
    - PAD: PORT A Data Register
    - PADD: PORT A Data Direction Register
    - PBD: PORT B Data Register
    - PBDD: PORT B Data Direction Register
  2. Timer start; interrupt request is **disabled**
    - CNTA: CLK1T (division factor = 1)
    - CNTB: CLK8T (division factor = 8)
    - CNTC: CLK64T (division factor = 64)
    - CNTD: CLK1KT (division factor = 1024)
  3. The flag register and timer
    - RDFLAG read the flag register
    - RTDEN read timer and enable IRQ
    - RDTDIS read timer and disable IRQ
  4. Timer start; interrupt request is **enabled**
    - CNTE: CLK1T (division factor = 1)
    - CNTF: CLK8T (division factor = 8)
    - CNTG: CLK64T (division factor = 64)
    - CNTH: CLK1KT (division factor = 1024)
  5. PA7 as the edge detector
    - EDETA: sensitive to negative edge, PA7-IRQ disabled
    - EDETB: sensitive to positive edge, PA7-IRQ disabled
    - EDETC: sensitive to negative edge, PA7-IRQ enabled
    - EDETD: sensitive to positive edge, PA7-IRQ enabled
- Each register will be dealt with in turn.

## PORT A and PORT B

The 6532 contains two ports: PORT A and PORT B. Together they provide the input and output connections for the peripheral interface adapter. Each of the sixteen lines can be individually programmed as an

input or an output. The operator can connect various forms of peripheral equipment to any of the input/output lines. In the Junior Computer each of the sixteen lines is fed to a multi-way connector, or port connector. Figure 1 shows that the hexadecimal keyboard and display are also connected to the PIA, so that the operator does not have the full 16 lines at his/her disposal. The lines that can be used are PA7, PB0 and PB5 . . . PB7. If the keyboard and display were made redundant, however, all the port lines would be available.

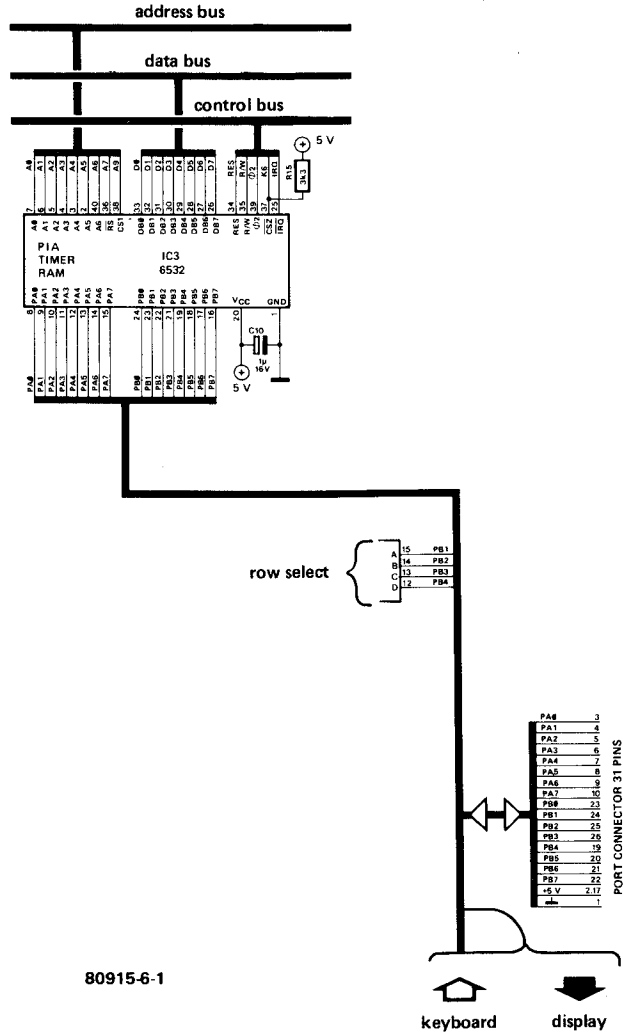


Figure 1. The hexadecimal keyboard and display are connected to the input/output ports of the PIA. The remaining port lines, PA7, PB0 and PB5 . . . PB7 can be used for the operator's programs.

## The internal structure of the PIA

A similar diagram to that shown in figure 1b (Book 1, chapter 3) for the microprocessor may be drawn for the PIA. It is given in figure 2.

The PIA is connected to the microprocessor by means of three buses: the data bus, the address bus and the control bus. Thus, it is connected to the CPU just like any other "normal" memory device. The three buses operate as follows:

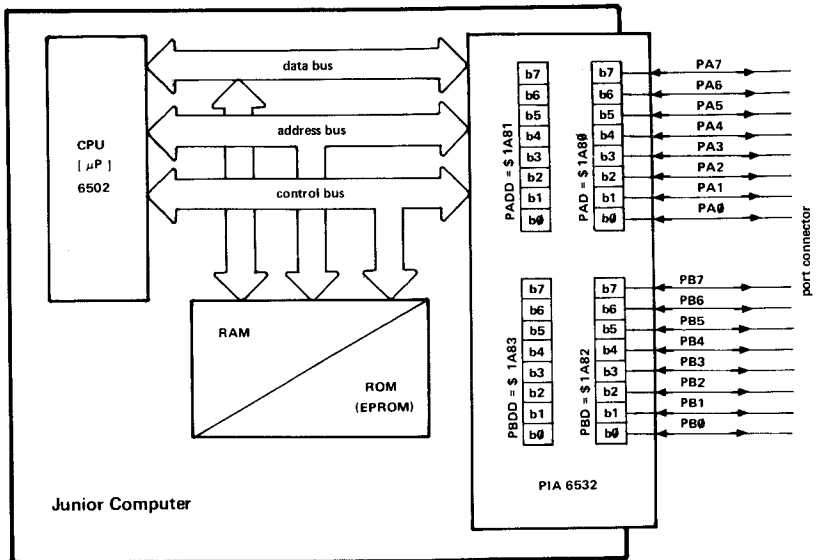
- The address bus selects internal memory locations.
- The data bus allows bi-directional data transfer between the CPU and the PIA. The direction of data transfer is controlled by the  $R/\overline{W}$  line connected to the processor.
- The CPU transmits the following signals to the PIA by way of the control bus:
  - \* the  $R/\overline{W}$  signal
  - \* the  $\Phi 2$  clock signal
  - \* the  $\overline{CS2}$  signal
  - \* and the  $\overline{RES}$  (reset) signal

The  $\overline{IRQ}$  line of the PIA is also connected to the CPU by means of the control bus. The PIA can therefore request an interrupt. How this is put into effect will be discussed later.

This covers all the connections between the PIA and the microprocessor. It is now time to take a closer look at the 'internal organs' of the PIA. When discussing the PIA, hardware and software are equally important. The operator sees the PIA as:

- **Hardware:** sixteen lines which can be independently programmed as inputs or outputs. They are divided into two sections: port A and port B. Both ports consist of eight lines each, that is to say, they are 8 bits wide. The port lines are called  $PA0 \dots PA7$  and  $PB0 \dots PB7$ . The data bus is connected to the individual port lines and, as there are two ports, it can be switched either to port A or to port B. Since the data bus is bi-directional, the CPU may write onto the port lines, or read data from them. When the microprocessor writes a certain bit pattern onto port lines which have been programmed as outputs, this pattern will remain stored on the output port lines, even after the write operation has ceased (**latch function** of an output port line). When the microprocessor reads one or more port lines which have been programmed as inputs, the logic level present at that instant will be read (there is no latch function for an input port line).
- **Software:** four registers for storing or retrieving data. They are:
  - PAD = PORT A Data register
  - PADD = PORT A Data Direction register
  - PBD = PORT B Data register
  - PBDD = PORT B Data Direction registerAs can be seen, each port is made up of two registers:
  1. a data register
  2. a data direction register

The data direction register controls the direction of data transfer for each individual port line. As this register is 8 bits wide, the CPU can write a word into the data direction register by means of the data bus.



PAD = Port A Data Register  
PADD = Port A Data Direction Register  
PBD = Port B Data Register  
PBDD = Port B Data Direction Register

80915-6-2

**Figure 2.** The internal view of the peripheral interface adapter (PIA). It contains two ports, each with eight input/output lines. The contents of the data direction registers determine whether a particular port line is to be used as an input or as an output. The PIA is connected to the CPU by means of the address bus, the data bus and the control bus.

This word will be a bit pattern of “noughts” and “ones” and will have the following effect on the port lines:

- a nought in any position in the data direction register will program the corresponding port line to be an input.
- a one in any position in the data direction register will program the corresponding port line to be an output.

In the Junior Computer the four registers have been assigned the following address locations:

PAD : \$ 1A80  
PADD : \$ 1A81  
PBD : \$ 1A82  
PBDD : \$ 1A83

Now for a few examples to familiarise ourselves with port programming. To start with, several port lines are to be programmed either as inputs or outputs.

1. All the port lines belonging to port A are to be inputs, while all those belonging to port B are to be outputs. The program will then look like this:

LDA # 00 all bits in the accumulator are zero  
 STA-PADD lines PA0 . . . PA7 are inputs  
 LDA # FF all bits in the accumulator are ones  
 STA-PBDD lines PB0 . . . PB7 are outputs

As you know, the individual bits in the port data direction register determine whether a port line is an input or an output. In the data direction register PADD they are all noughts, so that the port A lines are all programmed as inputs. The CPU can now read the data on the port lines and enter that data into the accumulator, the X register or the Y register. This will be further described later on. In the data direction register PBDD all the bits are ones, causing all the lines to port B to be programmed as outputs. The CPU can now write any bit pattern onto the port lines by way of the data bus. Again, this will be expanded on later.

2. Port lines PA4 and PB0 are to be outputs; the other port lines are to be inputs. The following bit patterns are therefore entered into the two data direction registers:

PADD	b7	b6	b5	b4	b3	b2	b1	b0
	0	0	0	1	0	0	0	0
PBDD	b7	b6	b5	b4	b3	b2	b1	b0
	0	0	0	0	0	0	0	1

The program for the above will be:

LDA # 10 load accumulator with required bit pattern  
 STA-PADD line PA4 is an output  
 LDA # 01 load accumulator with required bit pattern  
 STA-PBDD line PB0 is an output

All the other port lines are programmed as inputs.

## Reading from and writing to the port lines

We now know that the port lines can be programmed as inputs or outputs by the information contained in the data direction registers, but how can the microprocessor read or write data from or to them? This facility is provided by the two data registers, PAD and PBD.

### Reading port lines PA0 . . . PA7 and PB0 . . . PB7

When the data registers PAD or PBD are read, the bit pattern on the port lines at that moment is transferred, via the data bus, to one of the internal CPU registers. In other words, the port lines are an extension of the data bus. **If the processor is to read one or more port lines, they must be programmed as inputs.**

The following example illustrates this:

LDA # 00  
 STA-PADD PA0 . . . PA7 are programmed as inputs  
 LDX-PAD the bit pattern on the port lines is stored in the X register of the CPU

If the bit pattern at port A is 10101110,

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	Port A lines
1	0	1	0	1	1	1	0	bit pattern

the hexadecimal number AE will be stored in the X register.

If the voltage on a particular port line is less than 0.4 V, the microprocessor will interpret it as logic "0". The voltage must be greater than 2.4 V for a logic "1" to be read.

**Note:** If a port line programmed as an input is connected to the output of a gate, a transistor or another type of logic device, a current of up to 1.6 mA could be produced. This must be taken into account, especially when connecting certain CMOS ICs, so as not to destroy the particular device. When the microprocessor reads the port lines the bit pattern present at that moment will be transferred to one of the CPU registers. Port lines programmed as inputs **do not** therefore, have a latch function.

### Writing to port lines PA0 . . . PA7 or PB0 . . . PB7

When writing to data registers PAD or PBD, the bit pattern contained in one of the CPU registers will be transferred to the port lines, which can be considered as an extension of the data bus. **If the processor is to write to one or more port lines, they must be programmed as outputs.** This is illustrated by the following:

```
LDA # FF
```

```
STA-PADD PA0 . . . PA7 are programmed as outputs
```

```
LDX # C3
```

```
STX-PAD the bit pattern contained in the X register is transferred to port A lines.
```

The contents of the X register is C3 = 11000011. This bit pattern will appear on port lines PA7 . . . PA0:

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	port A lines
1	1	0	0	0	0	1	1	

The hexadecimal number C3 will remain on the port lines until the next write operation (latch function).

**Note:** If a port line programmed as an output is connected to the input of a gate, a transistor or another type of logic device, these may not draw more than 1.6 mA from the output port. The port lines are only capable of driving one standard TTL load. If TTL compatibility is not required, port lines PB0 . . . PB7 may be used as a current source to directly drive the base of a transistor switch (maximum of 3 mA at 1.5 V).

**Warning!** The port lines of the PIA are **not** protected against excess voltage or current. When they are directly connected to an external device such as a printer, the voltage at the port lines should be prevented from exceeding +7 V or from becoming negative, for then the 6532 is likely to go up in smoke!!

## First steps

Now that we have discovered all there is to know about port lines, it is high time to write a short demonstration program which uses all four I/O registers. The program should achieve the following:

1. At port A the state of eight switches is to be read by the CPU.
2. The Junior Computer is to convert this information into an audible signal.
3. The signal is to be fed out of PB0. An amplifier must be provided to control a small 8 Ω loudspeaker.

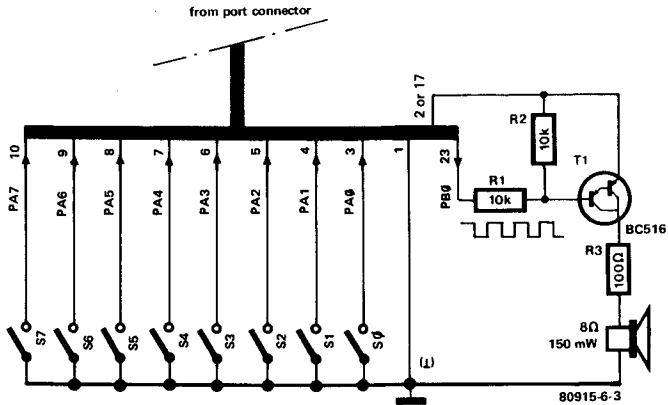


Figure 3. The hardware required for the DEMO program of figure 4. The bit pattern produced by the (closed) switches is read into the computer via port A. This information is then converted into a squarewave signal which is fed to the loudspeaker amplifier via port line PB0.

4. A circuit diagram must be drawn to show how these requirements are to be met.
5. Subsequently, a flow chart must be drawn so that the program can be entered into the computer via the editor and then assembled.

Points 1 . . . 3 will be fulfilled by the program itself. Point 4 tackles the problem by producing a circuit diagram. This is given in figure 3 and shows how eight switches S0 . . . S7 are connected to port lines PA0 . . . PA7. Since the Junior Computer must read the state of these switches (whether they are on or off), all port A lines must be programmed as inputs. An audible signal is to be produced at PB0, this line must, therefore, be programmed as an output. Transistor T1 is included to enable the signal to be heard via a loudspeaker. This completes the necessary hardware to solve the problem.

Next, the software, the program (point 5) must be developed. Figure 4 gives a program which converts the switch information into a particular frequency or squarewave. It should be noted that this program is not particularly elegant, but it serves the purpose of illustrating how the four I/O registers PAD, PADD, PBD and PBDD are controlled.

At the start of the demonstration program, DEMO, port lines PA0 . . . PA7 are programmed as inputs and port line PB0 as an output. All the bits in the data direction register PADD are zero and b0 in the data direction register PBDD is a one. This situation will remain unchanged throughout the program. The following part of our program involves the frequency loop (label FREQ). Here, the processor reads the state of the switches S0 . . . S7 into the accumulator. Prior to this we must decide whether a closed switch corresponds to a logic one or to a logic nought level:

- a closed switch is logic 1
- an open switch is logic 0

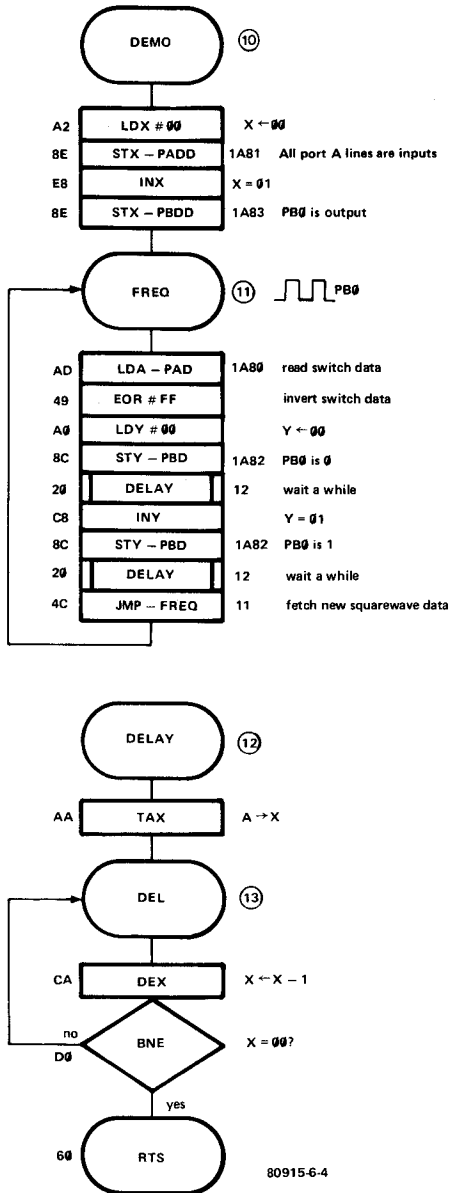


Figure 4. The flowchart of the DEMO program which, with the hardware from figure 3, will produce a separate tone for each individual key (or combination of keys). The program can be entered into the Junior Computer with the aid of the editor and assembler contained in the monitor program.



After the bit pattern present on the port lines has been copied into the accumulator (LDA-PAD), it must be inverted: EOR # FF. An audible signal must now be produced which is dependent on the switch information. The easiest way to achieve this is to alternately set the output port line PB0 high and low for a certain amount of time. This will effectively produce a squarewave signal which can be rendered audible by means of the loudspeaker interface connected to PB0.

The Y register is used to "swing" the port line PB0 high and low. The least significant bit (b0) alternates from nought to one and is copied into the data register PBD. Subroutine DELAY determines the length of time that is high or low. Here the CPU copies the bit pattern of the switch information into the X register (TAX). Then the X register is decremented until it becomes zero. The amount of time this process takes depends on the state of the switches. Thus, the processor is able to generate a whole range of frequencies.

### Editing and assembling the DEMO program

Once the flow chart has been drawn up, the program can be entered into the computer. It is to be assembled in page zero, where locations 0000 . . . 00E0 are available. For this reason the BEGAD pointer is set to 0000 and the ENDA pointer to 00E0. By placing the start address of the assembler 1F51 in locations 1A7A and 1A7B (the NMI vector), the assembler can be started by pressing the ST key. The editor will then be exited from via a non-maskable interrupt. The "keying-in" procedure shown below illustrates how the DEMO program is entered:

keyboard	display	comments
RST		
AD 0 0 E 2	00E2 XX	
DA 0 0	00E2 00 }	BEGAD = 0000
+ 0 0	00E3 00 }	
+ E 0	00E4 E0 }	
+ 0 0	00E5 00 }	ENDAD = 00E0
AD 1 A 7 A	1A7A XX	
DA 5 1	1A7A 51 }	NMI vector = 1F51
+ 1 F	1A7B 1F }	
AD 1 C B 5	1CB5 20	start address of editor
GO	77	editor running
INSERT F F 1 0 0 0	FF 10 00	label 10: DEMO
INPUT A 2 0 0	A2 00	LDX # 00
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 0 0	FF 11 00	label 11: FREQ
INPUT A D 8 0 1 A	AD 80 1A	LDA-PAD
INPUT 4 9 F F	49 FF	EOR # FF
INPUT A 0 0 0	A0 00	LDY # 00
INPUT 8 C 8 2 1 A	8C 82 1A	STY-PBD
INPUT 2 0 1 2 0 0	20 12 00	JSR-DELAY (label 12)

INPUT	C 8	C8	INY
INPUT	8 C 8 2 1 A	8C 82 1A	STY-PBD
INPUT	2 0 1 2 0 0	20 12 00	JSR-DELAY (label 12)
INPUT	4 C 1 1 0 0	4C 11 00	JMP-FREQ (label 11)
INPUT	F F 1 2 0 0	FF 12 00	label 12: DELAY
INPUT	A A	AA	TAX
INPUT	F F 1 3 0 0	FF 13 00	label 13: DEL
INPUT	C A	CA	DEX
INPUT	D 0 1 3	D0 13	BNE to label 13
INPUT	6 0	60	RTS
ST		XXXX XX	start assembler via NMI
AD	0 0 0 0	0000 A2	start address of DEMO
GO			start DEMO

## Music on the Junior Computer

Tunes can be played on the Junior Computer by connecting a small keyboard to port lines. The melody played can be heard via a simple loudspeaker interface. Construction of the Junior "piano" involves the following requirements:

1. The keyboard should have black and white keys similar to that of a piano or organ.
2. The keys should be arranged in an electrical matrix and the entire keyboard must be connected to port A.
3. The tones must be audible!!

A summary of the requirements is given in figure 5. It includes:

- a piano keyboard
- a key matrix with connections to port A
- an amplifier circuit connected to port line PB0.

As the keys are arranged in a matrix (figure 5), each key can be assigned a certain value. The matrix format is 4 times 4. The rows are designated as ROW 0 . . . ROW 3 and the columns as COL 0 . . . COL 3. The values of the individual keys are also indicated. Figure 5 also shows an alternative construction for the keyboard using Digitast switches.

Since all the keys are connected to only one port (port A), some of the port lines must be used as inputs and some as outputs. Thus:

- Columns COL 0 . . . COL 3 are connected to PA0 . . . PA3. Since the computer has to read the column information to calculate key values, port lines PA0 . . . PA3 will have to be inputs.
- Rows ROW 0 . . . ROW 3 are connected to PA4 . . . PA7 and must become logic 0 one after the other, so that the computer can detect whether a key is depressed, identify it and calculate its value. As the computer has to write certain patterns onto matrix rows ROW 0 . . . ROW 3, port lines PA4 . . . PA7 must be outputs.

We have already seen how to connect a simple loudspeaker interface. Again, PB0 is to be programmed as the sound output.

This covers the mechanical and electrical structure of the "piano" keyboard. Let us now develop a short program that will enable a tune to be played on the keyboard. It must be constructed step by step and should comprise the following:

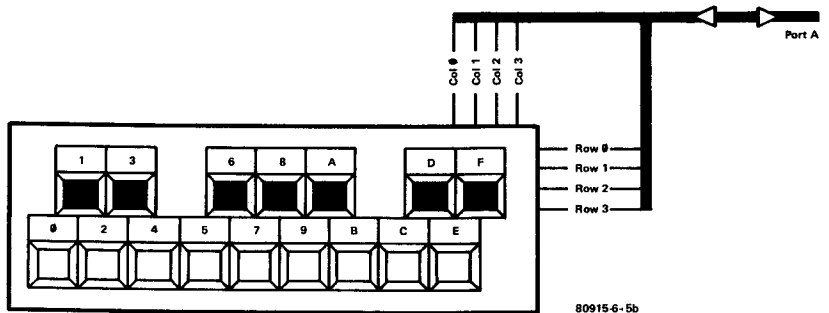
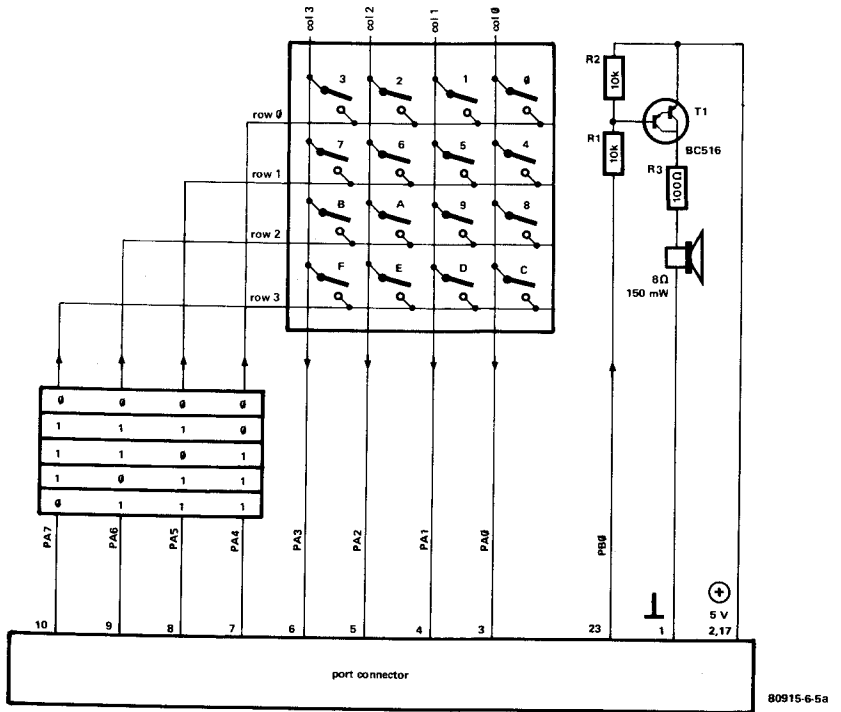


Figure 5. The hardware required for the PLAY program (see figure 7). The keys are arranged in a four-by-four matrix (5a). Each key will produce a separate note. The frequency codes for each of these notes are contained in the look-up table DEL (see figure 10). Figure 5b shows a possible layout for an inexpensive alternative to a (proper) piano keyboard.

- a routine to calculate the values of the 16 keys
- a routine to establish whether any key is depressed
- a routine to "debounce" the keys
- a routine to allocate a particular frequency to each key that is depressed
- a look-up table in which the 16 key frequencies are stored

Figure 6 shows the flowchart of a program to calculate the value of the 16 keys. Since it scans the key matrix given in figure 5 the I/O lines of port A must be programmed before the processor jumps to this subroutine (KEYVAL). As we know, port lines PA7 . . . PA4 are outputs and PA3 . . . PA0 are inputs. Thus, the data direction register belonging to port A must be loaded with the bit pattern 11110000 = F0. This can be accomplished by the instructions:

```
LDA # F0
STA-PADD
```

All the I/O lines for port A have now been programmed and the subroutine KEYVAL can be called. During its description the key matrix in figure 5 and the flowchart in figure 6 will be referred to.

At the start of the routine memory location ROW is loaded with the value F7 (= 11110111). The X register operates as a row counter, its contents determine which row of the keyboard matrix is being read into the computer via port A:

```
X = 03 corresponds to ROW 0
X = 02 corresponds to ROW 1
X = 01 corresponds to ROW 2
X = 00 corresponds to ROW 3
```

Each time the X register is decremented, the processor moves the contents of memory location ROW one bit position to the left and stores it in the data register PAD. As a result, the matrix lines will each go low in succession:

X register	PA7 . . . PA4
X = 04	1111
X = 03	1110
X = 02	1101
X = 01	1011
X = 00	0111

As each individual matrix row goes low in turn, the processor can easily establish which key has been depressed in which row. If no key is depressed, port lines PA3 . . . PA0 will all be high. The least significant "nibble" (four bits) of the data byte will therefore consist entirely of ones. If a single key is depressed, port lines PA3 . . . PA0 will be 1110 or 1101 or 1011 or 0111. If the least significant nibble contains several noughts, it means that several keys in the matrix row in question were depressed at the same time.

Since the entire range of port A data register is read (LDA-PAD) the four most significant bits, which we are not interested in, must be masked out:

```
LDA-PAD load information from port A into accumulator
AND # 0F mask out four most significant bits (0000XXXX)
```



Then by comparing this information with  $\emptyset F$  (CMP #  $\emptyset F$ ) and branching (BEQ), the processor establishes whether a key was depressed in the matrix row being scanned. When a row is found to contain a depressed key, the row number is saved in location TEMPX and the bit number of the column (PA3 . . . PA $\emptyset$ ) in location KEY.

In the next section of the program (KEYB) the X register will operate as a column counter (before, it acted as a row counter). The processor shifts the contents of KEY (= column information) one bit to the right until the carry flag is zero. If after four shift operations, the carry flag is still high, there must be a fault in the program and the computer will try to calculate the key values once more (branch to label KEYVAL). If, however, the carry flag has been reset, the program will ask in which matrix row the key was depressed. This information is stored in the memory location TEMPX. By a series of simple comparisons (CMP) the true value of the key can now be established.

The value of the keys increases by four after each row. The keys in ROW 1 will therefore be four times greater than those in ROW  $\emptyset$ , the keys in ROW 2 eight times greater and the keys in ROW 3 twelve times greater. By simply adding this value to the column number, the true key value can be found. Once the key value has been calculated, the computer stores it in location KEY under the section of program labelled KEYC.

Finally, all the keyboard matrix rows are cleared before returning to the main routine. This is important for the next subroutine KEYIN (figure 8a) which is used to check whether any key has been depressed. If a key has been depressed, one of the matrix columns (PA3 . . . PA $\emptyset$ ) must be logic  $\emptyset$ . Since KEYIN inverts the column pattern read (EOR #  $\emptyset F$ ) it can be seen that:

the contents of the accumulator =  $\emptyset\emptyset$ , if no key is depressed

the contents of the accumulator  $\neq \emptyset\emptyset$ , if a key is depressed.

The next subroutine is called DELAY (see figure 8b). It is used to 'debounce' a depressed key. The required delay is obtained by means of the loop DELA.

Note: In chapter 7 of this book we will discuss the keyboard and display routines in the monitor program of the standard Junior Computer. The keyboard routine is very similar to the subroutine KEYVAL. Nevertheless, there is a considerable difference as far as programming the I/O is concerned.

In KEYVAL one section of the port A lines acts as an input and the other as an output (matrix structure). In the case of the monitor routines SCANDS, SCAND and GETKEY (described in the next chapter), the whole of port A is programmed as an input and the whole of port B as an output during keyboard scanning.

The flowchart of the main routine PLAY is shown in figure 7. This routine assigns a certain frequency to each key that is depressed and renders a tune audible via the loudspeaker. Each of the required port lines are defined at the start of this program:

- PA7 . . . PA4 are outputs
- PA3 . . . PA $\emptyset$  are inputs
- PB $\emptyset$  is an output

When label PA is reached the Junior Computer waits for a key to be

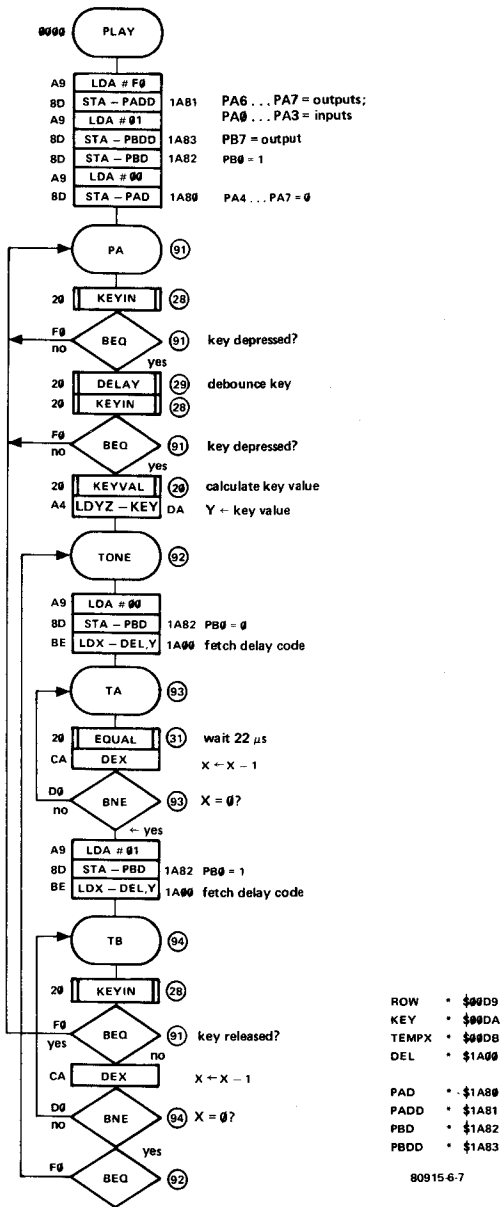


Figure 7. The flowchart of the main portion of the PLAY program. This program decodes the value of a depressed key and converts it into an audible tone. During this procedure the processor utilizes the look-up table DEL to determine the actual frequency of the note to be played.

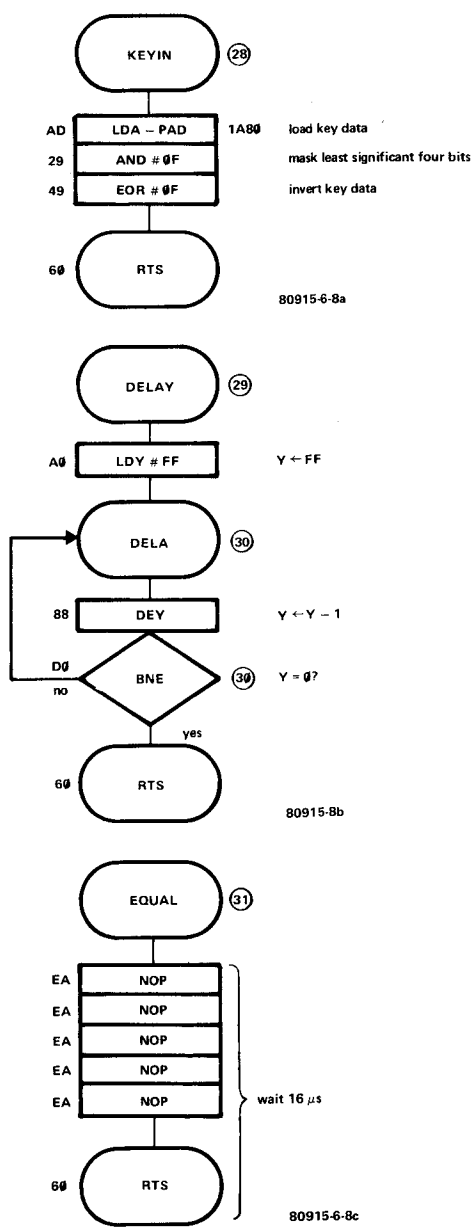


Figure 8. The flowcharts of the subroutines KEYIN, DELAY and EQUAL are also used by the PLAY program. Subroutine KEYIN detects whether or not a key is depressed while DELAY 'debounces' the key. Subroutine EQUAL is incorporated to ensure that the mark/space ratio of the generated squarewave signal is equal.



depressed. This is detected by means of the subroutine KEYIN (figure 8a). Once a depressed key is detected the program jumps to the subroutine DELAY (figure 8b), which debounces the keyboard. After this, the computer runs through the subroutine KEYIN once more to check whether the key is still depressed. Only then will the computer calculate the value of the depressed key by jumping to the subroutine KEYVAL (figure 6). The key value is yet to be converted into a frequency. This happens after label TONE. Before considering the details, let's look at the TONE routine as a whole:

- label TONE: the loudspeaker is switched 'on' for a certain length of time. The actual duration will be determined by the program loop TA . . . BNE . . . TA and will depend on the contents of the X register. Before the computer reaches this program loop, it loads a number (delay) corresponding to the frequency of the depressed key into the X register. The value of this frequency is stored in the look-up table DEL, the structure of which will be considered later.

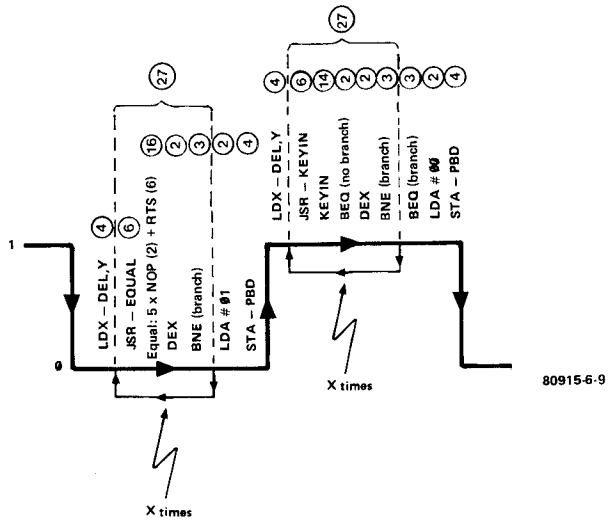
- label TB: the loudspeaker is switched 'off' for a certain length of time. The off time is determined by the program loop TB . . . BNE . . . TB. Again, the duration of the program loop depends on the contents of the X register. As can be seen from the flowchart, the microprocessor jumps repeatedly to the subroutine KEYIN to check whether the key is still depressed. If so, a squarewave signal will appear at PB0. If, however, the key has now been released, the program will branch to label PA where it will wait for a new key to be depressed.

### The look-up table and frequency values

Before the values of the delays for each frequency can be calculated for the look-up table, the program loops TA . . . BNE . . . TA and TB . . . BNE . . . TB must be examined. At the start of TB the CPU jumps to the subroutine KEYIN, carries out the subsequent BEQ instruction without branching and, if a key is depressed, decrements the X register before branching back to label TB. All the instructions require a certain period of time for execution. The execution times for the relevant instructions are listed below:

JSR-KEYIN	=	6 $\mu$ s
LDA-PAD	=	4 $\mu$ s
AND # 0F	=	2 $\mu$ s
EOR # 0F	=	2 $\mu$ s
RTS	=	6 $\mu$ s
BEQ	=	2 $\mu$ s no branch while the key is depressed
DEX	=	2 $\mu$ s
BNE-TB	=	3 $\mu$ s branch is executed
<hr style="width: 100%; border: 0.5px solid black;"/>		
total	=	27 $\mu$ s

The loop time therefore lasts 27  $\mu$ s times the contents of the X register. The loop TA . . . BNE . . . TA must also last 27  $\mu$ s. Since no jump is made to the subroutine KEYIN, the loop time will have to be corrected. That is why the subroutine EQUAL (figure 8c) is included to supplement the missing 22  $\mu$ s. This is shown in the form of a graph in figure 9.



**Figure 9.** This is how the Junior Computer actually manages to produce a squarewave signal at the output PB0. As can be seen, without the EQUAL subroutine the signal would be asymmetric.

The above enables us to calculate the values for the look-up table. Figure 10 shows some figures on the keys which call for a few words of explanation. If, for example, we take note 'a' with a frequency of 440 Hz, the relationship between time and frequency will be:

$$T = \frac{1}{f}.$$

A frequency of 440 Hz therefore has a period of

$$\frac{1}{440} \text{ Hz} = 2273 \mu\text{s}.$$

On a piano keyboard the frequency of each note either rises or falls from key to key by a factor of  $12\sqrt{2} = 1.0594613$ . This enables frequencies or periods to be allocated to all the keys, as shown in figure 10. Furthermore, the duration of both program loops is known to be  $27 \mu\text{s}$  and so the exact delays for each key can be calculated.

Let's look at note "a" again. Its period is  $2273 \mu\text{s}$  and so the value for the delay in the look-up table must be:

$$\frac{2273}{27} = 84_{10} = \$ 54 \mu\text{s}.$$

Once the delay values for each key have been calculated the look-up table DEL can be completed. Since the microprocessor fetches the delay information for each note from the look-up table twice, the played notes will be an octave lower than those calculated.

It is now time to enter the complete program PLAY including its sub-routines KEYVAL, KEYIN, DELAY and EQUAL into the computer. For this we utilise the editor and assembler contained in the monitor program.

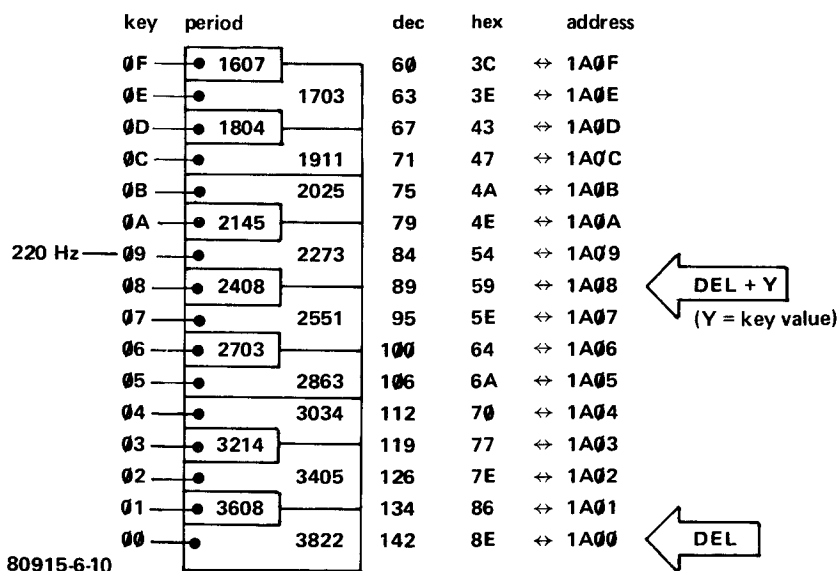


Figure 10. The 'software keyboard' of the PLAY program. The frequency of the note produced by each key of figure 5 is stored in the look-up table DEL (address location 1A00).

Once the program has been assembled and the look-up table has been entered (page 1A), the keyboard and the loudspeaker interface can be connected to the PIA.

The complete "keying in" procedure is given below:

key:	display:	comments:
RST	xxxx xx	
AD 0 0 E 2	00E2 xx	
DA 0 0	00E2 00	} BEGAD = 0000
+ 0 0	00E3 00	
+ E 0	00E4 E0	} ENDA = 00E0
+ 0 0	00E5 00	
AD 1 A 7 A	1A7A xx	} NMI-vector = 1F51 (start address of assembler)
DA 5 1	1A7A 51	
+ 1 F	1A7B 1F	
AD 1 C B 5	1CB5 20	start address of editor
GO	77	editor running
INSERT 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 A 9 0 0	A9 00	LDA# 00
INPUT 8 D 8 0 1 A	8D 80 1A	STA-PAD

key:	display:	comments:
INPUT F F 9 1 0 0	FF 91 00	label 91: PA
INPUT 2 0 2 8 0 0	20 28 00	JSR-KEYIN (label 28)
INPUT F 0 9 1	F0 91	BEQ to PA (label 91)
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 9 1	F0 91	BEQ to PA (label 91)
INPUT 2 0 2 0 0 0	20 20 00	JSR-KEYVAL (label 20)
INPUT A 4 D A	A4 DA	LDYZ-KEY (00DA)
INPUT F F 9 2 0 0	FF 92 00	label 92: 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 (DEL = 1A00)
INPUT F F 9 3 0 0	FF 93 00	label 93: TA
INPUT 2 0 3 1 0 0	20 31 00	JSR-EQUAL (label 31)
INPUT C A	CA	DEX
INPUT D 0 9 3	D0 93	BNE to TA (label 93)
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 9 4 0 0	FF 94 00	label 94: TB
INPUT 2 0 2 8 0 0	20 28 00	JSR-KEYIN (label 28)
INPUT F 0 9 1	F0 91	BEQ to PA (label 91)
INPUT C A	CA	DEX
INPUT D 0 9 4	D0 94	BNE to TB (label 94)
INPUT F 0 9 2	F0 92	BEQ to TONE (label 92)
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 to 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 to KEYA (label 21)
INPUT 8 6 D B	86 DB	STXZ-TEMPX (00DB)
INPUT 8 5 D A	85 DA	STAZ-KEY (00DA)
INPUT A 2 0 0	A2 00	LDX # 00
INPUT F F 2 2 0 0	FF 22 00	label 22: KEYB
INPUT 4 6 D A	46 DA	LSRZ-KEY (00DA)
INPUT 9 0 2 3	90 23	BCC to ROWA (label 23)
INPUT E 8	E8	INX
INPUT E 0 0 4	E0 04	CPX # 04
INPUT D 0 2 2	D0 22	BNE to KEYB (label 22)
INPUT F 0 2 0	F0 20	BEQ to KEYVAL (label 20)

key:						comments:
INPUT	F F 2 3 0 0	FF 23	00			label 23: ROWA
INPUT	A 5 D B	A5 DB				LDAZ-TEMPX (00DB)
INPUT	C 9 0 3	C9 03				CMP # 03
INPUT	D 0 2 4	D0 24				BNE to ROWB (label 24)
INPUT	8 A	8A				TXA
INPUT	4 C 2 7 0 0	4C 27	00			JMP-KEYC (label 27)
INPUT	F F 2 4 0 0	FF 24	00			label 24: ROWB
INPUT	C 9 0 2	C9 02				CMP # 02
INPUT	D 0 2 5	D0 25				BNE to ROWC (label 25)
INPUT	8 A	8A				TXA
INPUT	1 8	18				CLC
INPUT	6 9 0 4	69 04				ADC # 04
INPUT	D 0 2 7	D0 27				BNE to KEYC (label 27)
INPUT	F F 2 5 0 0	FF 25	00			label 25: ROWC
INPUT	C 9 0 1	C9 01				CMP # 01
INPUT	D 0 2 6	D0 26				BNE to ROWD (label 26)
INPUT	8 A	8A				TXA
INPUT	1 8	18				CLC
INPUT	6 9 0 8	69 08				ADC # 08
INPUT	D 0 2 7	D0 27				BNE to KEYC (label 27)
INPUT	F F 2 6 0 0	FF 26	00			label 26: ROWD
INPUT	C 9 0 0	C9 00				CMP # 00
INPUT	D 0 2 0	D0 20				BNE to KEYVAL (label 20)
INPUT	8 A	8A				TXA
INPUT	1 8	18				CLC
INPUT	6 9 0 C	69 0C				ADC # 0C
INPUT	F F 2 7 0 0	FF 27	00			label 27: KEYC
INPUT	8 5 D A	85 DA				STAZ-KEY (00DA)
INPUT	A 9 0 0	A9 00				LDA # 00
INPUT	8 D 8 0 1 A	8D 80	1A			STA-PAD
INPUT	6 0	60				RTS
INPUT	F F 2 8 0 0	FF 28	00			label 28: KEYIN
INPUT	A D 8 0 1 A	AD 80	1A			LDA-PAD
INPUT	2 9 0 F	29 0F				AND # 0F
INPUT	4 9 0 F	49 0F				EOR # 0F
INPUT	6 0	60				RTS
INPUT	F F 2 9 0 0	FF 29	00			label 29: DELAY
INPUT	A 0 F F	A0 FF				LDY # FF
INPUT	F F 3 0 0 0	FF 30	00			label 30: DELA
INPUT	8 8	88				DEY
INPUT	D 0 3 0	D0 30				BNE to DELA (label 30)
INPUT	6 0	60				RTS
INPUT	F F 3 1 0 0	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 0	60				RTS

The program can then be tested for any errors that may have cropped up during entry:

key:	etc.
SEARCH A 9 F 0	A9 F0
SKIP	8D 81 1A
SKIP	A9 01
SKIP	8D 83 1A
SKIP	8D 82 1A
SKIP	A9 00
SKIP	8D 80 1A
SKIP	FF 91 00
SKIP	20 28 00
SKIP	F0 91
SKIP	20 29 00

Once everything is correct the assembler can be started:

key: display:  
ST xxxxxx

key:		display:
AD	1 A 0 0	1A00 XX
DA	8 E	1A00 8E
+	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

Finally, the program can be started:

key: display:  
AD 0 0 0 0 0000A9  
GO

Now you can have fun playing tunes on your Junior Computer!

## The interval timer

Another major feature of the Junior Computer is the interval timer which is also incorporated in the 6532 multi-function device. This timer is programmable and operates independently from the microprocessor. This means that while the timer is operating, the CPU is at the disposal of the programmer, which is an obvious requirement if the operator is to work efficiently and effectively with the microcomputer.

The interval timer and the PIA are both built into the same chip, thereby reducing the component count and the overall cost. These two important components increase the performance and versatility of the Junior Computer considerably. However, before being able to operate the interval timer we must know how it works. Let us therefore describe the internal structure of the timer and then develop a few demonstration programs to show how it can be used.

### Block diagram

The block diagram of the interval timer is shown in figure 11. It consists of three sections:

— **Programmable register - timer:** The timer register is eight bits wide and is connected to the data bus. It can be compared with an ordinary RAM location, for data may be written to or read from it.

— **Divider:** The divider is connected to both address lines A0 and A1 and to the clock signal  $\Phi 2$ . The programmable register and the divider combined constitute the complete interval timer.

The interval timer may be compared to a pre-settable down counter (TTL or CMOS). Such a device can be preset to a certain binary number. Each clock pulse will then decrement the counter by one until its contents reach zero.

This is exactly how the interval timer in the Junior Computer works. The clock signal  $\Phi 2$  is the clock pulse for the interval timer which also counts down. With the aid of the divider the clock signal can be divided by four different factors. The dividing factor is determined by the information presented to address lines A0 and A1. Thus:

A1	A0	division factor
0	0	+ 1T
0	1	+ 8T
1	0	+ 64T
1	1	+ 1024T

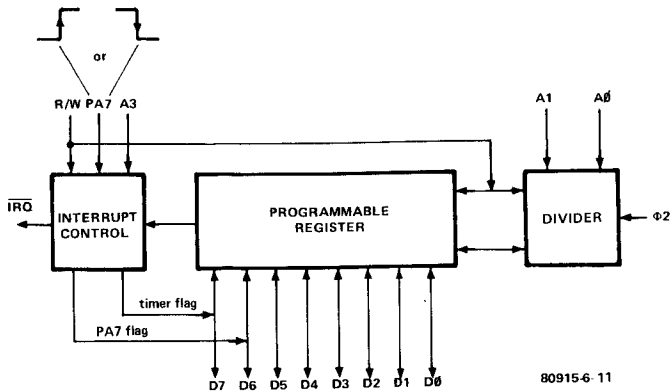
$T = 1 \mu s$  (in the Junior Computer)

The clock signal  $\Phi 2$  is therefore divided by 1, 8, 64 or 1024 before it decrements the contents of the timer register by one. The following example serves to clarify this.

With the aid of the CPU the number \$ 1E is entered into the timer register and a division factor of 64 is selected by means of address lines A0 and A1. The following will then be true:

$$\begin{aligned} \$ 1E &= 30_{10} \text{ and} \\ 30_{10} \text{ times } 64_{10} &\text{ equals } 1920_{10} \end{aligned}$$

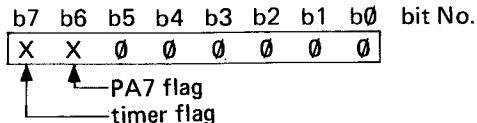
In other words, when the timer register is preset to \$ 1E and the division factor is 64, the contents of the timer register will become zero after



**Figure 11.** The block diagram of the interval timer contained in the PIA. It consists of three basic units. The programmable register (timer) can be likened to a presettable down counter. When data is entered into this register the interval timer will start to count down. The rate at which it counts is determined by the divider which allows division factors of 1, 8, 64 or 1024 to be selected. When the timer reaches the end of its count, the interrupt flag contained in the interrupt control unit is set.

1920  $\mu$ s. Before we can describe the interval timer in detail however, another important aspect of the 6532 IC has to be considered: the interrupt flag register.

– **Interrupt control:** The interrupt control section of the interval timer contains the interrupt flag register which consists of two flags: the timer interrupt flag and the PA7 interrupt flag (to be discussed in greater detail later). The interrupt flag register is also 8 bits wide, but only two are used (b7 and b6) the remainder (b5 . . . b0) will always be zero when this register is read by the processor. The timer interrupt flag is set by the interval timer (as you may have expected!) and the PA7 interrupt flag is set by a positive or negative going pulse on port line PA7. When a read operation is performed on the interrupt flag register, the bits are transferred to the processor on the data bus, as the diagram below indicates:



The PA7 flag is reset when the interrupt flag register is read. The timer flag is cleared when the timer register is accessed (either written to or read from).

### When is the timer flag set?

Bit b7 in the interrupt flag register is used as the timer flag. This is set (b7 = 1) when the timer has reached a 'time out'. With respect to the previous example this will be:



timer register contents	comments
1E = 00011110	starting value of the timer register
1E = 00011101	after 64 $\mu$ s
1C = 00011100	after a further 64 $\mu$ s
1B = 00011011	after a further 64 $\mu$ s
:	:
:	:
:	:
:	:
02 = 00000010	$30 \times 64 \mu\text{s} = 1920 \mu\text{s}$
01 = 00000001	until the timer register
00 = 00000000	is zero
FF = 11111111	after a further 64 $\mu$ s
FE = 11111110	after a further 64 $\mu$ s (time out)
FD = 11111101	after <i>one</i> $\mu$ s timer flag is set
:	:
:	:
:	:
:	:
02 = 00000010	after another $\mu$ s
01 = 00000001	after another $\mu$ s
00 = 00000000	after another $\mu$ s
FF = 11111111	after another $\mu$ s
FE = 11111110	after another $\mu$ s
:	:
:	:
:	:

It should now be clear how the interrupt flag register works in conjunction with the interval timer. Once the CPU has entered a number — the initial counter setting — into the timer register and the division factor has been determined during the write operation by address lines A0 and A1, the interval timer will start its 'countdown'. With a division factor of 64 and an initial counter setting of  $1E = 30_{10}$ , the contents of the timer register will become zero after 1920  $\mu$ s. The moment at which the timer register contents becomes zero is termed the 'time out'.

### Points to note:

1. The timer flag in the interrupt flag register is reset (logic 0) when the timer register is read (read locations RDTEN or RDTDIS).
2. The timer flag is also reset when something is entered into the timer register (write into locations CNTA . . . CNTH). The timer flag is therefore always reset when the timer is started (when the initial counter setting is entered).
3. The timer flag in the interrupt flag register is set (logic 1) after the time out. In our example:  
initial setting =  $1E = 30_{10}$   
division factor = 64  
time out after  $30 \times 64 \mu\text{s} = 1920 \mu\text{s}$   
the timer flag is set after  $1920 + 1 = 1921 \mu\text{s}$

4. After the time out the division factor is set to one automatically. This is independent of the division factor which was previously selected. If this were 64, for instance, then:
  - before the time out the contents of the timer register are decremented by one every  $64 \mu\text{s}$
  - after the time out the contents of the timer register are decremented by one every  $1 \mu\text{s}$ . One microsecond corresponds to 1 MHz – the clock frequency of the Junior Computer.

The interrupt flag register also controls the interrupt line connected to the IRQ input of the microprocessor. If the interrupt flag (I) in the status register of the CPU is reset, the 6532 can cause an interrupt request (IRQ). The IRQ line will then go low.

The interrupt request line is controlled by both flags in the interrupt flag register. If either the timer flag belonging to the interval timer *or* the PA7 flag is set, the IRQ line can go low to cause an interrupt. The programmer, however, is able to prevent the interrupt flag register from causing an IRQ. Address line A3 is connected to the interrupt flag register and is used to enable or disable the IRQ line:

- A3 = 0 to disable interrupt from timer to IRQ
- A3 = 1 to enable interrupt from timer to IRQ

### Summary

This completes the description of the interval timer block diagram. To sum up, it consists of a divider, a programmable register (the timer) and an interrupt control section. The latter contains the interrupt flag register in which there are two flags: the timer interrupt flag (b7) and the PA7 interrupt flag (b6). These two flags can be used to control the interrupt request line (IRQ) of the microprocessor. Note that the instructions SEI and CLI (set interrupt disable and clear interrupt flag) in the main program have an over-riding effect on the above.

The timer interrupt flag will be reset after a read or write operation from or to the programmable register or after an interrupt occurs. This flag is only set (b7 = 1) one microsecond after a time out. (Note that if the interrupt occurs at the same time that the timer is read, the interrupt flag will *not* be reset).

In addition to its use as a peripheral input/output line, the PA7 pin can function as an edge sensitive input. In this mode, an active transition on the PA7 line will set the internal interrupt flag (b6 of the interrupt flag register). Providing the PA7 interrupt is enabled, the IRQ output will then go low. When this occurs, of course, the CPU will branch to an interrupt routine.

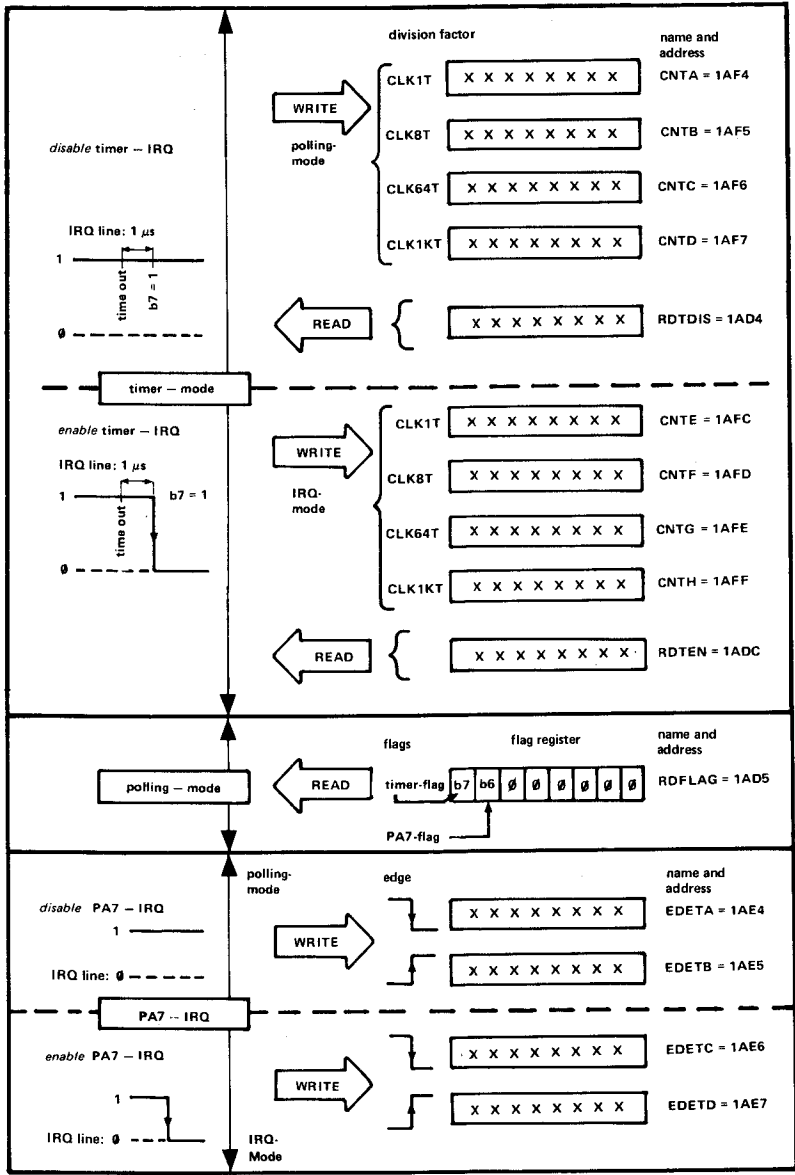
### Internal structure of the interval timer: timer mode

Now that the block diagram of the interval timer has been fully explained, it is time to take a close look at the 'insides' of the device. Figure 12 gives a clear indication of what the interval timer and the interrupt flag register look like as far as the programmer is concerned.

1. The interval timer can be looked upon as having a total of eight data registers. In the Junior Computer they have been given the names:

- CNTA, CNTB, CNTC, CNTD, CNTE, CNTF, CNTG and CNTH. Each of these data registers can be individually addressed and have been allocated locations 1AF4 . . . 1AF7 and 1AFC . . . 1AFF respectively.
2. The contents of the data registers can be altered by the processor by means of a write operation.
  3. Each data register has been assigned a specific division factor:
    - CLK1T = divide by 1; CNTA and CBTE
    - CLK8T = divide by 8; CNTB and CNTF
    - CLK64T = divide by 64; CNTC and CNTG
    - CLK1KT = divide by 1024; CNTD and CNTH
 The division factor is effectively the number of  $\Phi 2$  clock pulses that must pass before the contents of the programmable register (timer) are decremented by one.
  4. The interval timer is split into two sections of four data registers each. One is made up from the registers CNTA . . . CNTD and the other from registers CNTE . . . CNTH.
  5. If the processor enters data into one of the registers CNTA . . . CNTD, the interval timer will be started. The division factor will be determined by which of the registers is being used. During the write operation the timer interrupt flag will automatically be reset (b7 in the interrupt flag register = 0). In addition, the IRQ line will go high and the timer will be inhibited from causing an interrupt. The timer interrupt flag will be set after the timer has reached the time out (contents of the programmable register = FF).
  6. *IRQ mode:* If the processor enters data into one of the registers CNTE . . . CNTH, the interval timer will again be started. As above, the division factor will be determined by which of the registers is used, the interrupt timer flag will be reset and the IRQ line will go high. This time however, the timer will be able to cause an interrupt. After the time out has been reached, the timer interrupt flag will be set and the IRQ line will be taken low causing an interrupt request (provided, of course, that the interrupt flag in the status register of the CPU is reset: CLI).
  7. The computer is also able to read the instantaneous value of the contents of the programmable timer register. When the data register RDTDIS is read (address location 1AD4) the period of time remaining (before the time out) can be determined. This read operation will cause the timer interrupt request to be inhibited.

**Figure 12. The data and control registers of the interval timer and the edge detector.** The interval timer is started by entering information into one of the data registers CNTA . . . CNTH. The amount of time left before, or elapsed since, a time out can be established by reading one of the data registers RDTDIS or RD TEN. The former, along with data registers CNTA . . . CNTD, will place the interval timer in the polling mode and disable the interrupt request line. The latter, along with data registers CNTE . . . CNTH, will enable the interrupt request line. The state of the flags in the interrupt flag register can be examined by reading the data register RD FLAG. The four control registers EDETA . . . EDETD determine the polarity of the pulse to be detected on PA7. Control registers EDETA and EDETB place the edge detector in the polling mode and control registers EDETC and EDETD place the edge detector in the interrupt mode.



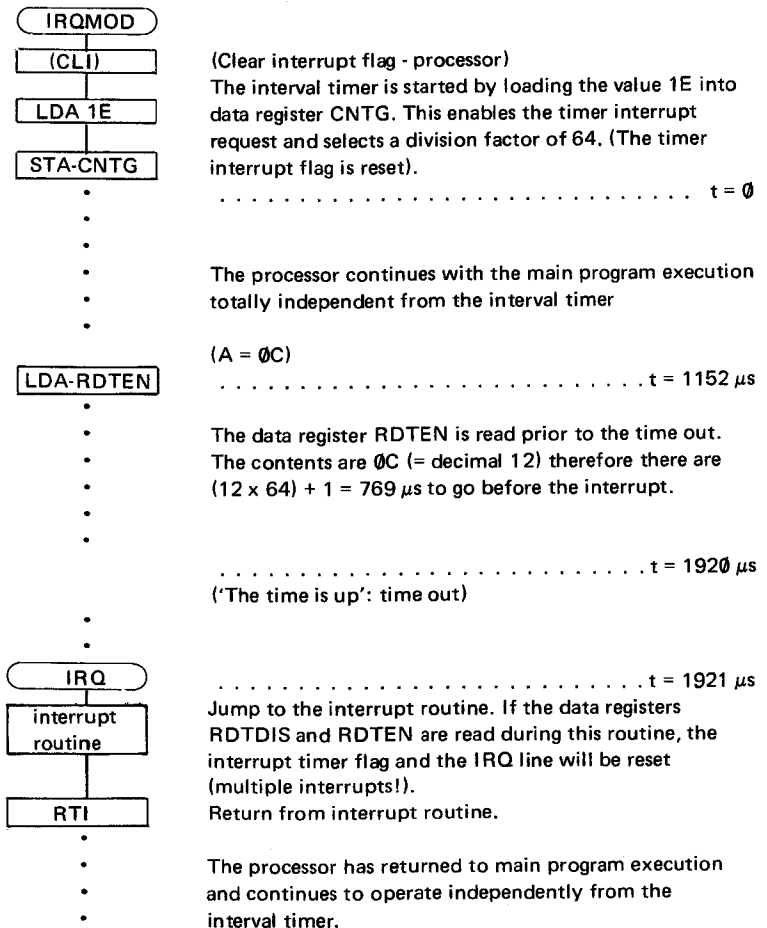
80915-6-12

On the other hand, when the data register RDTEN is read (address location 1ADC) the remaining period of time can again be determined, but this operation will enable the interrupt request. By reading the data registers RDTDIS and RDTEN the interval timer

can be switched from the polling mode (see section 8) to the interrupt mode or vice versa. The read operation will not alter the contents of the timer register, but will merely convey an idea of the state it is in at that particular moment. If this takes place before the time out, the division factor which was previously introduced will be maintained. When RDTDIS and RDTEN are read after the time out, the elapsed time (the period of time since the time out occurred) can be calculated by complementing the value read and adding one (two's complement).

If we now return to our previous example and load data register CNTG with an initial timer value of 1E (decimal 30) the division factor will be set at 64 and the interrupt will be enabled. The period of time between the start of the interval timer and the IRQ line going low (and the timer interrupt flag being set) will be  $(30 \times 64) + 1 = 1921 \mu s$ .

The operation of the interval timer and how its contents can be read in the *interrupt mode* are illustrated by the following 'flow chart':



LDA-RDTEN

(A = A4 = 10100100) = -92 decimal)

..... t = 2013 μs

(Since the interrupt 92 μs have passed, the two's complement of A4 is 5C (= 92 decimal)).

(The contents of the interval timer are decremented by one every microsecond until it is restarted by the processor).

8. *Polling mode*: The timer can also be operated in what is described as the polling mode. This means that the programmable timer register contents can be tested periodically to see whether a time out has occurred. In this mode of operation the interrupt request line is disabled by writing into one of the data registers CNTA . . . CNTD. The contents of the interval timer can then be checked by reading the data register RDTDIS (keeping the interrupt request line disabled). Alternatively, the contents of data register RDFLAG (address location 1AD5) can be examined to see whether the timer interrupt flag (in the interrupt flag register) has been set. Therefore, the properties of the interval timer can still be used to their full advantage while leaving the interrupt request line free for another (external) device.

The following example shows the same program sequence but this time in the polling mode. For this a new instruction will be introduced, which was not discussed in Book I: BIT. The BIT instruction enables certain bits in any (programmable) memory location to be tested. Therefore, the contents of the interrupt flag register can also be tested with this new instruction. (Actually, it's been around for some time). The general operation of the BIT instruction is as follows:

- \*  $A \wedge M$  The contents of the accumulator are ANDed bit by bit with the contents of the memory location. The previous contents of the accumulator are not altered by this instruction (similar to the CMP instruction). However, the Z-flag in the CPU status register is affected by the result of the AND operation ( $Z = 1$  if result = 0). Either zero page or absolute addressing can be used for the BIT instruction.
- \*  $M_7 \rightarrow N$  If the contents of the tested memory location are negative, b7 of that location will be 'one' and, after the BIT instruction, the N-flag in the CPU status register will be set. Effectively, b7 of the memory location is copied into the status register (N-flag).
- \*  $M_6 \rightarrow V$  If b6 of the tested memory location is 'one' the V-flag (overflow) in the status register will be set. Again, b6 is copied into the status register.

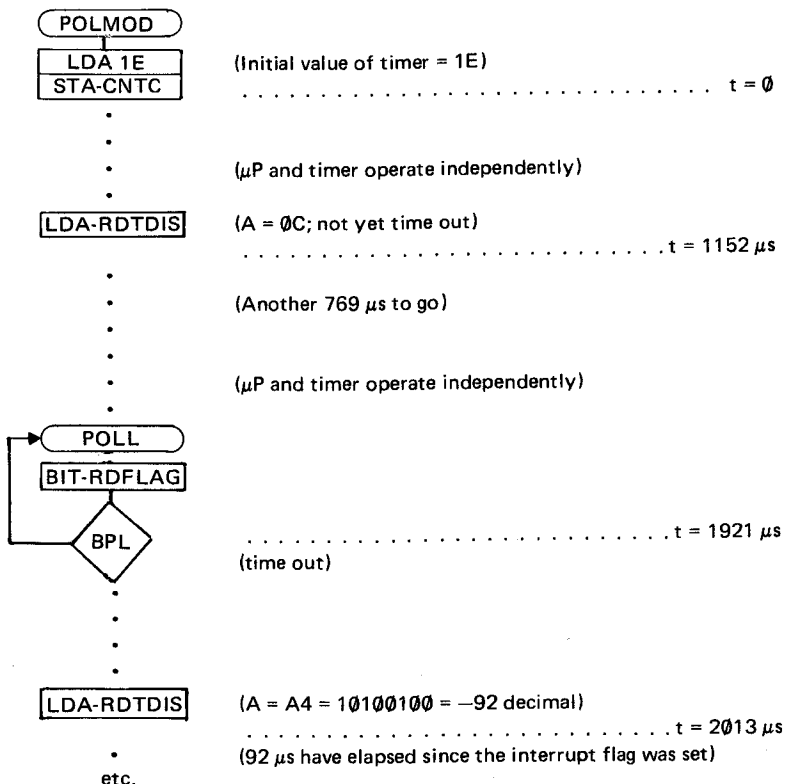
As we know, the interrupt flag register in the interval timer consists of two flags: b7 is the timer interrupt flag and b6 is the PA7 interrupt flag. The state of these flags can therefore be determined by reading data register

RDFLAG. By using the BIT instruction together with a suitable branch instruction only five memory locations are required for the whole procedure. The relevant branch instructions for testing these flags are listed below:

- a. to test the timer interrupt flag (timer in polling mode):  
 BPL: branch if N-flag = timer interrupt flag = 0  
 BMI: branch if N-flag = timer interrupt flag = 1
- b. to test the PA7 interrupt flag (PA7 in polling mode):  
 BVC: branch if V-flag = PA7 interrupt flag = 0  
 BVS: branch if V-flag = PA7 interrupt flag = 1

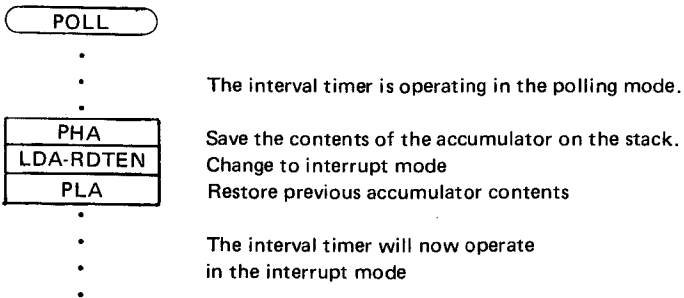
If the processor writes into one of the data registers CNTA . . . CNTD, the interval timer will be started, the interrupt flag will be reset and the interrupt request line will be disabled. The interval timer will then be in the polling mode.

If we use data register CNTC (instead of CNTG) in our previous example, the division factor will still be 64, but this time no interrupt is possible. To detect when a time out occurs, therefore, we need to examine the contents of the interrupt flag register to see when the interrupt flag (b7) is set. To do this, all that is required is to examine the contents of the data register RDFLAG as illustrated below:

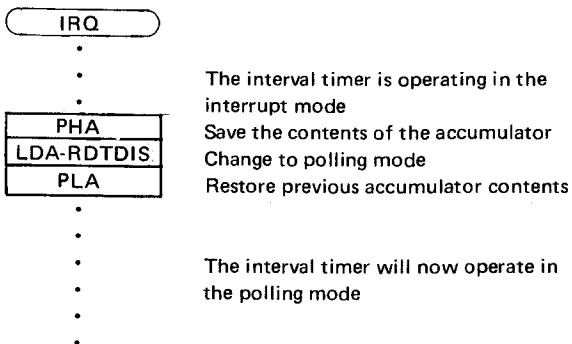


The interval timer is again loaded with the initial value of 1E. This means, as we now well know, that the timer interrupt flag will be set after 1921  $\mu$ s (CNTC has a division factor of 64). The first time that the data register RDTDIS is read the timer has not yet reached the time out therefore the timer interrupt flag will not be reset (as it has not yet been set!!). The processor and the timer will continue to operate independently until the loop POLL. Here, the data register RDFLAG is continually interrogated until the timer interrupt flag is set (1  $\mu$ s after the time out). As soon as the interrupt flag is set the processor will continue with the main program operation (again, independently from the timer). When the data register RDTDIS is read the second time, the timer interrupt flag will be reset and will remain so until the next time out (256  $\mu$ s) or until new information is entered into one of the data registers.

9. As we are well aware, the timer may operate either in the polling mode or in the interrupt mode. In some instances it may be desirable to alternate between the two modes of operation. This means, for example, that if the timer is operating in the polling mode, and the timer interrupt flag has not yet been set, the programmer can put the interval timer into the interrupt mode without having to start it afresh. This can be accomplished by storing the contents of the accumulator on the stack, reading the data register RD TEN and then replacing the contents of the stack in the accumulator. Thus:



It follows therefore, that the operation of the interval timer can also be changed from the interrupt mode to the polling mode:





In both cases, the contents of the programmable timer register are unaffected and the division factor will not alter when the data register is read.

10. *Reading from and writing to the interval timer*

a. *Reading the data registers RDTDIS, RD TEN and R DFLAG*

- The microprocessor is able to read the instantaneous value of the data registers RDTDIS and RD TEN. From this information it can calculate how much time is left before a time out occurs, or how much time has elapsed since the time out.
- Reading the data register RDTDIS causes the interval timer to be placed in the polling mode and disables the interrupt request line. The timer interrupt flag is set after a time out.
- Reading the data register RD TEN causes the interval timer to be placed in the interrupt mode. When a time out occurs, the timer interrupt flag is set and the IRQ line is pulled low to cause an interrupt request.
- By reading the data register R DFLAG, the processor can determine whether or not the timer interrupt flag has been set or not.
- The division factors 1, 8, 64 or 1024 are preset depending on which of the data registers CNTA . . . CNTH are written into. Reading any of the data registers RDTDIS, RD TEN or R DFLAG will not affect the division factor, if this is done before a time out.
- If one of the data registers RDTDIS or RD TEN is read after a time out, the timer interrupt flag and the interrupt request line will both be reset. The timer interrupt flag will not be reset, however, if the interrupt occurs at the same time the data register is read.
- When the interval timer causes an interrupt request, the microprocessor will branch to an interrupt (sub)routine. It is advisable to start this routine by reading the data register RDTDIS so that the timer interrupt flag and the interrupt request line are reset. This will prevent the same interrupt routine from being run when the processor returns to the main program.

b. *Writing into data registers CNTA . . . CNTH*

- Writing into one of the data registers CNTA . . . CNTD causes the interval timer to be started in the polling mode. The division factor is also established and will not change until the time out occurs.
- Writing into one of the data registers CNTE . . . CNTH causes the interval timer to be started in the interrupt mode. The division factor is again established and will not change until the time out occurs.
- When information is entered into one of the data registers CNTA . . . CNTH, the timer interrupt flag and the interrupt request line are both reset.

This concludes the description of the interval timer. We have now discovered how to use eleven data registers and the timer interrupt flag. We are also familiar with the two modes of operation: the polling mode and the interrupt mode. However, before we can put all this theory into practice, yet another important aspect of the PIA is still to be discussed: the edge detector.

## Edge detection: the PA7 mode

The peripheral interface adapter is capable of detecting pulses. The structure of the edge detector is very simple; it has one input and one output. Its input is port line PA7 and its output is the interrupt request line. The edge detector can be programmed to activate the interrupt request line when there is either a positive or negative pulse transition on the PA7 input line. When a pulse is detected, b6 in the interrupt flag register will be set. *If the microprocessor is to operate in conjunction with the edge detector, port line PA7 must be programmed as an input.*

The programmer has the option of preventing the edge detector from causing an interrupt request. As shown in figure 12, the edge detector consists of four 'control registers':

EDETA; address location 1AE4	polling mode
EDETB; address location 1AE5	
EDETC; address location 1AE6	interrupt mode
EDETD; address location 1AE7	

The control registers EDETA and EDETC are used when a negative going pulse is to be detected and registers EDETB and EDETD are used to detect positive going pulses. When control registers EDETA and EDETB are used the edge detector will be placed in the polling mode and the interrupt request line will be disabled. When control registers EDETC and EDETD are used the edge detector will be placed in the interrupt mode and the interrupt request line will be enabled.

Two common applications for the edge detector are shown in figure 13. The first of these uses port line PA7 as a serial data input and the second uses all eight of the port A input/output lines (programmed as inputs) to transfer a 7-bit ASCII code from a keyboard to the computer.

## Serial input

As we know from Book I, the majority of data manipulated inside the computer is transferred in parallel form (byte by byte). However, the computer is also capable of transferring data serially.

Serial data transfer is an economical method of conveying information and is absolutely necessary if the computer is to be connected to a printer, a video terminal or other such peripheral device. The most often used method of serial data transfer is the ASCII (American Standard Code for Information Interchange) code which, as can be seen from figure 13a, consists of the following components:

1. one start bit
2. eight data bits (b0 . . . b7)
3. one parity bit
4. two stop bits

More often than not, only seven of the eight available data bits are actually used. A complete breakdown of the ASCII code will be given in the Junior Computer Book III.

A certain bit time,  $t_2$ , is assigned to each bit in the serial data train. The time interval between each (expected) data bit is therefore constant. The duration of the start bit is  $1\frac{1}{2}$  times the duration of the bit time, and, as

there are two stop bits, the duration of the stop bits is twice that of the bit time.

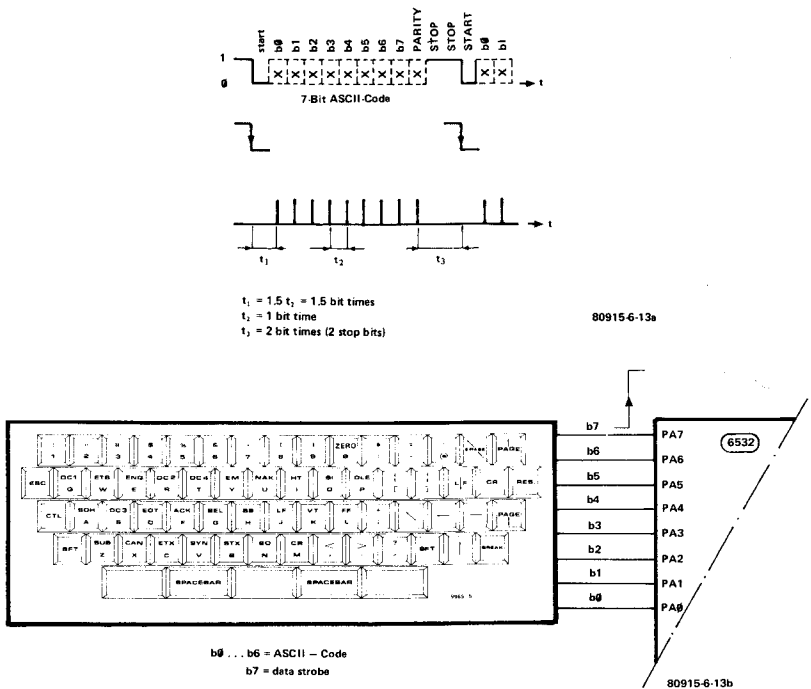
The transfer of serial data is always preceded by the start bit. This means that when the serial data input line to the particular device is high, the computer understands that there is no data to be input. However, when the serial input line goes low it means that information is on its way. This level transition can easily be detected by the edge detector of the peripheral interface adapter. In the given example the edge detector must be programmed to react to a negative going pulse transition (the start bit going low). Once the computer has recognised a start bit by means of the edge detector, it is no trouble at all for it to read in the serial data train:

1. The start bit is recognised (negative going pulse on port line PA7)
2. The computer waits for  $1\frac{1}{2}$  bit times and then 'tests' the signal level on PA7. In other words, it checks whether the logic level at the serial input is high or low. In this way it can assign the value 0 or 1 to the first data bit. One bit period later the second data bit will arrive at the serial input. Again the processor tests to see whether the logic level is high or low. This process is repeated until the last data bit (b7) is read into the computer. Each bit is tested halfway through a bit period.
3. One bit period after the last data bit is read the parity bit arrives at the serial data input. This bit is used for testing purposes and informs the processor whether the serial data stream that has just been transferred contains an even or odd number of ones or noughts. It is then a simple task for the computer to establish whether an error has occurred during the data transfer.
4. The serial data transfer is then ended by two stop bits. It should be noted that both stop bits have the opposite polarity to the start bit (start bit is low, stop bits are high).
5. The microprocessor will then wait for the next negative going pulse on the serial data input line (PA7). As soon as the edge detector receives this information the above procedure is repeated.

## Parallel input

As we know, the Junior Computer is also capable of reading parallel input signals at port A (or at port B for that matter). Figure 13b shows the connections required to transfer data from an ASCII keyboard to the computer. When a key is depressed the (ASCII) code for that key is output from the keyboard in parallel form. Seven of the available bits are used for the actual code. As soon as the key code is stable on the data lines b0 . . . b6, the keyboard will generate a 'strobe' pulse. The eighth data bit (b7) is used for the data strobe and when this line goes high the key code will be accepted by the computer (as this is the line that is connected to the edge detector). In other words, when a positive going pulse is detected on port line PA7 the stable key data can be read into the computer.

These two applications illustrate the operation and possible uses for the edge detector. When data is to be transferred between the computer and certain peripheral devices connected to it, the edge detector becomes absolutely indispensable. Several other uses for the edge detector will be described in Book III.



**Figure 13. Two practical examples of the possible uses of the edge detector: serial (a) and parallel (b) data entry into the Junior Computer via port A. It is clear that prior to using the edge detector, port line PA7 should be programmed as an input.**

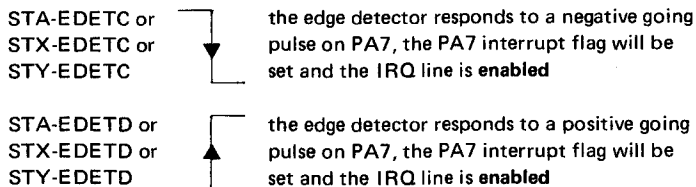
### Programming the edge detector

As far as software is concerned, the edge detector consists of four 'control' registers. As shown in figure 12, these are EDETA, EDETB, EDETC and EDETD. When data is entered into one of these control registers, the edge detector is programmed to react to a positive or negative going pulse on port line PA7 (EDETA and EDETC for negative going pulses, EDETB and EDETD for positive going pulses).

The edge detector operates in conjunction with the interrupt flag register inside the PIA. Here, bit 6 is the flag for the edge detector and is referred to as the PA7 interrupt flag. If, for example, the edge detector is programmed to react to a negative going pulse on PA7 and such a pulse occurs, the PA7 interrupt flag in the interrupt flag register will be set. Of course, if it is programmed to detect positive going pulses, the latter type will cause the same reaction.

Similarly, such pulses can be used to control the interrupt request line. If the programmer wishes to enable an edge detector interrupt request (by entering any bit pattern into one of the control registers EDETC or EDETD), the interrupt request line of the PIA will go low the moment the PA7 interrupt flag is set. Thus, the following similarities exist between





### Further comments on the PIA

As we already know, the timer and the edge detector operate in conjunction with the interrupt flag register. The timer interrupt flag (b7) and the PA7 interrupt flag (b6) are set after a time out and a pulse transition on PA7 respectively. If the programmer permits the timer or the edge detector to cause an interrupt request, the IRQ line will go low as soon as the timer interrupt flag or the PA7 interrupt flag is set. To prevent the same interrupt request from being caused several times, the IRQ line must be reset as soon as the interrupt is serviced (in the interrupt routine). When resetting the IRQ line, the following points should be borne in mind:

1. *The interval timer causes an IRQ:* If the programmer has enabled the interval timer to cause an interrupt request, the timer interrupt flag is set after the time out and the IRQ line will go low. The processor will now branch to an interrupt routine. At the start of this routine the IRQ line must be reset, so that the same timer IRQ cannot be caused after the return to the main program (RTI). *The timer interrupt flag and the IRQ line can be reset during the interrupt routine by writing into data registers CNTA . . . CNTH or by reading one of the data registers RDTDIS or RD TEN.* When one of the data registers CNTA . . . CNTH is written into, the interval timer starts afresh and defines whether the timer should cause a further interrupt request or not. When data registers RDTDIS or RD TEN are read, the interval timer does not start from scratch, but only defines whether the interrupt request is enabled or disabled.
2. *The edge detector causes an IRQ:* If the programmer has enabled the edge detector to cause an interrupt request, the PA7 interrupt flag is set after a certain pulse transition has been detected at the PA7 input and the IRQ line will go low. The processor will now branch to an interrupt routine. At the start of this routine the IRQ line must be reset, to prevent the same PA7 IRQ from being caused a second time once the processor has returned to the main program. *The PA7 interrupt flag and the IRQ line can be reset by reading the contents of the interrupt flag register RDFLAG.*
3. *The IRQ vector:* Memory locations 1A7E and 1A7F in the Junior Computer are reserved for the interrupt request vector. This means that the computer is able to determine or alter the IRQ vector during the course of a program. This enables the interval timer and the edge detector to be allotted their very own interrupt routines. By reading the interrupt flag register RDFLAG the processor can determine which of the two devices causes the interrupt request. If, for

example, the timer interrupt flag was set, the processor will set the IRQ vector to the start of the timer interrupt routine and otherwise to the start of the PA7 interrupt routine. When very high processing speeds are required, it is often necessary to incorporate several 'interlaced' interrupt routines. This aspect will be considered in greater detail in Book III.

4. *The reset line RES:* The Junior Computer is initiated when the reset line is taken low. This line also affects various registers inside the PIA. When the reset line (RES) goes low the following will happen:
  1. The contents of all the input/output registers are cleared. Thus, all port A and port B lines are declared inputs. Peripheral devices are now unable to inadvertently destroy the output registers PAD and PBD.
  2. No interval timer or PA7 interrupt request can be caused. Thus, it is impossible for the processor to branch to an interrupt routine – for which no IRQ vector has yet been set.
  3. The RES signal programs the edge detector to respond to a negative going pulse on PA7. During the reset, however, the PA7 interrupt flag may be accidentally set by a peripheral device. Before the edge detector is used therefore, it is good practice to reset the PA7 interrupt flag by reading the interrupt flag register RDFLAG.

This completes the discussion of the PIA. The following program examples show how to utilise the interval timer and the PIA. The edge detector has been deliberately omitted as it will be dealt with in the next book. Figure 14 shows the complete block diagram of the 6532 IC. It gives the internal structure of the PIA, the interval timer, the edge detector and the RAM memory. The software configuration of these components is shown in figure 15. The address locations required for the microprocessor to access the various PIA registers are also shown. Address line A7 is

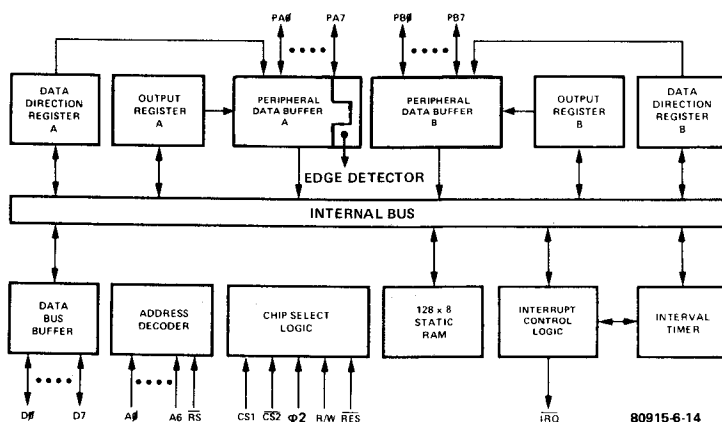


Figure 14. The block diagram of the multi-function peripheral interface adapter. The input/output ports, the data direction registers, the interval timer, the edge detector and the 128 bytes of RAM are all interconnected via the internal (data) bus.

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

80915-6-15

**Figure 15.** This table gives the information required to address the individual data and control registers of the PIA. Address line A7 is used to select between the 128 bytes of RAM and the functions of the interval timer and the edge detector.

used as the  $\overline{RS}$  line for the PIA ( $\overline{RS} = \overline{RAM\ Select}$ ). When this line is low the 128 RAM locations in the PIA can be addressed. If, however,  $\overline{RS}$  is high, the CPU can access the interval timer, the input/output registers or the edge detector. Therefore, address locations 1A00 ... 1A7F belong to the RAM memory and locations 1A80 ... 1AFF will address the various registers. It should be noted that not all the address locations from 1A80 ... 1AFF are actually used. This is because the address decoding pertaining to the 6532 chips is incomplete.

### Operating the interval timer in the interrupt mode

After all this theory concerning the interval timer etc, it is high time to put what we have learnt into practice. For this purpose we will write two programs: INPUT and REPEAT. The first of these two programs will make it possible to enter a tune into the Junior Computer with the aid of the keyboard given in figure 5. The tune can then be played back by means of the second program.

Since the remainder of this chapter serves to explain how to program the interval timer, we will refer to subroutines developed earlier for the PLAY program. The subroutines KEYIN, KEYVAL and EQUAL are of particular interest. We can also adopt the keyboard and loudspeaker interface without having to modify anything.

The problem to be solved here concerns the entering of a tune into the Junior Computer, which the computer will then repeat automatically. What information do we have to enter into the computer? Two factors are involved when playing a note:



1. The pitch or frequency of the note. In the PLAY program it was seen how each key that was depressed was assigned a separate frequency. The same principle also applies to the INPUT program.
2. If the computer is to repeat the entered melody it must know the length of time that each of the keys was depressed. Therefore, the duration of each note must also be ascertained. This can be measured quite simply by utilising the interval timer.

Thus, the computer must store both the frequency of the tone and the period of time that each key was depressed. Two memory locations must therefore be reserved for each note in the melody to be played. The frequency of the note depends on the value of the key. Since there are sixteen keys available, the value of each key will be between 0 . . . F. The interval timer will measure the time duration for which a key is depressed and will assign a value of between 00 . . . FF to it.

### The INPUT program

The entry routine INPUT must fulfil the following requirements:

1. When a key is depressed a tone corresponding to the value of that key should be heard from the loudspeaker.
2. Simultaneously, the computer must determine the length of time that the key was depressed for. Since two events are apparently happening at the same time, usual program techniques will no longer be valid. For this reason we will have to resort to using interrupt routines so that we are able to execute two parallel events on the Junior Computer.
3. Every key that is depressed must have a particular value assigned to it. This is specified by the computer during the KEYVAL subroutine.
4. It must be possible to scan the keyboard at a very high speed. This in fact poses no problems due to the subroutine KEYIN. Once a depressed key is detected, it must be debounced, which is taken care of by the (well known) DELAY routine.
5. When the key is released, the loudspeaker should go quiet and the value of the key together with the length of time it was depressed must be stored. These values must be stored in the memory locations 0100 . . . 01D8 of page 1. If this memory area happens to be filled, the computer must jump back to the monitor program and ignore the keyboard.
6. Before a melody can be stored in the Junior Computer, the 'melody memory' must be prepared. This involves filling the entire melody memory with the 'dummy' value 77 and setting a pointer to the initial are location.
7. The key value must be stored first followed by the length of time that the key was depressed. If, for instance, the keys 9, A, B and C (values) are depressed in that order, the following information must be stored in the melody memory:

Address:	Data:
0100	09 = key value
0101	WW = 00 . . . FF = duration of depressed key
0102	0A = key value

0103	XX	= 00 . . . FF = duration of depressed key
0104	0B	= key value
0105	YY	= 00 . . . FF = duration of depressed key
0106	0C	= key value
0107	ZZ	= 00 . . . FF = duration of depressed key
0108	77	= dummy value
0109	77	
.	.	
.	.	
.	.	
.	.	
01D7	77	
01D8	77	= end of melody memory

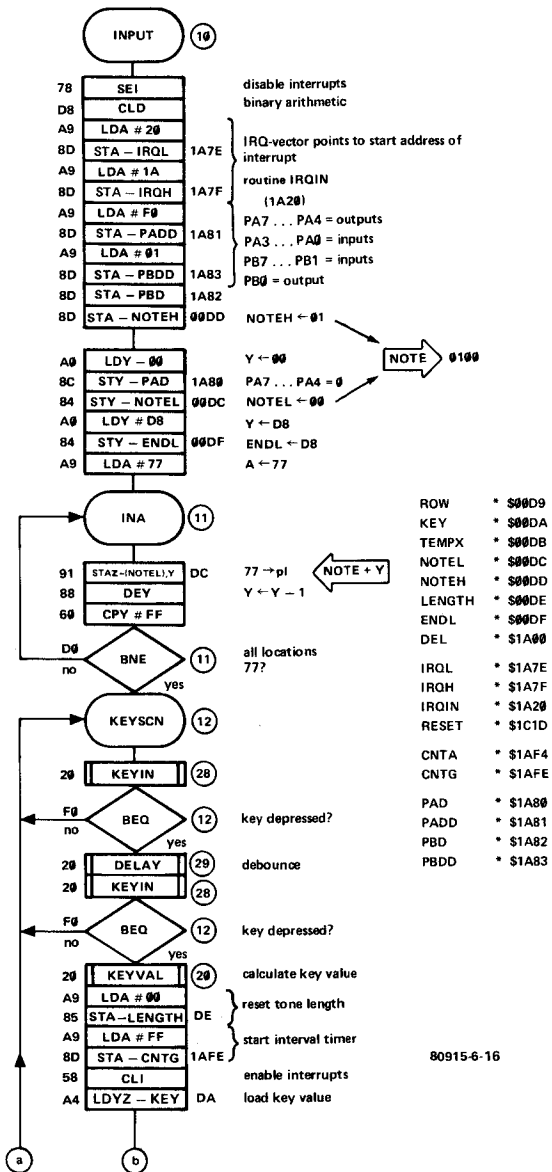
All the above requirements are met by the INPUT program, the flow chart of which is shown in figure 16. Initially, the program disables the interrupt request line with the instruction SEI. As a result, no accidental interrupts can be caused. The computer then sets the IRQ vector to point to address location 1A20, the start address of the interrupt routine IRQIN (figure 17a), which determines the length of time a key is depressed. We will return to that later.

Next, the input/output ports of the PIA are programmed. As shown in figure 5, the keyboard matrix is connected to port A. Lines PA7 . . . PA4 are programmed as outputs and lines PA3 . . . PA0 are programmed as inputs. The loudspeaker interface is connected to port line PB0. This line will therefore have to be programmed as an output.

Furthermore, the NOTE pointer for the melody memory (figure 17b) has to be set. This pointer indicates an address location on page 1, in which the value of the next key to be depressed is to be stored. Initially, therefore, this pointer will point to address location 0100. The end address of the melody memory is 01D8. If the NOTE pointer exceeds this address, the Junior Computer must jump back to the monitor program. Thus, the value \$D8 is stored in location ENDL. A comparison of the low order address byte of the NOTE pointer with the contents of ENDL will determine whether or not the memory area is full.

The computer has now reached the label INA. Here, the melody memory area from locations 0100 . . . 01D8 are filled with the dummy value of 77. The computer then scans the keyboard. If no key has been depressed, it will remain in the loop KEYSN - BEQ - KEYSN. If, however, the contents of the accumulator are not equal to zero following a return from the KEYIN routine, a key must have been depressed. The key is then debounced by the subroutine DELAY (figure 8b) and if the key is still depressed after this, the processor will jump to the subroutine KEYVAL (figure 6) where it will calculate the value of that key. Since there are 16 possible keys, the value will be in the range 00 . . . 0F. The calculated key value is then stored in location KEY after the return to the main program. The flow charts of all the subroutines involved are given in figures 6 and 8. They were described in detail in relation to the PLAY program.

And now the fun begins!! Once the value of the depressed key has been determined, the computer has to produce a tone from the loudspeaker. To



achieve this the processor uses the same look-up table (DEL) as it did for the PLAY program (see figure 10). This is done following the label TONE – this section of the program is identical to that of the program PLAY. Meanwhile the length of time the key is depressed must also be calculated.

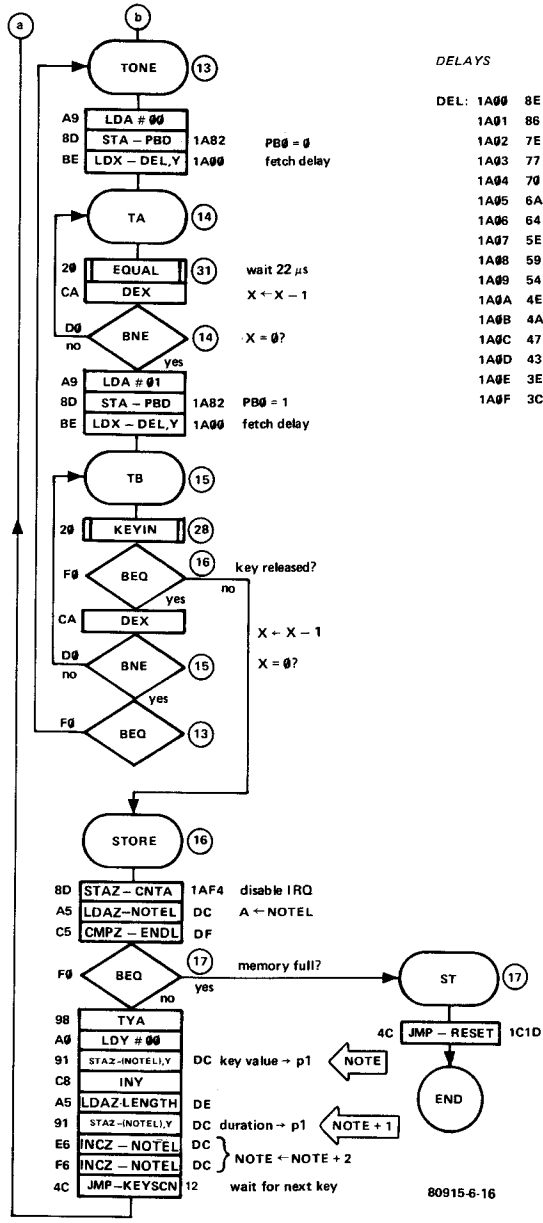
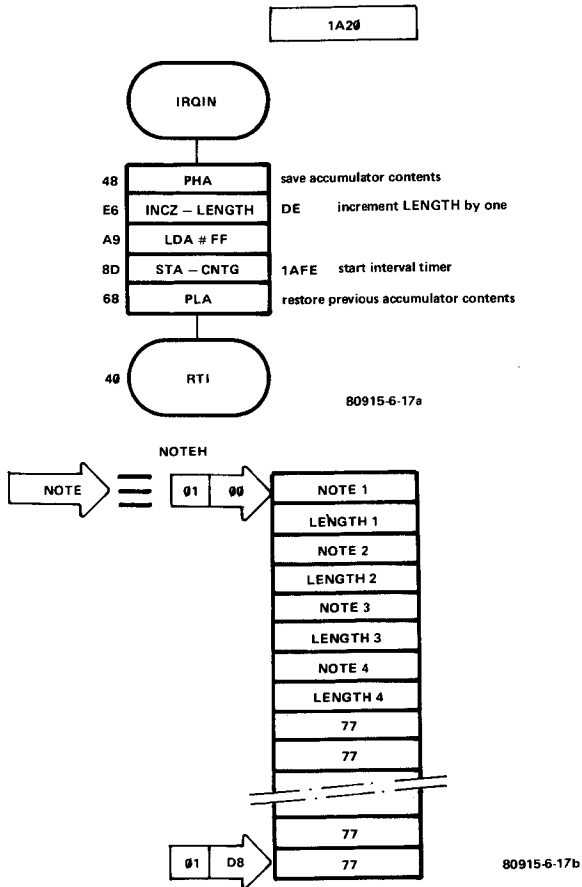


Figure 16. The flowchart of the INPUT program is very similar to that of the PLAY program given in figure 7. The main difference is that this time the played melody is stored in the memory of the Junior Computer.

This is done by means of interrupt programming. A few more instructions have been inserted between the jump to subroutine KEYVAL and the label TONE. These instructions perform the following operations:

\* The contents of memory location LENGTH will be equal to the length of time the key is depressed. Before the computer can produce a tone for the specific key, the contents of LENGTH must be reset (LDA 00, STAZ-LENGTH). The value \$ FF is then entered into the programmable timer register CNTG to start the interval timer. As we (should) know (by now), this will enable the interrupt request line and preset the division factor to 64. This means that the timer interrupt flag will be set and the interrupt request line will go low 16,321 microseconds after the data register CNTG was written into  $(255 \times 64) + 1 = 16,321 \mu s$ .



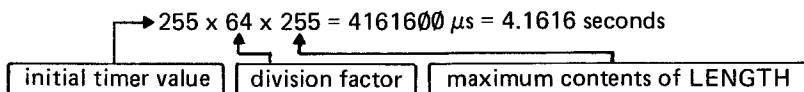
**Figure 17. The interrupt routine IRQIN (a) is used by the INPUT program to determine the length of time the key is depressed. Figure 17b shows how the key value and its duration are stored in the melody memory.**

When the interval timer is started the interrupt request line of the PIA is enabled. The next instruction in the program (CLI) clears the interrupt flag of the CPU and enables its interrupt request line. Therefore, if the interval timer causes an interrupt request the CPU will be able to acknowledge the interrupt.

Only now will the computer start producing a tone. For this it fetches the value of the depressed key from location KEY followed by the frequency of the note to be played from the look-up table DEL. The loudspeaker is then switched on and off in exactly the same way as for the PLAY program. While the tone is being produced the keyboard is scanned by the subroutine KEYIN to check whether the key is still being depressed. While the key remains depressed, the Junior Computer will continue to produce a tone.

As the interval timer operates independently from the CPU, the timer will cause an interrupt request after 16,321  $\mu$ s. The interrupt request line will then be pulled low and the computer will branch to the interrupt routine IRQIN (figure 17c). At the start of the interrupt routine the contents of the accumulator are saved on the stack (PHA) and then the contents of location LENGTH are incremented by one. The CPU then restarts the interval timer (LDA # FF, STA-CNTG) and enables the IRQ line once more. The interval timer will again operate independently from the CPU. Before returning from the interrupt routine the previous accumulator contents are restored (PLA).

Once the computer has returned to the main program it will recommence producing the tone until the key is released. After 16320  $\mu$ s the interval timer will have again reached a time out following which the computer will branch to the interrupt routine once more, thereby incrementing the contents of location LENGTH by one. The maximum duration that can be measured while a key is depressed is therefore:



(not counting the time required by the processor to service the interrupt routine). Thus, while producing a tone, the computer can simultaneously measure how long a key is depressed. The maximum duration can therefore be up to four seconds. If this period is exceeded, the processor will measure an incorrect time. No error will be reported (to keep the program as simple as possible).

Each time an interrupt request occurs the computer will leave the tone producing routine (TONE) to which it returns once the interrupt routine has been executed. The CPU then runs through the subroutine KEYIN to check whether the key has been released. If the key has been released, the processor will return to the TONE routine with a value of zero in the accumulator. Thus, the processor will stop producing the tone and will branch to the STORE routine. All the details concerning the key that has just been released will have been stored in memory locations KEY and LENGTH.

In the meantime, however, the interval timer could have caused another interrupt request which would lead to miscalculation of the length of time

the key was depressed. To prevent the interval timer from losing control of the situation after leaving the TONE loop, further interrupt requests are disabled. For this reason the data register CNTA is loaded with zero, although any of the data registers CNTA . . . CNTD could have been used instead. By writing into one of these data registers the interval timer is prevented from causing an interrupt request.

The computer then compares the low order address byte of the NOTE pointer with the contents of location ENDL to find out whether or not all of the melody memory area has been filled up. If so, the CPU will return to the monitor program. If not, the processor will transfer the contents of locations KEY and LENGTH into the melody memory. The dummy value of 77, which was present previously, will be overwritten by the data concerning the key. The CPU will then modify the NOTE pointer (2 x INCZ-NOTEL) so that it points to a new address where the value of the next key to be depressed will be stored. The computer will then wait until the next key is operated.

The memory area from 0200 . . . 03FF was chosen for the INPUT program. After editing the program the assembler is started with the ST key. However, before this can be done, and before the editor is started, the correct NMI vector data will have to be stored in locations 1A7A and 1A7B. These two locations should contain the start address of the assembler program (1F51).

After the INPUT program has been assembled, the look-up table (DEL) and the interrupt routine will have to be loaded, into the Junior Computer. When everything has at last been loaded, the INPUT program can be started. Provided, of course, that the keyboard is connected. As in the PLAY program, a tune can then be played on the keyboard. The only difference being that this time the tune will be stored in the melody memory. The details for the entry of the complete INPUT program are given below:

key:		display:	comments:
RST		XXXX XX	call monitor
AD	0 0 E 2	00E2 XX	
DA	0 0	00E2 00	} BEGAD = 0200
+	0 2	00E3 02	
+	F F	00E4 FF	} ENDAD = 03FF
+	0 3	00E5 03	
AD	1 A 7 A	1A7A XX	
DA	5 1	1A7A 51	} NMI vector = 1F51 (start assembler with ST)
+	1 F	1A7B 1F	
AD	1 C B 5	1CB5 20	start editor
GO		77	editor running
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

key:		display:	comments:
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 DD	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 DC	84 DC	STYZ-NOTEL
INPUT	A 0 D 8	A0 D8	LDY #D8
INPUT	8 4 DF	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 DC	91 DC	STA-(NOTEL),Y
INPUT	8 8	88	DEY
INPUT	C 0 FF	C0 FF	CPY #FF
INPUT	D 0 1 1	D0 11	BNE to 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 to 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 to 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 DE	85 DE	STAZ-LENGTH
INPUT	A 9 FF	A9 FF	LDA #FF
INPUT	8 D F E 1 A	8D FE 1A	STA-CNTG
INPUT	5 8	58	CLI
INPUT	A 4 DA	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 to 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 to STORE (label 16)
INPUT	C A	CA	DEX
INPUT	D 0 1 5	D0 15	BNE to TB (label 15)
INPUT	F 0 1 3	F0 13	BEQ to 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



key:		display:	comments:
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 to 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
			last INPUT instruction
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 to 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 #KEYA (label 21)
INPUT	8 6 D B	86 DB	STXZ-TEMPX (00DB)
INPUT	8 5 D A	85 DA	STAZ-KEY (00DA)
INPUT	A 2 0 0	A2 00	LDX #00
INPUT	F F 2 2 0 0	FF 22 00	label 22: KEYB
INPUT	4 6 D A	46 DA	LSRZ-KEY (00DA)
INPUT	9 0 2 3	90 23	BCC to ROWA (label 23)
INPUT	E 8	E8	INX
INPUT	E 0 0 4	E0 04	CPX #04
INPUT	D 0 2 2	D0 22	BNE to KEYB (label 22)
INPUT	F 0 2 0	F0 20	BEQ to KEYVAL (label 20)
INPUT	F F 2 3 0 0	FF 23 00	label 23: ROWA
INPUT	A 5 D B	A5 DB	LDAZ-TEMPX (00DB)
INPUT	C 9 0 3	C9 03	CMP #03
INPUT	D 0 2 4	D0 24	BNE to ROWB (label 24)
INPUT	8 A	8A	TXA
INPUT	4 C 2 7 0 0	4C 27 00	JMP-KEYC (label 27)
INPUT	F F 2 4 0 0	FF 24 00	label 24: ROWB
INPUT	C 9 0 2	C9 02	CMP #02
INPUT	D 0 2 5	D0 25	BNE to ROWC (label 25)
INPUT	8 A	8A	TXA

key:	display:	comments:
INPUT 1 8	18	CLC
INPUT 6 9 0 4	69 04	ADC #04
INPUT D 0 2 7	D0 27	BNE to KEYC (label 27)
INPUT F F 2 5 0 0	FF 25 00	label 25: ROWC
INPUT C 9 0 1	C9 01	CMP #01
INPUT D 0 2 6	D0 26	BNE to ROWD (label 26)
INPUT 8 A	8A	TXA
INPUT 1 8	18	CLC
INPUT 6 9 0 8	69 08	ADC #08
INPUT D 0 2 7	D0 27	BNE to KEYC (label 27)
INPUT F F 2 6 0 0	FF 26 00	label 26: ROWD
INPUT C 9 0 0	C9 00	CMP #00
INPUT D 0 2 0	D0 20	BNE to KEYVAL (label 20)
INPUT 8 A	8A	TXA
INPUT 1 8	18	CLC
INPUT 6 9 0 C	69 0C	ADC #0C
INPUT F F 2 7 0 0	FF 27 00	label 27: KEYC
INPUT 8 5 D A	85 DA	STAZ-KEY (00DA)
INPUT A 9 0 0	A9 00	LDA #00
INPUT 8 D 8 0 1 A	8D 80 1A	STA-PAD
INPUT 6 0	60	RTS
INPUT F F 2 8 0 0	FF 28 00	label 28: KEYIN
INPUT A D 8 0 1 A	AD 80 1A	LDA-PAD
INPUT 2 9 0 F	29 0F	AND #0F
INPUT 4 9 0 F	49 0F	EOR #0F
INPUT 6 0	60	RTS
INPUT F F 2 9 0 0	FF 29 00	label 29: DELAY
INPUT A 0 F F	A0 FF	LDY #FF
INPUT F F 3 0 0 0	FF 30 00	label 30: DELA
INPUT 8 8	88	DEY
INPUT D 0 3 0	D0 30	BNE to DELA (label 30)
INPUT 6 0	60	RTS
INPUT F F 3 1 0 0	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 0	60	RTS

Now that the INPUT program, including all its subroutines, has been entered into the Junior Computer, a check can be carried out to ensure that there were no errors during entry:

key.	display:	comments:
SEARCH F F 1 0	FF 10 00	INPUT begins at label 10
SKIP	78	
SKIP	D8	
SKIP	A9 20	
SKIP	8D 7E 1A	

When we are certain that everything has been entered correctly we can assemble the INPUT program. This can be carried out quite simply by pressing the ST key. The next task is to enter the interrupt routine IRQIN (figure 17a) and the look-up table DEL (figure 10). When entering the interrupt routine the editor must be used, but this routine is not to be assembled!!

key:		display:	comments:
AD	0 0 E 2	00E2 xx	
DA	2 0	00E2 20	} BEGAD → 1A20
+	1 A	00E3 1A	
+	7 9	00E4 79	} ENDAD → 1A79
+	1 A	00E5 1A	
AD	1 C B 5	1CB5 20	start address of editor
GO		77	editor running
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 0	40	RTI
RST			

} 128 bytes  
PIA-RAM

That takes care of the interrupt routine, and as for the look-up table:

key:		display:
AD	1 A 0 0	1A00 xx
DA	8 E	1A00 8E
+	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

Now that everything has been entered, the program can be started:

key:		display:	comments:
AD	0 2 0 0	020078	INPUT program start address
GO			

The INPUT program can be left in one of three (preferred) ways:

\* by depressing the RST key

- \* by filling the entire available melody memory with data; the processor will then jump to the monitor program
- \* by depressing the ST key. In this instance the NMI vector will have to point to the start address of the following REPEAT program (0000). This means that once a melody has been entered, it can be played back as often as required by pressing the ST key.

*Warning!* The Junior Computer should not be switched off after entering the INPUT program, as the REPEAT program is still to be entered. Although INPUT and REPEAT are two entirely separate programs, the one is not much use without the other.

## The REPEAT program

The REPEAT program is capable of reproducing the melody entered into the memory during the INPUT program. In other words, the Junior Computer is capable of repeating performances to perfection!! The following requirements must be met for the computer to be able to replay a stored melody:

1. The computer is to convert the value of the key stored in memory into a note. During this process it will make use of the look-up table DEL.
2. Immediately following the key value of each note in the melody memory is the code for the length of time the note was held. With the aid of an interrupt routine, this value will have to be converted back into a definite time period. For this operation the interval timer will be in the interrupt mode.
3. To prevent the notes from 'running into' each other a short interval should be inserted between each note. The Junior Computer uses the interval timer to obtain this pause without any outside help. The interval timer will be operating in the polling mode for this operation.
4. As soon as the entire melody has been replayed, the computer should jump back to the monitor program.

The flowchart for the REPEAT program is given in figure 14. As it is very similar to the INPUT program in many ways it can be summarized very briefly.

To start with, the processor is again inhibited from acknowledging an interrupt request. The IRQ vector is then set to point at the start address of the interrupt routine. In this instance it points to address location 1A30 (for the INPUT program it pointed to address location 1A20) so that both programs can be run independently.

Since the loudspeaker interface is connected to port line PB0, this line must be programmed as an output. The NOTE pointer must also be set, initially it points to address location 0100, where the first key value of the melody to be repeated is stored. The next instruction (STA-CNTA) simply clears the timer interrupt flag and resets the interrupt request line. This is just a safety precaution and ensures that the IRQ line will not be low when the next instruction is executed (CLI = clear interrupt flag = enable interrupt request) which will enable the CPU to accept interrupt requests.

Now the microprocessor will have arrived at label FETCH. This is where the interval timer is started. The initial value of the programmable register

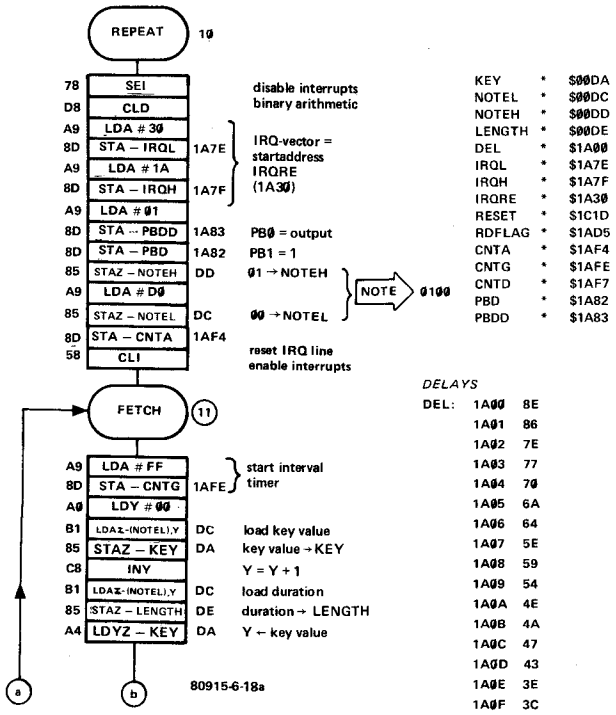
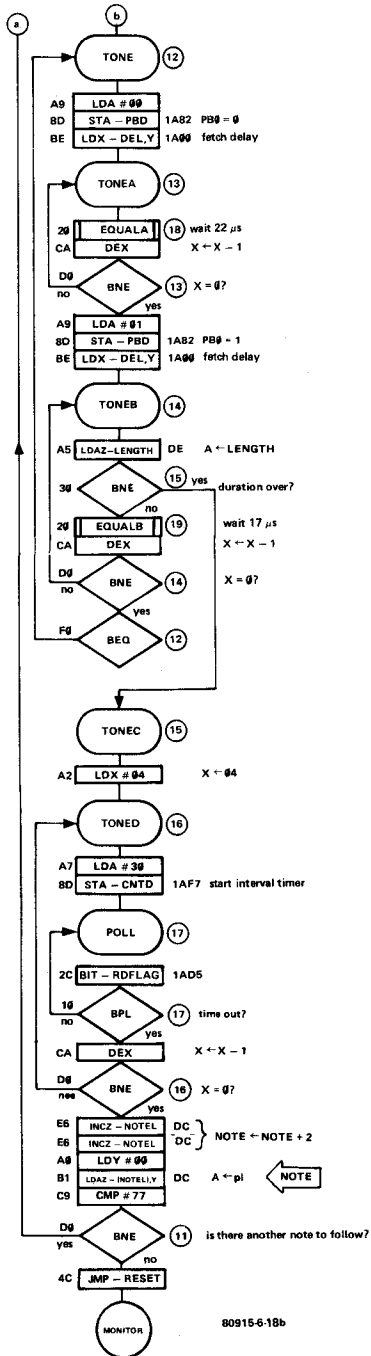


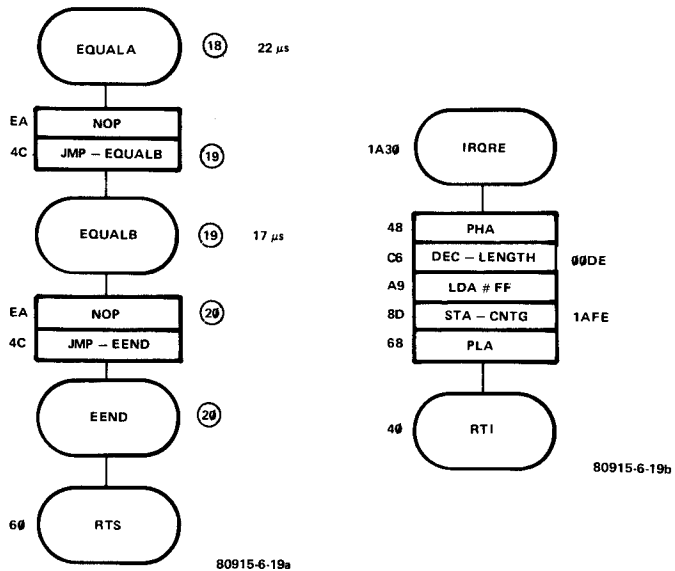
Figure 18. The flowchart of the REPEAT program. This program enables the Junior Computer to replay a previously stored melody. It uses the same look-up table (DEL) as the INPUT program.

is \$ FF and the division factor is again 64, as during the INPUT program. By writing into the data register CNTG the interrupt request line of the PIA will be enabled. The interval timer will now operate independently of the main routine until it reaches a time out (16320 μs). The computer therefore has plenty of time (relatively speaking!) to fetch the key value and the length of time the key was depressed from memory. The key value is then converted into a frequency with the aid of the look-up table DEL. This occurs at label TONE.

The loudspeaker is switched on and off at labels TONEA and TONEB respectively. The loop times during this process should be exactly 27 μs as during the INPUT program. For this reason two delay loops are required instead of just the one for the INPUT program: EQUALA provides a delay of 22 μs and EQUALB provides a delay of 17 μs.

The computer will continue to produce a tone until the interval timer reaches a time out and causes an interrupt request. The processor will then branch to the interrupt routine at address location 1A30 (the IRQ vector was set to this location at the start of the program). This interrupt routine performs the exact opposite operation of that in the previous program:







**Figure 19. The subroutines EQUALA and EQUALB (a) are required by the REPEAT program to ensure that the frequencies of the replayed notes are exactly the same as the previously 'recorded' ones. The interrupt routine IRQRE (b) determines the length of time the note is to be played.**

this time the contents of location LENGTH are decremented by one. During the REPEAT program therefore, the processor produces a note (from label TONE) until the contents of LENGTH become negative, as a result of several interval timer interrupts. When this situation occurs, the program will branch to label TONEC. Before the next note is fetched from the melody memory and played, the computer must insert a short interval so that the individual notes do not 'run' into each other. To achieve this the interval timer will have to be used once more, this time in the polling mode. At label TONED the interval timer is prevented from causing a further interrupt request. The instructions LDA #30 and STA-CNTD start the interval timer in the polling mode, reset the interrupt flag in the interrupt flag register and set the division factor to 1024. By using data register CNTD the interval timer is unable to cause an interrupt request after the time out. Once the interval timer has been started, the processor will remain in the delay loop POLL-BPL-POLL until the timer interrupt flag is set (after  $4 \times (48 \times 1024) + 1 = 196,612 \mu s$ ). This procedure is repeated four times, as the X register is decremented by one after each time out. This means that the total time spent in the delay loop is increased by a factor of four to give a total delay in the region of 0.2 seconds. After this time the computer increments the NOTE pointer by two so that it points to the location of the next note to be fetched and played. Before fetching this note however, the processor tests to see whether or not the NOTE pointer

has already arrived at the end of the melody. This can be detected by the presence of the dummy character 77. Once all the notes stored in the melody memory have been played, the Junior Computer will return to the monitor program.

Shown below are the steps involved in order to enter the REPEAT program from the hexadecimal keyboard (mustn't get confused must well!). It should now be fairly easy to assemble the program on page 0 (start address 0000), all the steps involved have been covered thoroughly. Finally, a well known melody has been provided for readers who may suffer from tone-deafness — have fun with it . . .

key:		display:	comments:
AD	00 E 2	00E2 xx	
DA	00	00E2 00	
+	00	00E3 00	 BEGAD 0000
+	E0	00E4 E0	 ENDAD 00E0
+	00	00E5 00	
AD	1 A 7 A	1A7A XX	} NMI-vector = 1F51 (start assembler with ST)
DA	5 1	1A7A 51	
+	1 F	1A7B 1F	
AD	1 C B 5	1CB5 20	start address of editor
GO		77	editor running
INSERT	F F 1 0 0 0	FF 10 00	label 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 DD	85 DD	STAZ-NOTEH
INPUT	A 9 0 0	A9 00	LDA #00
INPUT	8 5 DC	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 DC	B1 DC	LDA-(NOTEL),Y
INPUT	8 5 DA	85 DA	STAZ-KEY
INPUT	C 8	C8	INY
INPUT	B 1 DC	B1 DC	LDA-(NOTEL),Y
INPUT	8 5 DE	85 DE	STA-LENGTH
INPUT	A 4 DA	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 to 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
INPUT	A 5 D E	A5 DE	LDAZ-LENGTH
INPUT	3 0 1 5	30 15	BMI to label TONEC (label 15)
INPUT	2 0 1 9 0 0	20 19 00	JSR-EQUALB (label 19)
INPUT	C A	CA	DEX
INPUT	D 0 1 4	D0 14	BNE to TONEB (label 14)
INPUT	F 0 1 2	F0 12	BEQ to TONE (label 12)
INPUT	F F 1 5 0 0	FF 15 00	label 15: TONEC
INPUT	A 2 0 4	A2 04	LDX #04
INPUT	F F 1 6 0 0	FF 16 00	label 16: TONED
INPUT	A 9 3 0	A9 30	LDA #30
INPUT	8 D F 7 1 A	8D F7 1A	STA-CNTD
INPUT	F F 1 7 0 0	FF 17 00	label 17: POLL
INPUT	2 C D 5 1 A	2C D5 1A	BIT-RDFLAG
INPUT	1 0 1 7	10 17	BPL to POLL (label 17)
INPUT	C A	CA	DEX
INPUT	D 0 1 6	D0 16	BNE to TONED (label 16)
INPUT	E 6 D C	E6 DC	INCZ-NOTEL
INPUT	E 6 D C	E6 DC	INCZ-NOTEL
INPUT	A 0 0 0	A0 00	LDY #00
INPUT	B 1 D C	B1 DC	LDA-(NOTEL),Y
INPUT	C 9 7 7	C9 77	CMP #77
INPUT	D 0 1 1	D0 11	BNE to FETCH (label 11)
INPUT	4 C 1 D 1 C	4C 1D 1C	JMP-RESET (monitor; RESET = 1C1D)
INPUT	F F 1 8 0 0	FF 18 00	label 18: subroutine EQUALA
INPUT	E A	EA	NOP
INPUT	4 C 1 9 0 0	4C 19 00	JMP-EQUALB (label 19)
INPUT	F F 1 9 0 0	FF 19 00	label 19: EQUALB
INPUT	E A	EA	NOP
INPUT	4 C 2 0 0 0	4C 20 00	JMP-EEND (label 20)
INPUT	F F 2 0 0 0	FF 20 00	label 20: EEND
INPUT	6 0	60	RTS

Before assembling the program it should be verified:

<b>key:</b>	<b>display:</b>
SEARCH F F 1 0	FF 10
SKIP	78
SKIP	D8
SKIP	A9 30
SKIP	8D 7E 1A

The majority of the finger work is now complete. All that remains is to enter the interrupt program IRQRE. As with the previous interrupt routine, this routine must not be assembled:

key:		display:	comments:
AD	0 0 E 2	00E2 XX	
DA	3 0	00E2 30	
+	1 A	00E3 1A	 1A30
+	7 9	00E4 79	
+	1 A	00E5 1A	 1A79
AD	1 C B 5	1CB5 20	start address of editor
GO		77	editor running
INSERT	4 8	48	PHA
INPUT	C 6 D E	C6 DE	DECZ-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 0	40	RTI
RST			

All that needs to be done now is play a tune. First we start the INPUT program:

key:		display:	comments:
AD	0 2 0 0	0200 A9	start address of the INPUT program
GO		blank	

The display will remain blank until the processor returns to the monitor program. To enter a short melody:

key:		display:	comments:
0	2 4 5 7 7 9 9 9	blank	as these keys are
9	7 5 5 5 5 4 4 2		depressed, their value and
2	2 2 0		duration are stored in the
			melody memory

To replay the melody now that it has been entered:

key:		display:	comments:
AD	0 0 0 0	0000 78	start address of the REPEAT program
GO		blank	the display will remain blank until the
		0000 78	entire melody has been repeated;
			depressing the GO key will start the
			REPEAT program once more

This brings us to the end of this chapter. We have discovered how to program the interval timer, the edge detector and the input/output ports of the peripheral interface adapter. We are even able to amaze our relatives and friends by playing musical (?) tunes on the computer. By the way, a source listing of all the programs mentioned in this chapter is given in the appendix at the back of this book.

This chapter will serve as a useful reference when it comes to connecting peripheral devices such as printers, video terminals (elekterminal) etc. to the Junior Computer. Book III will introduce a cassette interface for the system which will include the software required to store programs on (and

retrieve them from) standard audio cassettes. Again, the PIA will be used to control the tape recorder and to produce the FSK signal. The Junior Computer will then evolve into a real mini-computer and will be well on the way to becoming a complete personal computer which will be capable of understanding high level languages such as BASIC. However, before that point is reached, we must understand completely how all the routines contained in the monitor program, including the editor and the assembler, work and how they can be incorporated into user programs.

# The Monitor program

## Basic 'housekeeping' software

For the programmer to be able to communicate with the Junior Computer, the machine must be capable of receiving and transmitting information in a form that is understandable to both. The computer must be able to decode information entered from the keyboard and output the processed data to the display. Therefore, the computer needs to be programmed to recognise various input and output parameters. This is where the monitor software comes in.

By monitor software is meant a program, usually stored in ROM (Read Only Memory), which provides the user with all the control functions required to operate the system satisfactorily. A monitor program typically contains a number of (sub)routines which perform such chores as program loading, debugging and general 'housekeeping'.

The monitor program of the Junior Computer contains a large number of subroutines which, for instance, indicate address and data information on the display, decode the data and command keys and execute the particular command function encountered (AD, DA, +, PC and GO). One of the advantages of the monitor program is the fact that the programmer is able to incorporate one or more of its subroutines into his/her own program without having to develop them first.

Furthermore, the IRQ and NMI vectors are stored in RAM, so they can be changed during the main program if required. This is essential if more than one interrupt routine is to be used with a single interrupt vector. All this (and more) will be discussed in the course of this chapter.

## A review of the monitor

The monitor program of the Junior Computer is stored in EPROM. In computer jargon such a program is a 'resident', inasmuch as it is permanently available — as soon as the power supply is turned on. The monitor program requires a few memory locations in page zero to be kept 'free' for its own use. These locations contain data which the computer must refer to frequently and (if necessary) update in order to carry out certain tasks. You may recall from Book 1 that many programs ended with the break (BRK) instruction. When the computer encountered this instruction the contents of the internal CPU registers were stored in page zero. This was so that the programmer could examine these registers before the Junior Computer returned to the monitor program and altered their contents.

Basically, the Junior Computer monitor program can be compared to a delay loop. When the computer is in this loop, it periodically scans the keyboard (waiting for a key to be depressed) and the contents of the three display buffers (POINTH, POINTL and INH) are multiplexed to the display.

If the programmer presses one of the five command keys, the computer will perform the operations corresponding to these keys. If, for example, the AD key was depressed, the computer will interpret the next data entry (keys 0 . . . F) as being an address. If the DA key was depressed, the following data will be stored in the address location shown on the display. You should be totally familiar with the functions of the other command keys from Book 1.

There are a number of ways in which to 'jump' to and from the monitor program:

### 1. Jumping to the monitor program

- \* Via the RST key: the Junior Computer will then jump into the monitor program and will immediately initialise the two input/output ports of the peripheral interface adapter. After this the computer will wait in a 'loop' until a key is depressed. The Junior Computer may now be 'spoken to' via the command keys AD, DA, +, PC and GO by the data keys 0 . . . F.

- \* Via a non-maskable interrupt: the monitor program may also be accessed via the NMI vector. If the NMI vector is set to point at address location 1C00, the computer will save the contents of the internal CPU registers. The computer will then enter a loop where it once again scans the keyboard and the display. Again, it will wait until one of the above mentioned keys is depressed. The PIA will not be initialised if the NMI vector is pointing to address location 1C00.

- \* Via an interrupt request: the monitor program may also be entered via the IRQ vector. This is, however, only possible when the interrupt flag in the status register of the CPU is reset (CLI). If the IRQ vector is pointing to address location 1C00, the situation will be identical to that of the NMI vector above.

- \* Via the BRK instruction: as we know from Book 1, the BRK instruction utilises the IRQ vector. If this instruction is to be used to enter the monitor program, the IRQ vector must be pointing to address location

1C00. The Junior Computer will then store the contents of the internal CPU registers and will enter a loop as in the cases of RST, NMI and IRQ.

\* Via the JSR instruction: the programmer is able to incorporate the subroutines of the monitor program into his/her own program. The RTS instruction at the end of each subroutine ensures that the processor will return to the user program upon completion of the subroutine.

\* Via the JMP instruction: the programmer may include a 'jump to monitor' instruction at the end of his/her program. By doing this the Junior Computer display will light when the end of a user program is reached as an indication to the programmer that the processor has completed the allotted task. If a JMP instruction is to be used to enter the monitor program, the jump should always lead to address location 1C1D (JMP-1C1D or 4C 1D 1C)). Jumping to this address will initiate a reset sequence and the PIA will again be programmed to scan the keyboard and display. Here too the Junior Computer will wait in a loop until a key is operated.

## 2. Jumping from the monitor program

\* Via the GO key: the most common method of leaving the monitor program is to depress the GO key. After this the processor will start operating from the address shown on the display. The display will then go blank.

\* Via a non-maskable interrupt: the processor can also leave the monitor program by means of the NMI vector. If the interrupt routine concludes with an RTI instruction, the monitor program will be returned to upon completion of the routine. If, however, there is no return to the monitor program, the input/output lines of the PIA should all be programmed as outputs immediately after leaving the monitor program to ensure that the display does not light inadvertently. This can be accomplished quite simply via the following program sequence:

(PHA) (save accumulator contents)

LDA 00

STA-PBD all display cathodes are 'off'

STA-PAD all display segments are 'off'

(PLA) (restore previous accumulator contents)

It should be noted that the contents of the accumulator need only be saved if the accumulator is to be used by the interrupt routine. This is why the instructions PHA and PLA are shown in parentheses.

\* Via an interrupt request: the processor can also leave the monitor program by means of the IRQ vector, provided that the interrupt flag in the CPU status register is reset. The situation will then be the same as that for the non-maskable interrupt.

As a reminder, the NMI and IRQ vectors are stored in page 1A of the Junior Computer memory. They have been allocated the following address locations:

NMIL: \$ 1A7A

NMIH: \$ 1A7B

IRQI: \$ 1A7E

IRQH: \$ 1A7F

The contents of the above memory locations determine where the pro-

cessor branches to after an IRQ, NMI or BRK instruction. Both interrupt vectors can be set either manually or by the program itself. So far we have been recapping on chapter 3 of Book 1 to make sure that you will fully understand the following section.

### The basic flowchart of the monitor program

The basic flowchart of the monitor program is shown in figure 1. It is composed of the usual flowchart symbols. The monitor includes three inputs and two outputs. The flowchart does not, however, allow for the monitor to be exited after an interrupt.

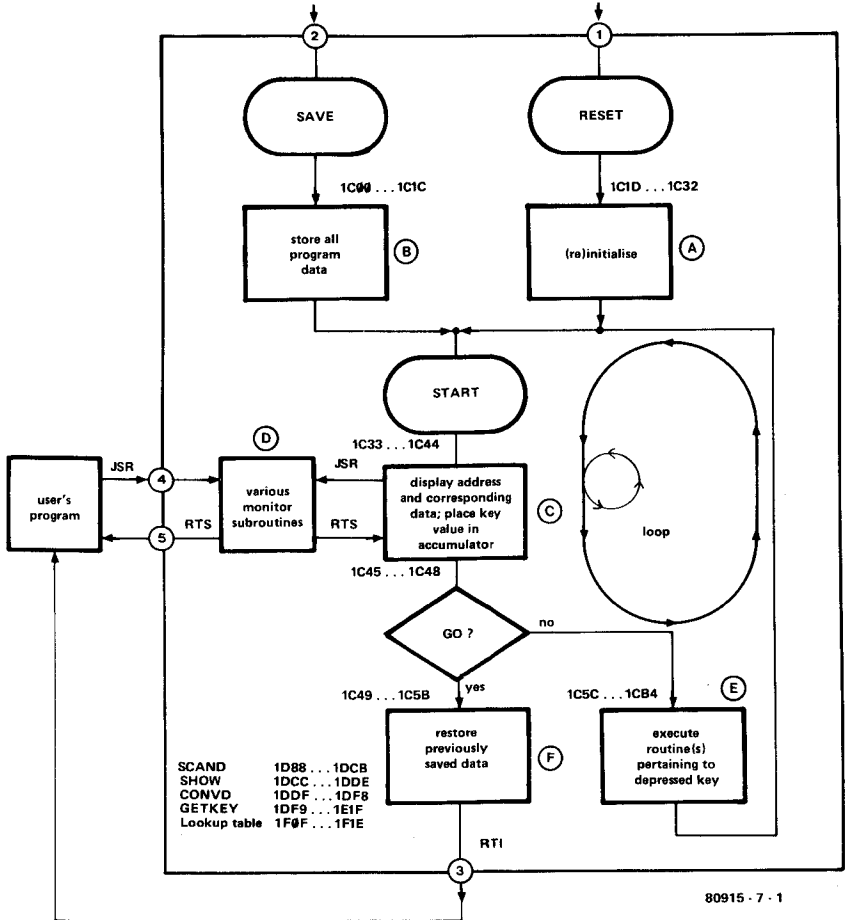


Figure 1. The basic flowchart of the Junior Computer monitor program. The various entry and exit points are clearly indicated (1 . . . 5). The address locations shown in the figure correspond to those given in the program listing at the back of this book.

### The ins and outs of the monitor program

**Input ①** leads to the label RESET. The CPU will enter the monitor program at this point if the RST key is depressed. The RESET label can be found at address location 1C1D. From this point on the processor will initiate the PIA and reset the stack pointer. The CPU status register is also preset — exactly how will be discussed later. What is important to know at this stage is that between the labels RESET and START the Junior Computer is programmed to control the display and scan the keyboard. In addition, the computer will operate in the binary mode (CLD) and the interrupt flag in the status register will be set. The Junior Computer will therefore not acknowledge any interrupt requests.

Summary: There are various methods of entering the monitor program via input ①:

- by depressing the RST key
- by means of a JMP instruction: JMP-1C1D
- via a non-maskable interrupt: the NMI vector points to address location 1C1D.
- via an interrupt request (if the interrupt flag is reset): again, the IRQ vector should point to address location 1C1D.
- via the BRK instruction: the BRK instruction utilises the IRQ vector. If this instruction is to be used to jump to the monitor program, the IRQ vector must point to address location 1C1D.

**Input ②** will lead to the label SAVE. This routine starts at address location 1C00. Between the two labels SAVE and START, the contents of the internal CPU registers are stored in page zero at the following address locations:

PCL at address location 00EF

PCH at address location 00F0

status register at address location 00F1

user stack pointer at address location 00F2

accumulator at address location 00F3

Y register at address location 00F4

X register at address location 00F5

**Warning!!** The monitor program should only be entered via input ② with the aid of one of the interrupt vectors. Only then can the CPU registers PCH, PCL and the status register be saved in the correct manner.

The stack pointer will then also be pointing to the correct address location as if the monitor program had been left via the GO key. Note: If the user program alters the contents of either of the PIA data direction registers, the input/output lines will not be re-initialised immediately!

Summary: There are various methods of entering the monitor program via input ②.

- by depressing the ST key: this causes a non-maskable interrupt. The NMI vector must therefore be pointing to address location 1C00. The NMI vector is stored at locations 1A7A and 1A7B.
- via the STEP mode: this also uses the NMI vector. Many of the programs given in Book 1 were run in the STEP mode. In this mode the individual program instructions will be executed one after the other each time the STEP key is operated. The processor will carry out the instruction being shown on the display and will then save the contents of



all the CPU registers in page zero. The next instruction will then appear on the display. Before the Junior Computer can operate in the STEP mode, the NMI vector must be pointing to address location 1C00 (label SAVE). This step-by-step procedure will be dealt with in greater detail later on.

- via a non-maskable interrupt caused by a peripheral device (such as a printer): here too the NMI vector must point to address location 1C00.
- via an interrupt request (as long as the interrupt flag in the status register is reset — 1 = 0): in this instance the IRQ vector must be pointing to the start of the SAVE routine (1A7F = 1C, 1A7E = 00).
- **not** via the BRK instruction: the BRK instruction should not be used to jump repeatedly into the monitor program via input ②. Just to be on the safe side, we should **never** branch to the monitor program via a BRK instruction and input ② as this could easily cause the wrong program counter value to be stored on the stack (PC + 2) and the stack itself to exceed its limits.

**Input ④** leads to various subroutines contained in the monitor program. The operator may incorporate these into his/her own programs by calling them up with a jump-to-subroutine (JSR) instruction. Each of the subroutines ends with instruction RTS (return-from-subroutine) to ensure that the processor always returns to the user's main program (output ⑤). This brings us up to date on all the tasks which the Junior Computer performs between the labels RESET and START and between the labels SAVE and START. From the START label onwards the Junior Computer enters a loop which can only be exited by operating the GO key.

Once in the loop the computer will call up various monitor subroutines and scan the keyboard and display. If a key is depressed, the processor will check to see whether it was the GO key. If it wasn't, the Junior Computer will remain in the loop. If a command key was depressed, the particular command will be carried out; if a data key was depressed the value of the key will appear either in the address or data section of the display (depending on which command key was depressed previously). The same procedure will then begin all over again at label START.

If some time during the loop the GO key is depressed, the processor will exit from the monitor program via the RTI (return-from-interrupt) instruction. In other words, the monitor program would normally be left via output ③.

### The STEP mode

Any discussion of the monitor program must of course include a word or two on the STEP mode. In this mode the hardware and the software of the Junior Computer work 'hand in glove'. To fully understand this mode of operation we will refer to the circuit diagram of the Junior Computer which was given in chapter 1 of Book 1 (figure 4). This shows that pin 7 of the microprocessor (IC1) is connected to one input of the NAND gate N5. The other input to this gate is connected to the chip select signal K7. The output of the gate, when switch S24 is closed, is connected to the non-maskable interrupt line of the microprocessor. When the Junior Computer is operated in the STEP mode the LED (D2) above the STEP/GO key will be lit.

An address location containing an instruction will now be shown on the

display. If the programmer wishes to execute that particular instruction all he/she needs to do is depress the STEP key. The processor will then leave the monitor program via output ③ and start operating from the address shown by fetching the instruction from memory. While the computer is fetching the instruction, the SYNC output of the microprocessor (pin 7) will go high. At this time the EPROM (1C2) is not being accessed therefore the CS line (K7) will also be high. This means that the output of N5 will go low to cause a non-maskable interrupt. The processor therefore executes the instruction currently shown on the display and then services the interrupt. The computer will then jump to address location 1C00 (label SAVE) via the NMI vector.

The processor will now have entered the monitor program once more. This means that the EPROM will be addressed and line K7 will be low. No further NMI will occur as the output of N5 will now be high.

During the SAVE routine the contents of all the internal CPU registers are stored in page zero and the program counter is loaded with the address location of the next instruction to be executed. The new contents of the program counter are then loaded into the display buffers POINTH and POINTL. After the START routine the computer will display the next instruction and its corresponding address. If the STEP key is then pressed once more the complete procedure described above will be repeated.

## **The interrupt vectors and the BRK instruction**

Up to now we have only been interested in where to store the NMI and IRQ vectors so that the CPU is able to service an interrupt. Now it is time to find out exactly how the processor fetches these interrupt vectors from memory locations 1A7A, 1A7B and 1A7E, 1A7F. The complete procedure is shown in detail in figure 2. The memory map to the left of the diagram can be split into two sections. The upper section consists of RAM locations, where the operator's program is stored. As you know, this is made up of pages 0, 1, 2, 3 and the 1/8 k of RAM in the PIA (page 1A). The lower section is where the monitor program is stored in the EPROM. To clarify the interrupt procedure in the Junior Computer and to keep the diagram as simple as possible only the most important locations have been selected. All the various possibilities of how to jump into and exit from the monitor program are given to the right of figure 2. By monitor program we mean the section contained in the EPROM that we dealt with in Book 1 and the six memory locations for the NMI, reset and IRQ vectors at addresses 1FFA...1FFF. The following points provide all the necessary information about these vectors:

### **1. The RST key**

When the RST key is depressed the processor examines the contents of address locations 1FFC and 1FFD to obtain the reset vector. These locations permanently point to address 1C1D (they are stored in the EPROM). The processor will therefore branch to the RESET label shown in figure 1 (output ④). This same label can however also be accessed by jumping from the user's main program via a JMP instruction.

### **2. Non-maskable interrupt**

As far as the Junior Computer is concerned, a non-maskable interrupt can

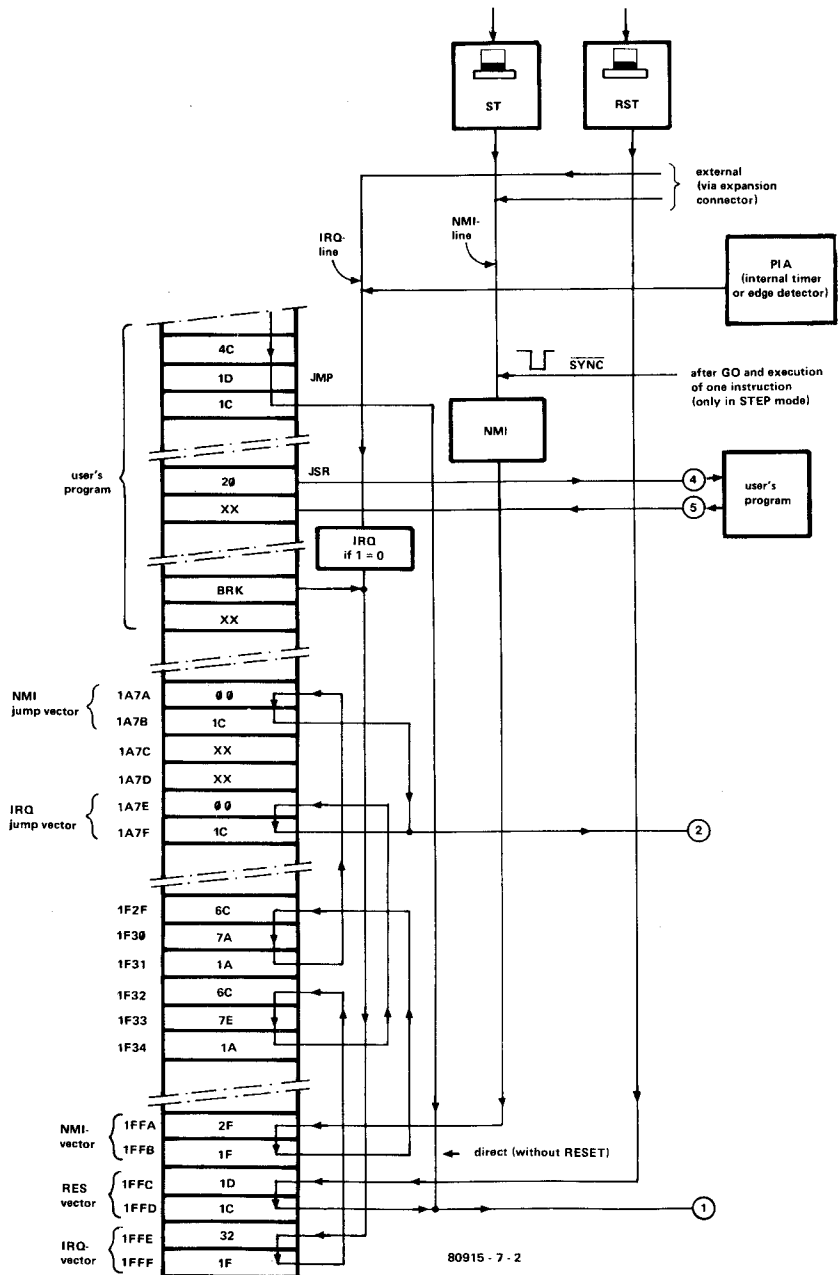


Figure 2. Diagrammatic representation of the interrupt procedure of the monitor program. The points marked 1, 2, 4 and 5 correspond to those given in figure 1.

be initiated in three different ways:

- \* by depressing the ST key
- \* by a peripheral device connected to the NMI line of the control bus. This line is fed to the external world via the expansion connector.
- \* by the output of N5 going low when the computer is in the STEP mode.

Whenever the NMI input line of the microprocessor is taken low (no matter how), the computer will perform a set procedure (see figure 2).

The processor will acknowledge the interrupt upon completion of the current instruction's execution. Following the interrupt acknowledge, the processor disables further interrupts by setting the interrupt flag in the status register. The CPU then pushes the contents of all the internal registers onto the stack (page zero). The program counter is then loaded with the NMI vector which will be fetched from memory locations 1FFA and 1FFB. In the case of the Junior Computer, this vector points to address location 1F2F. The processor will therefore jump to this address and carry out the instruction it finds there, which is an indirect jump instruction. As we know from Book 1, an indirect jump causes the processor to obtain the absolute address from the contents of the indirect address location. As the instruction at locations 1F2F . . . 1F31 is 6C 7A 1A (jump indirect to address location 1A7A), the processor will effect a jump to the address contained at locations 1A7A and 1A7B and from there will start the interrupt routine.

Figure 2 shows that locations 1A7A and 1A7B contain 00 and 1C respectively. This means that the Junior Computer will service the interrupt from location 1C00, which, as we know, is the address of the SAVE routine in the monitor program (input ②). As address locations 1A7A and 1A7B are in fact RAM locations, the interrupt vector can point to any address (not necessarily in the monitor program).

### **3. Interrupt request**

Before the processor can acknowledge an interrupt request, the interrupt flag in the status register will have to be reset (CLI). An interrupt request can be initiated in four different ways in the Junior Computer:

- \* by a peripheral device connected to the IRQ line of the control bus. This line is fed to the external world via the expansion connector.
- \* by means of an interval timer IRQ (see chapter 6)
- \* by means of an edge detector IRQ
- \* by the BRK instruction

Whenever the IRQ input line of the microprocessor is taken low (provided the interrupt flag is reset), the computer will perform a set procedure. The programmer however, must ensure that the interrupt request line is reset before returning from the interrupt routine, so that the processor is unable to service the same interrupt request more than once (see chapter 6). Figure 2 again illustrates the various steps taken during the IRQ procedure.

The processor will acknowledge the interrupt request upon completion of the current instruction's execution. Following the interrupt acknowledge, the processor disables further interrupt requests by setting the interrupt flag in the CPU status register. The CPU then pushes the contents of all the internal registers onto the stack (page zero). The program counter is

then loaded with the IRQ vector which will be fetched from memory locations 1FFE and 1FFF.

In the case of the Junior Computer, this vector points to address location 1F32. The processor will therefore jump to this address and carry out the instruction it finds there, which is an indirect jump instruction. As the instruction at locations 1F32...1F34 is 6C 7E 1A (jump indirect to address location 1A7E), the processor will effect a jump to the address contained at locations 1A7E and 1A7F and from there will start the interrupt request routine.

Figure 2 shows that locations 1A7E and 1A7F contain 00 and 1C respectively. This means that the Junior Computer will once again jump to the start of the SAVE routine (input ②). As before, the IRQ vector can point to any (addressable) location.

#### **4. The BRK instruction**

Although the BRK instruction was used in many of the program examples given in Book 1, this useful instruction was not fully described. All the more reason to do so now:

Imagine that the processor is manipulating a program which the operator has entered into the computer. At the end of the program the processor encounters a BRK instruction (op-code 00). As we know, the BRK instruction utilises the IRQ vector, that is to say, the following procedure will be exactly the same as that for an interrupt request (see point 3). Before fetching the IRQ vector, the B flag in the status register is set and the contents of the internal registers are pushed onto the stack. Note that in this instance the low order byte of the program counter is incremented twice before being stored on the stack. This means that if, for example, the BRK instruction was located at address 022A, the saved value of the program counter will be 022C **not 022B!** Therefore extreme care must be taken when using this instruction.

One advantage however, is that the BRK instruction can be used to 'debug' a suspect program (section). When the operator decides that a particular program is not functioning correctly, the BRK instruction can be used to overwrite the first byte of an existing instruction. The program under test will then operate normally up to this point, where the processor will branch to the address location pointed to by the IRQ vector. As the (absolute) IRQ vector is stored in RAM, the BRK instruction can be used to jump to any part of addressable memory. In figure 2 the IRQ vector (contents of locations 1FFE and 1FFF) is pointing to address location 1F32. The computer will therefore execute the indirect jump instruction it finds there and will continue operation from location 1C00 (start of the monitor program).

#### **5. The interrupt flag versus the break flag**

When the processor acknowledges an interrupt it automatically sets the interrupt flag in the status register. When executing a BRK instruction it automatically sets the break flag in the status register.

Why the need for these flags? Simple. Both the interrupt request and the BRK instruction utilise the IRQ vector. If there were no flags there would be no way to determine which of the two operations caused the interrupt in the first place. If, for example, several visual display units are connected to a central computer and are operated via the two interrupt lines (NMI

and IRQ), several interrupt routines would have to be nested. If the interval timer is also being used, the number of possible nested interrupt routines would increase even more. As the Junior Computer has 16 input/output lines it is theoretically possible to connect as many as 16 peripheral devices to these lines simultaneously. To detect whether an interrupt was caused by hardware or software, the status register can be examined to check whether or not the break flag is set. This can be done quite simply by including the following series of instructions at the start of the initial interrupt routine:

```
PLA      load contents of status register
PHA      restore onto stack
AND 10  mask break flag
BNE      branch to break routine if set
·
·
·
```

This interrupt routine could then determine which of the peripheral devices was responsible for the interrupt (presuming of course that it was not the BRK instruction).

An important difference between the interrupt flag and the break flag is that the interrupt flag can be set or reset during a program, but there are no instructions to set or reset the break flag. The latter is set when a BRK instruction is encountered and is only reset when the interrupt flag is set following an interrupt request.

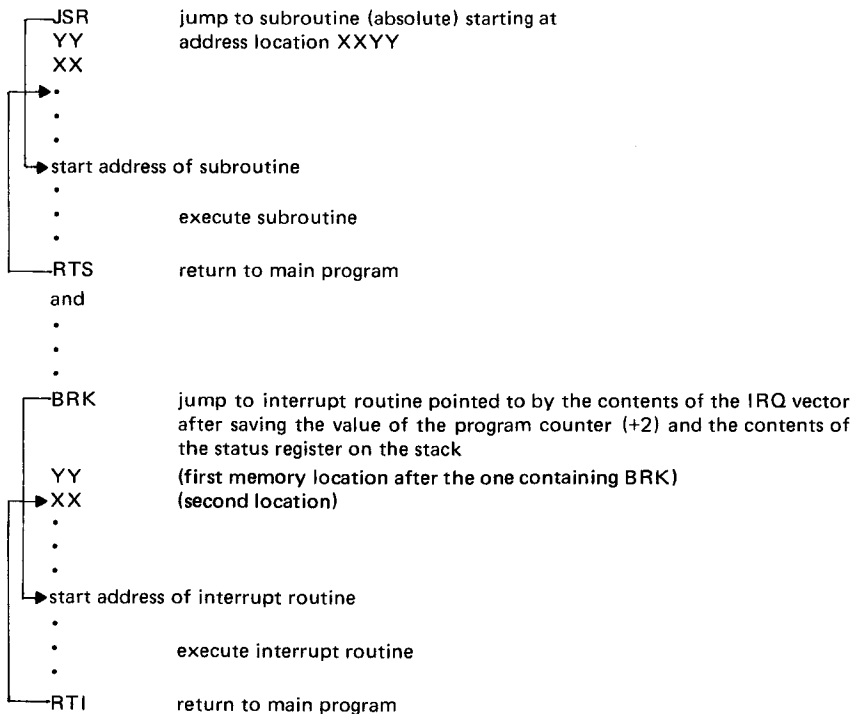
### Using the BRK instruction for debugging

When debugging a program, it is quite feasible that the operator frequently needs to utilise the BRK instruction to branch to an interrupt. This can be compared to the jump-to-subroutine (JSR) instruction, but in this instance the total instruction length is shortened to one byte (instead of three). For this reason, the value for the program counter contained on the stack will correspond to the address location two bytes higher than that immediately following the BRK instruction. This does not usually cause any problems during the debugging as, once the interrupt routine has been completed, the processor will return to the main program to carry out the instruction following the one which was 'patched' over by the BRK instruction, provided it is a two-byte instruction.

old code	0236	LDA (AD)
	0237	3F
	0238	01
	0239	next opcode
patched code	0236	BRK (00)
	0237	NOP (EA)
	0238	NOP (EA)
	0239	next opcode

Thus the following analogies can be made between the JSR and BRK instructions:

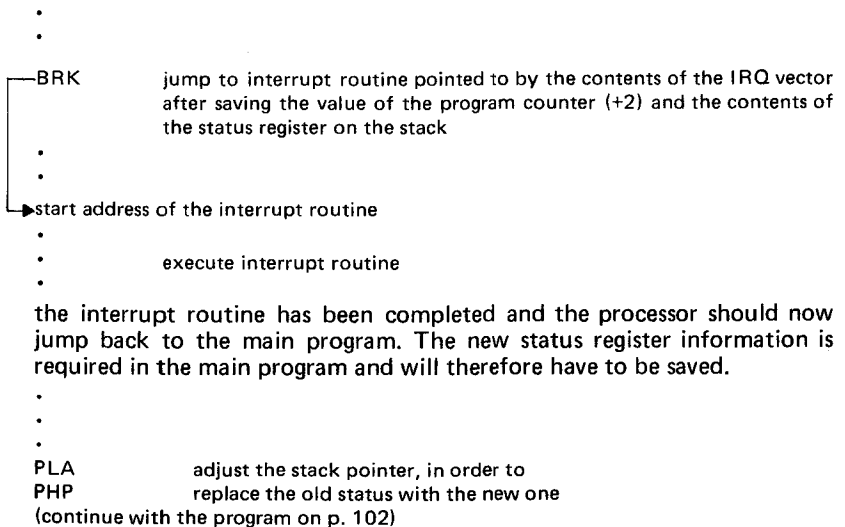
·  
·  
·



If, however, the operator wishes the computer to continue main program execution from the address location immediately following the BRK instruction, the value of the program counter will have to be altered **before** the return-from-interrupt instruction is encountered. This can be accomplished by including the following program section:

PLA	fetch status from stack
STA-MEM	into MEM
PLA	fetch low order PC (PCL)
STA-MEM+1	into MEM+1
PLA	fetch high order PC (PCH)
STA-MEM+2	into MEM+2
SEC	set carry
LDA-MEM+1	subtract one from PCL
SBC # 01	
STA-MEM+1	
LDA-MEM+2	subtract one from PCH
SBC # 00	if corrected PCL = FF
PHA	corrected PCH back to stack
LDA-MEM+1	corrected PCL via A
PHA	back to stack
LDA-MEM	status via A
PHA	back to stack
RTI	return to main program

It may well transpire that the contents of the status register are altered during the course of the interrupt routine. If the operator wishes to use this new information in the main program — provided the old status register is not required — the above sequence can be modified so that only the new contents of the status registers are saved:



The new contents of the status register can now be examined by the operator during the execution of the main program. This concludes the description of all the inputs and outputs of the monitor program. The rough flow chart (figure 1), the BRK instruction and the interrupts have all been explained in detail. The BRK instruction and the interrupt structure of the 6502 microprocessor enable programs to be tested and corrected both quickly and efficiently, even though the programs are getting longer and longer! The next step is to examine the monitor program in greater detail — instruction by instruction.

## The monitor program

As can be seen from figure 1, the monitor program contains three major routines: RESET, SAVE and START. These routines can be evaluated by examining each instruction in turn. The flow charts and diagrams of the monitor program enable the beginner to understand just how the Junior Computer operates. There is, of course, another reason for describing the monitor program in detail: by studying the flow charts etc. the newcomer can assimilate his/her thought processes with those of professional programmers. This educational aspect should be borne in mind when reading the following passages.

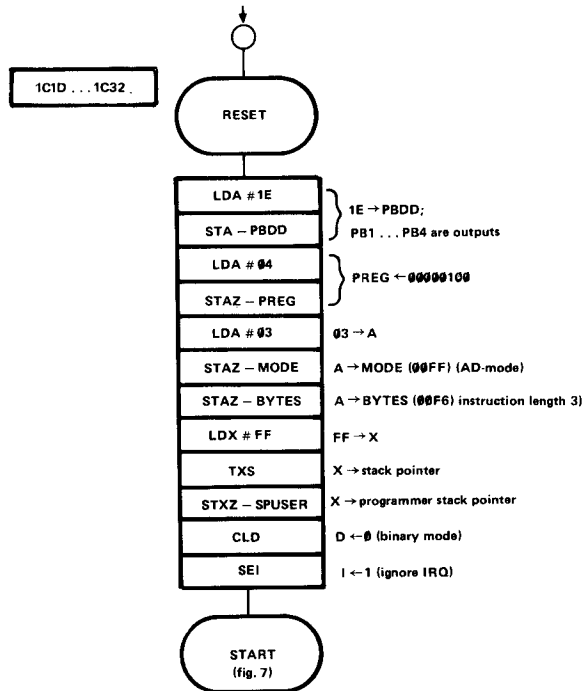


## RESET

Before the Junior Computer is able to communicate with the operator, via the keyboard and display, it needs to be brought into a certain initial condition. This is carried out by means of the RST key. As we already know from chapter 6, the peripheral interface adapter first has to be programmed to provide the correct input and output lines for the keyboard and display. Internal CPU registers, such as the stack pointer and the status register also have to be initialised.

When the RST key is depressed the CPU 'instinctively' examines the contents of address locations 1FFC and 1FFD to ascertain the reset vector which is, of course, address 1C1D. The processor will therefore jump to this location and arrive at the label RESET (point 1 in figure 1).

As can be seen from figure 3, the processor then defines which of the I/O lines of port B are to be programmed as outputs. By storing the data 1E in the port B data direction register (PBDD) port lines PB1 . . . PB4 become outputs. Hopefully, you will remember from book I that certain of the memory locations contained on page zero are reserved for the purpose of saving the contents of the internal CPU registers. One such location is called PREG. This is where the contents of the processor status register are



80915 - 7 - 3

Figure 3. The RESET-routine is executed when the Junior Computer is (re)initialised.

saved. If the monitor program is left by depressing the GO key immediately after the RST key (see figure 1), the contents of memory location PREG must have a particular value. The reason for this will become clear later.

Initially, 04 is stored in location PREG. This corresponds to the interrupt flag being set and all the other flags in the status register being reset. The processor then enters a positive number (03) into the address location MODE. If the contents of location MODE are not equal to zero the Junior Computer will operate in the AD (address) mode, if however the data held there is equal to zero, the DA (data) mode will be employed. In addition, the processor enters the value 03 into location BYTES. This means that three bytes are to be indicated on the six displays: two address bytes and a single data byte.

At this stage it is useful to remember chapter 5 where programs were entered into the computer with the aid of the editor. In this instance, the display has a variable length which depends on the instruction shown.

If the value contained in BYTES is 01, only the two extreme left hand displays will be lit. If BYTES contains 02, on the other hand, the two left hand and the two centre displays will be lit. However, all the displays will be lit if the content of BYTES is 03.

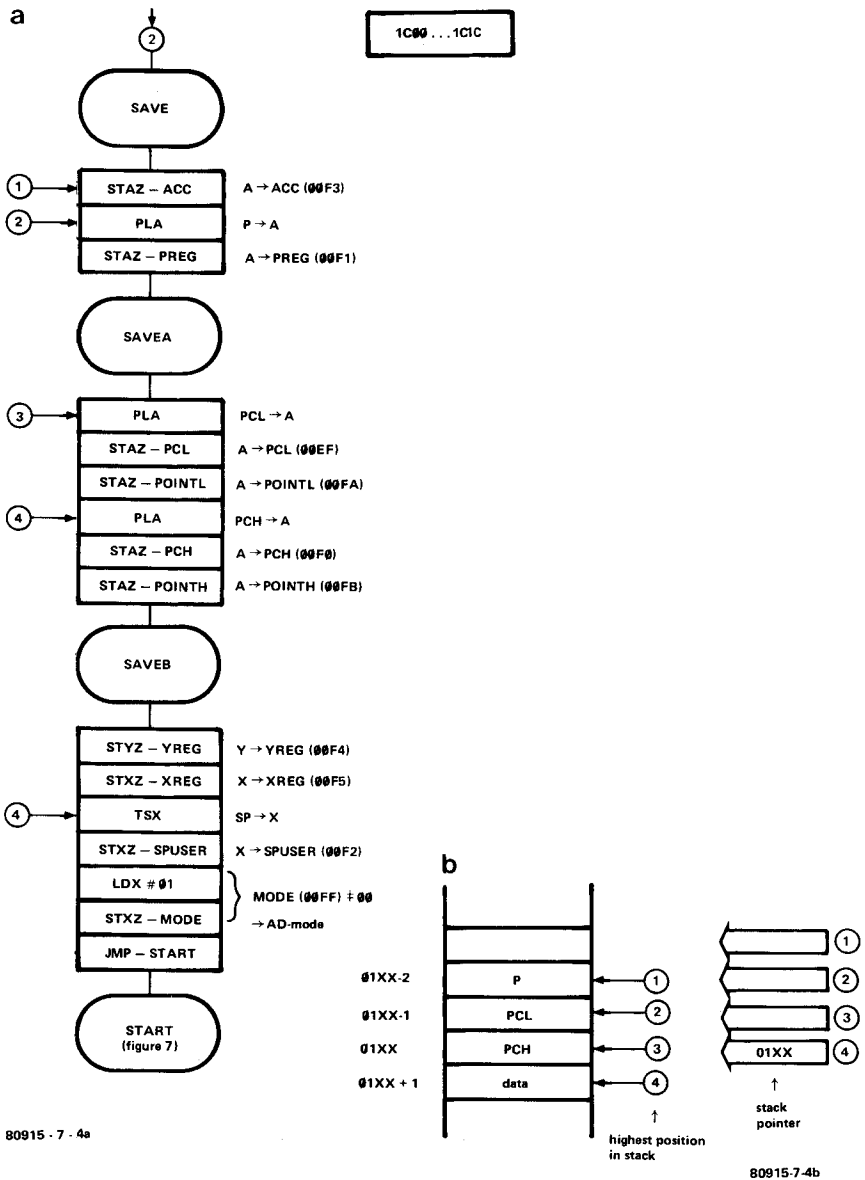
After this, the CPU will set up the stack pointer to point at address location 01FF. Since the stack pointer in the 6502 microprocessor is always situated on page 1, only the low order byte of the pointer value will have to be set (the high order byte will always be 01!). The corresponding instructions are: LDX #FF and TXS. As the stack pointer must also be saved in page zero, the low order byte will also have to be stored in location SPUSER.

The two subsequent instructions (CLD and SEI) ensure that the Junior Computer is operating in the binary mode and that any interrupt requests are inhibited. When the CPU finally reaches the label START, the reset sequence has been completed.

## SAVE

The next item in the monitor program to be discussed is the SAVE routine. As we know from figure 1, the SAVE routine is entered by way of input ② of the monitor program. This section of the monitor saves the contents of all the internal CPU registers in page zero. The SAVE routine is also utilised when the computer is in the step mode.

The way in which the CPU registers are saved can be seen from figure 4. As mentioned previously, the SAVE routine is accessed in the step mode by way of a non-maskable interrupt. We also know (or should do by now!) that the high and low order bytes of the program counter and the contents of the processor status register are stored on the stack (in that order) after a non-maskable interrupt. Prior to the NMI the stack pointer would be directed to location 01xx. The CPU will store the value of the high order byte of the program counter at this location and will then decrement the low order byte of the stack pointer. Now the stack pointer will be pointing to location 01xx - 1. This is where the processor stores the low order byte of the program counter and the stack pointer is once again decremented



**Figure 4a.** The detailed flowchart of the SAVE routine, which is executed after a jump to monitor via an interrupt.

**Figure 4b.** The state of the stack and stack pointer at certain moments during the SAVE routine.

by one. This means it will now be pointing at location  $01xx - 2$ . The contents of the status register are then stored at this location and the stack pointer is decremented to point at location  $01xx - 3$ .

By this time the processor will have reached the label SAVE in the monitor program (see figure 4). Firstly, the contents of the accumulator are stored in location ACC ( $00F3$ ). The routine continues by fetching the previous contents of the status register from the stack (PLA) and storing it in location PREG ( $00F1$ ). At the beginning of the next section of the SAVE routine (SAVEA) the processor encounters another PLA instruction. This obtains the low order byte of the program counter from the stack, which is then saved in locations PCL ( $00EF$ ) and POINTL ( $00FA$ ). The following PLA instruction fetches the high order byte of the program counter from the stack. This is then saved in locations PCH ( $00F0$ ) and POINTH ( $00FB$ ).

Just to recap briefly: before reaching the label SAVE, the stack pointer was directed to address location  $01xx$ . After the interrupt the CPU saved the high and low order bytes of the program counter and the contents of the status register on the stack. Once the processor reaches the label SAVE, the stack pointer will be directed to address location  $01xx - 3$ . Between labels SAVE and SAVEB the processor carries out three PLA instructions, therefore the stack pointer will again point to address location  $01xx$  (in other words it will resume its original state).

Thus, up to now, the processor has used the stack as a temporary memory to save the value of the program counter and the contents of the status register in page zero. The display buffers have also been renewed as the value of the program counter has been entered into locations POINTH and POINTL. The contents of the rest of the CPU registers are yet to be saved. This is accomplished after the label SAVEB where the contents of the Y register are stored in location YREG ( $00F4$ ) and the contents of the X register are stored in location XREG ( $00F5$ ).

The value of the stack pointer is the final parameter to be saved. This is stored in location SPUSER with the instructions TSX and STYX-SPUSER. The state of the stack pointer has remained unaltered since the last PLA instruction. Therefore the low order byte of the stack pointer,  $xx$ , can be accessed from location SPUSER.

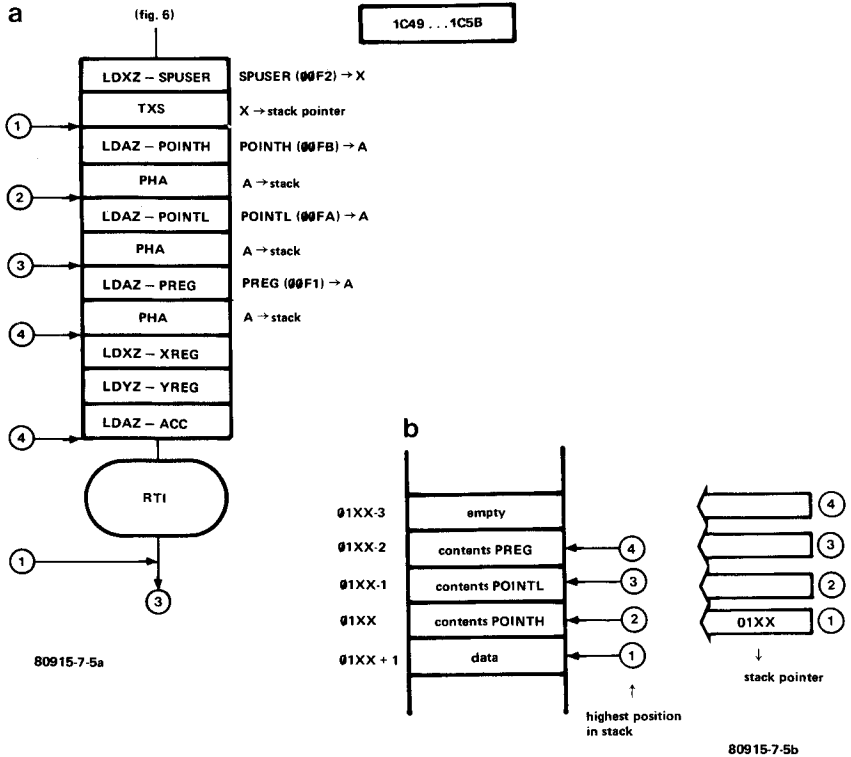
When the processor has reached this stage in the routine the contents of all the internal registers will have been saved. The processor then goes on to fulfil an important task. It enters a number that is not equal to zero into address location MODE ( $00FF$ ). As a result, the Junior Computer being in the step mode, after carrying out an instruction will automatically switch to the address (AD) mode. This prevents the programmer from inadvertently overwriting a machine instruction with incorrect data.

The processor has now reached the label START and will once again control the keyboard and display. By depressing the GO key the Junior Computer will leave the monitor program (see figure 1) and branch to the user's program. The following passage deals with the process involved step by step.

## Leaving the monitor program

As we know, the Junior Computer remains inside a loop in the monitor program. Here it constantly multiplexes the display and scans the keyboard waiting for a key to be depressed. If the programmer depresses the Go key, the computer will leave the loop and will branch into the user's program. From Book I we know that when the GO is depressed the Junior Computer will start work from the address shown on the display at that particular moment. As soon as it has left the monitor program, it no longer worries about the keyboard and display, but devotes its entire attention to the user's program.

Figures 5a and 5b clearly show what happens when the computer leaves the monitor program by way of the GO key. Figure 5 shows the actual program steps, which are virtually the reverse of the SAVE routine, while figure 5b shows the state of the stack and the stack pointer.



**Figure 5a.** The detailed flowchart of the routine that is executed after the GO key is depressed. This routine is the exact inverse of the SAVE routine in figure 4a.

**Figure 5b.** The state of the stack and stack pointer at certain moments during the routine of figure 5a.

First the CPU obtains the address of the user stack from location SPUSER and stores that address in the stack pointer (LDXZ-SPUSER, TXS). This means that the stack pointer will once again indicate the address 01xx. The contents of the display buffers, POINTH and POINTL, are then also placed on the user stack, in that order. The stack pointer will then, of course, be indicating address location 01xx - 2. This means that the contents of POINTH and POINTL will be used as the new program counter. The original contents of the status register are stored on the stack (LDAZ-PREG, PHA). The stack pointer now indicates location 01xx - 4.

Once the new program counter and the previous contents of the status register have been placed on the stack, the contents of the X and Y index registers and the accumulator are restored to their original values. This ensures that all the internal CPU registers contain specific data and the Junior Computer can branch into the user's program quite happily. This is actually carried out via the RTI instruction. As you will remember, following an RTI instruction the contents of the status register and the low order and high order bytes respectively the program counter are pulled from the stack. In this instance the value of the program counter is obtained from the contents of POINTH and POINTL. This enables the processor to leave the monitor and continue program execution from the address currently on display - the start address of the user program.

**Note:** If the Junior Computer happens to be in the step mode, it will execute the program sequence shown in figure 5a when the GO key is depressed. It will branch to the address shown in the display which contains a machine instruction and will carry out this instruction. However, when the opcode for the instruction is being loaded into the instruction register the CPU will generate a non-maskable interrupt (NMI) via its SYNC output. The processor therefore completes the current instruction and then deals with the interrupt.

In the step mode the NMI vector points to address location 1C00, which is the start of the SAVE routine. Now the CPU executes the program sequence shown in figure 4, saving the contents of all registers and switching the computer to the address (AD) mode. After this sequence the processor again arrives at the START label where it once more enters the loop and takes care of the display and scans the keyboard. The next machine instruction, together with its address, appears on the display and the computer waits until the GO key is depressed once more. When the GO key is depressed the processor branches to the address currently displayed to carry out the instruction shown and the whole procedure is repeated.

## How the Junior Computer responds to a depressed key

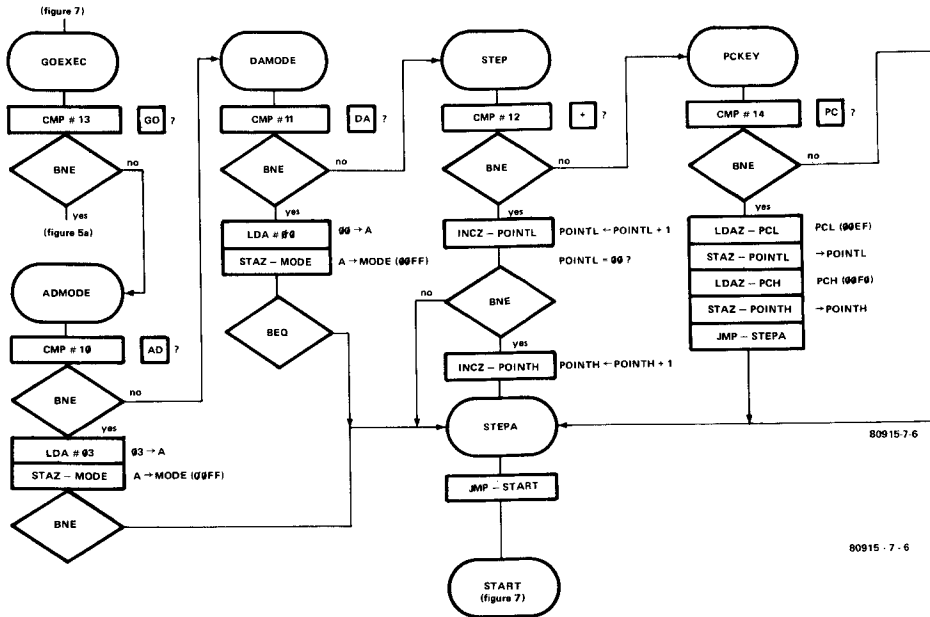
This section of the chapter discusses the way in which the Junior Computer responds to any key that is depressed. To start with, the computer discovers whether or not a key is depressed in the loop section of the monitor program. The identification of the particular key, or rather the calculation of its value, takes place in a subroutine. At the end of the subroutine the key value will be held in the accumulator. As already

discussed in Book I, the following values have been assigned to the various keys:

0:00	6:06	C:0C	+:12
1:01	7:07	D:0D	GO:13
2:02	8:08	E:0E	PC:14
3:03	9:09	F:0F	(illegal key: 15)
4:04	A:0A	AD:10	
5:05	B:0B	DA:11	

Figure 6 illustrates the complete key recognition sequence. A series of compare (CMP) instructions each followed by a branch-if-not-equal (BNE) instruction filter out the value of the depressed key. The CPU branches if the depressed key does not have the same value as that under comparison and passes on to the next compare instruction. If the two values are the same, the processor performs the relative operations pertaining to that particular key.

The keyboard of the Junior Computer can be divided into two separate sections: one for the data keys and one for the command keys. We have already seen what happens when the GO key is depressed. If the command key AD is depressed, the contents of address location MODE are made not equal to zero (03). The processor will now interpret the next keys to be depressed as an address. By means of the subsequent BNE instruction the program branches to the START label via STEPA.

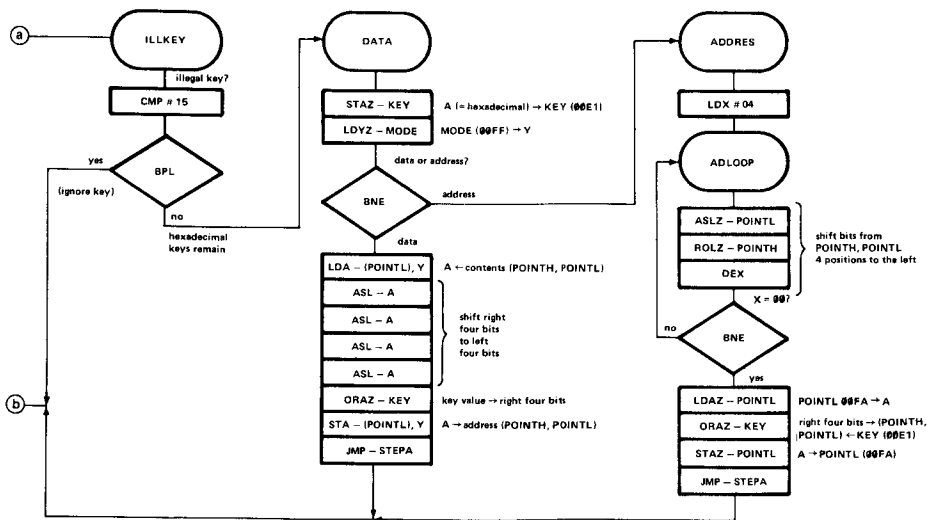


If however, the DA key was depressed, the contents of location MODE would be made equal to zero. The computer will then interpret the next keys to be depressed as being data to be stored in the address indicated and then shown on the display.

As we know, the computer increments the address shown on the display whenever the + key is depressed and then it displays the data contained in this new address. If the contents of POINTL were FF before the + key was depressed, for instance, this would now become 00. This means that the contents of the address buffer POINTH must also be modified, that is to say, they must be incremented by one. A BNE instruction detects whether or not this second correction is necessary. In either case the program again branches to START via STEPA.

When the PC key is depressed, the computer copies the current position of the program counter into the display buffer. Only a few instructions are required and as the contents of the program counter and the display buffers themselves are all situated on page zero, the actual routine is very short indeed. Once this has been accomplished the monitor program once again branches back to START.

By now we should know how the PC key works: If a program is executed in the step mode, it may be necessary to examine the contents of some, or all, of the internal CPU register. As these are all saved on page zero, the user's program will have to be left for a while. By depressing the PC key



**Figure 6. The detailed flowchart of the key recognition sequence. All the function keys are tested individually. The processor performs separate routines for each key according to its function. Details of the routine belonging to the GO key are given in figure 5. In the case of a hexadecimal key, a choice is made from two possible routines depending on whether the computer is in the address or data mode.**



the previous contents of the program counter will be restored into the display and when the GO key is depressed the computer will continue operation from the point it left off before the registers were examined. Up to now, the monitor program has been able to establish whether one of the command keys have been depressed. As each key has its own particular value, only the key values between 0 and 14 may arise. Nevertheless, there is a possibility that the Junior Computer may calculate the wrong key value due to mains interference or contact bounce. Normally speaking this should not be possible, but for greater security the processor will ignore any key value greater than 14 in the section of program labelled ILLKEY. Following this the program (yes you've guessed it!) jumps back to START via STEPA (do not pass GO, do not collect £ 200!!).

### Modification of data

If the programmer did not depress a command key, but a data key, the monitor program will branch to the DATA label. From this point on the computer processes the data keys 0 . . . F. The value of the depressed key is first stored in address location KEY (00E1) leaving the accumulator free for other tasks. The contents of location MODE are then loaded into the Y index register.

As far as the monitor program is concerned, the data now contained in the Y register can have one of two values: the data can either be equal to zero, or not equal to zero. If, for instance, the contents of the Y register are equal to zero, the depressed key will be interpreted as data entry. If, however, it is not equal to zero, the computer knows that the programmer would like to enter a new address and that the information stored in location KEY must be transferred to the address area of the display. A branch instruction (BNE) distinguishes between an address and a data entry: LDYZ-MODE, BNE-ADDRESS.

If the contents of the Y register are equal to zero, data is to be entered and the branch to the label ADDRESS does not occur. The Junior Computer shifts the information from the depressed key into the display from right to left. These are the two right hand digits on the display printed circuit board. The series of instructions to shift the data display one digit to the left operate as follows:

By using pre-indexed indirect addressing (the value contained in the Y register is equal to zero) the CPU loads the contents of the memory location indicated by the address pointers POINTH and POINTL (the data currently on display) into the accumulator. The value of the depressed key contained in location KEY has not as yet been shifted into the display. Assuming that the data now contained in the accumulator consists of bits p . . . w, the result after four shift left instructions will be as follows:

p	q	r	s	t	u	v	w	...	contents of accumulator after loading
q	r	s	t	u	v	w	0	...	contents of accumulator after the first ASL instruction
r	s	t	u	v	w	0	0	...	contents of accumulator after the second ASL instruction
s	t	u	v	w	0	0	0	...	contents of accumulator after the third ASL instruction
t	u	v	w	0	0	0	0	...	contents of accumulator after the fourth ASL instruction

After these four shifts the four most significant bits of the data byte are lost and are replaced by the four (previously) least significant bits. The four least significant bits of the data byte have now been made zero by the shift process.

The contents of address location KEY are 0000xxxx, where xxxx corresponds to the actual value of the depressed key. By ORing the contents of the accumulator with the value contained in location KEY the following result is obtained:

```
t u v w 0 0 0 0 ... contents of accumulator before the ORA instruction
0 0 0 0 x x x x ... contents of memory location KEY
t u v w x x x x ... result (in accumulator) after the OR operation
```

As can be seen, the result is that the four most significant bits remain unchanged and the four least significant bits become equal to the value of the depressed key. This effectively shifts the key value into the accumulator from right to left. The data byte thus obtained is stored in the location, again indicated by the address pointers POINTH and POINTL, shown on the display. Finally, the processor once more jumps back to START via STEPA.

The inclusion of the label STEPA may appear rather superfluous at first sight, especially as it is immediately followed by a further jump instruction which leads the processor to the label START. However, the 6502 microprocessor is unable to jump directly to any location outside a range of  $\pm 128$  bytes. For this reason the processor must jump to START indirectly by way of STEPA.

## Modification of address

Just as the processor can modify data stored in the RAM section of the computer's memory, so new address information can be entered. In other words, the contents of the two memory locations constituting the address pointer, POINTH and POINTL, can be altered (after having depressed the AD key). This procedure is carried out during the ADDRESS section of the routine shown in figure 6. After loading the X index register with 04, the section of program ADLOOP – BNE – ADLOOP is performed four times in succession. The contents of POINTL are shifted left four times, the contents of POINTH are rotated left four times and the contents of the X register are decremented by one four times.

During the shift instruction the most significant bit of POINTL is moved into the carry position and the least significant bit becomes zero. During the rotate instruction the most significant bit of POINTH is lost and the contents of the carry flag replace the least significant bit.

Let us examine this in greater detail. If we suppose that the values of the bits of

the four most significant bits of POINTH are h i j k, and  
the four least significant bits of POINTH are l m n o, and  
the four most significant bits of POINTL are p q r s, and  
the four least significant bits of POINTL are t u v w, and  
the initial state of the carry flag is x, the following will happen:

h	i	j	k	l	m	n	o	x	p	q	r	s	t	u	v	w	... initially
h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	0	... after ASL-POINTL
i	j	k	l	m	n	o	p	h	q	r	s	t	u	v	w	0	... after ROL-POINTH } X = 04
i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	0	0	... after ASL-POINTL
j	k	l	m	n	o	p	q	i	r	s	t	u	v	w	0	0	... after ROL-POINTH } X = 03
j	k	l	m	n	o	p	q	r	s	t	u	v	w	0	0	0	... after ASL-POINTL
k	l	m	n	o	p	q	r	j	s	t	u	v	w	0	0	0	... after ROL-POINTH } X = 02
k	l	m	n	o	p	q	r	s	t	u	v	w	0	0	0	0	... after ASL-POINTL
l	m	n	o	p	q	r	s	k	t	u	v	w	0	0	0	0	... after ROL-POINTH } X = 01

The four most significant bits of POINTH have been replaced by the four previous least significant bits and the four least significant bits have been replaced by the four previous most significant bits of POINTL. The four most significant bits of POINTL have been replaced by the previous four least significant bits of POINTL and the four least significant bits have been made zero.

The contents of address buffer POINTH can now remain as they are, but the contents of POINTL require modification. This is carried out by loading the contents of POINTL into the accumulator and ORing it with the data contained in location KEY (which, as we know, is the actual value of the depressed key). The result is exactly the same as that for the data key:

t	u	v	w	0	0	0	0	...	contents of accumulator before the ORA instruction
0	0	0	0	x	x	x	x	...	contents of memory location KEY
t	u	v	w	x	x	x	x	...	result (in accumulator) after OR operation

The final result is then replaced in POINTL before the computer jumps back to START via STEPA.

Since a hexadecimal key is represented by four bits of a byte and each of the six displays is capable of displaying four bits of information, it will be quite clear why there is no need to shift around all the information contained in the display buffers.

*All the monitor software, apart from block C of figure 1, has now been discussed in detail. Further instructions are required to actually show the contents of the display buffers on the display and to ascertain which of the keys were depressed. These instructions are part of several important subroutines which are about to be described.*

## Keyboard and display routines

### *communicating with the Junior Computer*

Immediately after the START label the Junior Computer examines the keyboard and multiplexes the display. The routines involved display the contents of the display buffers, establish whether a key has been depressed and, if so, which one. The display is activated immediately after the START label as its duty is to report back to the operator. The START label is reached after one of the keys to have been depressed by the operator, via

one of the key routines previously described, has finished. Although the routine for the GO key only partly takes place in the monitor program, even after one (step mode) or all the instructions of the user's program have been executed, the processor will return to START via the SAVE routine in due course. If this does not occur the programmer will have forgotten to include the BRK instruction at the end of the program (section) in question. Alternatively, the processor may 'sit' in the monitor loop and wait for new program data to be entered.

The final method of reaching the START label is to depress the RST key (initialise the Junior Computer). The instructions contained in the program section belonging to block C in figure 1 are shown in figure 7. Obviously, more instructions are used than are actually shown due to the subroutines involved. The SCAND subroutine (see later for greater detail) ensures that the contents of the three buffers, POINTH, POINTL and INH are displayed and that any depressed key is detected. The contents of the accumulator at the end of the SCAND subroutine will be equal to zero when no key is depressed, but will have a non-zero value when a key is depressed. The subroutine GETKEY simply loads the accumulator with the value of the depressed key.

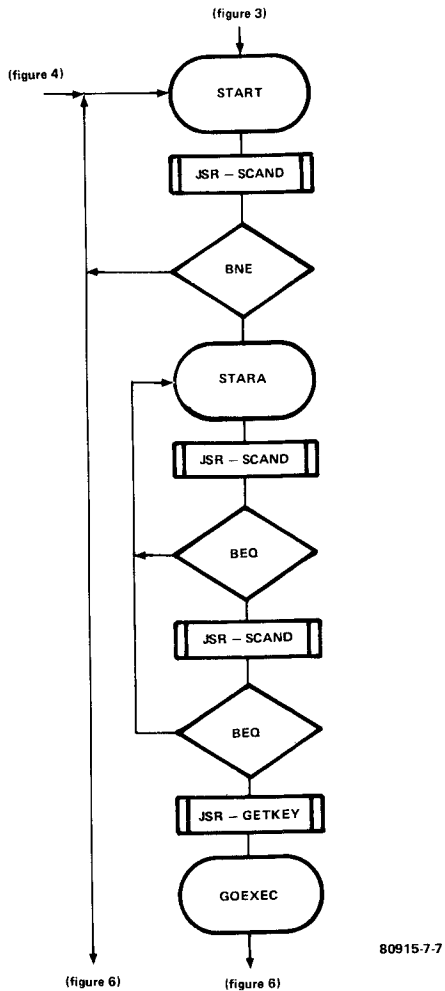
As far as figure 7 itself is concerned, something should be said about the BNE and BEQ instructions. A BNE instruction causes a branch if the contents of the accumulator are not equal to zero. Thus, the BNE instruction in figure 7 tests to see whether a key is depressed. The opposite holds for the BEQ instructions: the processor will only branch if the contents of the accumulator are equal to zero. In other words, the two BEQ instructions in figure 7 detect whether the key has been released.

When a key is depressed, any change in the display will occur immediately after it is depressed — not when it is released. The machine can therefore only handle one depressed key at a time, so that the key must be released before any other can be depressed. This is because of the combination of the first jump to the SCAND routine and the associated branch-if-not-equal-to-zero instruction (the processor will remain in this loop until a key is depressed).

The second jump to the SCAND routine and its associated branch (BEQ) instruction ensures that a depressed key is detected. Then a third jump to the SCAND routine takes place before the jump to the GETKEY routine. This may seem superfluous, but it effectively eliminates any effects of contact bounce between the release of one key and the depression of the next (see figure 8).

Let us assume that a depressed key produces a logic level of 0 and a key that is not depressed a logic 1 level (the voltage level on the port lines PA0 . . . PA6). When a key is depressed at moment t1 the curve produced should, ideally, be similar to that shown in figure 8a. In practice, however, contact bounce causes the effect shown in figure 8b. At moments t3 and t5 the contact bounce produces a logic 1 (non-depressed key) **after** the key was depressed at moment t1, resulting in a situation where the key is **apparently not depressed**.

Software ensures that the key is not actually detected until after moment t6 in figure 8b, in other words, after the effects of contact bounce have died away. Thus, the apparently superfluous jump to the SCAND routine

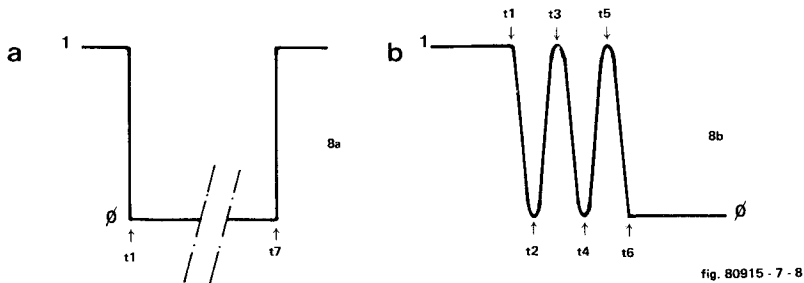


80915-7-7

**Figure 7. The flowchart for the program section that scans the keyboard and multiplexes the display. Obviously, the flowchart is not complete due to the various subroutines involved.**

fills the time lapse between moments  $t_1$  and  $t_6$  and the problem is solved. The loop including the second jump to SCAND and the BEQ instruction (figure 7) is left soon after the period  $t_1 - t_2$  when a key is depressed. After the third jump to SCAND and the associated BEQ instruction no branch will occur and the subroutine GETKEY will be reached soon after moment  $t_6$ .

Contact bounce will also occur when a key is released before another one is depressed. Similar to the previous situation, **the key will apparently**



**Figure 8.** When a key is depressed (or released) a specific change in logic level is involved (8a). The key is depressed at moment t1 and released at moment t7. In practice, however, errors could arise from contact bounce (8b). After a key has been depressed there are moments (t3 and t5 to be precise) when the logic level is high instead of low. By means of the monitor software the key is not accessed until after moment t6, after the contact bounce has subsided.

**still be depressed.** However, by the time the SCAND routine has been completed for the third time, the contact bounce will have subsided and the processor will be ready to detect any new key depressed.

Before we go into the subroutines SCAND and GETKEY in detail, let us examine the hardware involved in multiplexing the display and recognising which key is depressed.

## From software to hardware

### *Display multiplexing*

As we know, the programmer enters information into the computer through the use of the keyboard and the computer 'replies' via the six digit display. We also know that during this process the input/output (I/O) section of the peripheral interface adapter (PIA) has an important task to fulfil. The input section is the keyboard via which data is entered into the computer and the output section is the display which is controlled by software.

The relevant section of the circuit connected to ports A and B is shown in figure 9. Port B is programmed solely as an output and is connected to inputs A...D of IC7. Depending on the bit pattern present on these inputs, one of the ten outputs of IC7 will be low (logic zero). Six of the ten outputs are connected to the common cathode of a display. For one or more segments of a display to light, the cathode of that display must be grounded and so the corresponding output of IC7 must go low. The following truth table may be drawn up for the BCD to decimal decoder:

**Table 1**

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	display to be turned on
x	x	x	0	1	0	0	x	Di1 } ADH (POINTH buffer)
x	x	x	0	1	0	1	x	Di2 }
x	x	x	0	1	1	0	x	Di3 } ADL (POINTL buffer)
x	x	x	0	1	1	1	x	Di4 }
x	x	x	1	0	0	0	x	Di5 } data (INH buffer)
x	x	x	1	0	0	1	x	Di6 }

Where a port output is indicated with an x the bit may be a 0 or a 1 as the corresponding port line is not connected.

If a display is switched on and all of the outputs of IC11 are high, all seven segments of that display will light and an '8' will appear. By turning off certain segments it is possible to display all the hexadecimal numbers from 0...F (and other characters besides – more about this later). This is accomplished by making port A an output and using it as a 'segment switch'.

Let us suppose that we wish to turn off segment 'c'. The corresponding inverter output, pin 12 of IC11, must go low. For this to happen the inverter input, pin 5 (connected to PA2), must go high. In other words, a certain segment pattern is obtained by outputting a specific bit pattern on port A. If a particular bit is high the corresponding segment will not light; if it is low, it will.

Table 2 indicates all of the bit patterns required on the output lines of port A to produce the relevant hexadecimal display. The corresponding identification of a segment display is shown in figure 10.

**Table 2**

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	hexadecimal
	$\bar{g}$	$\bar{f}$	$\bar{e}$	$\bar{d}$	$\bar{c}$	$\bar{b}$	$\bar{a}$	
x	1	0	0	0	0	0	0	0
x	1	1	1	1	0	0	1	1
x	0	1	0	0	1	0	0	2
x	0	1	1	0	0	0	0	3
x	0	0	1	1	0	0	1	4
x	0	0	1	0	0	1	0	5
x	0	0	0	0	0	1	0	6
x	1	1	1	1	0	0	0	7
x	0	0	0	0	0	0	0	8
x	0	0	1	0	0	0	0	9
x	0	0	0	1	0	0	0	A
x	0	0	0	0	0	1	1	B
x	1	0	0	0	1	1	0	C
x	0	1	0	0	0	0	1	D
x	0	0	0	0	1	1	0	E
x	0	0	0	1	1	1	0	F

During segment control port line PA7 can be used as an input as there is nothing connected to this line. The bit patterns shown in table 2 are stored in the monitor program EPROM (location 1F0F tot 1F1E). They are, of course, represented in hexadecimal form (see source listing in the appendix of this book).

### Detecting a depressed key

As shown in figure 9, every key has two contacts. One is connected to an output of IC7 and the other to one of the (output) lines of port A. The 21 keys are arranged in a matrix of three rows and seven columns. All the keys in one row have a common connection to a certain output of IC7. Keys in the same column are connected to a single port line.

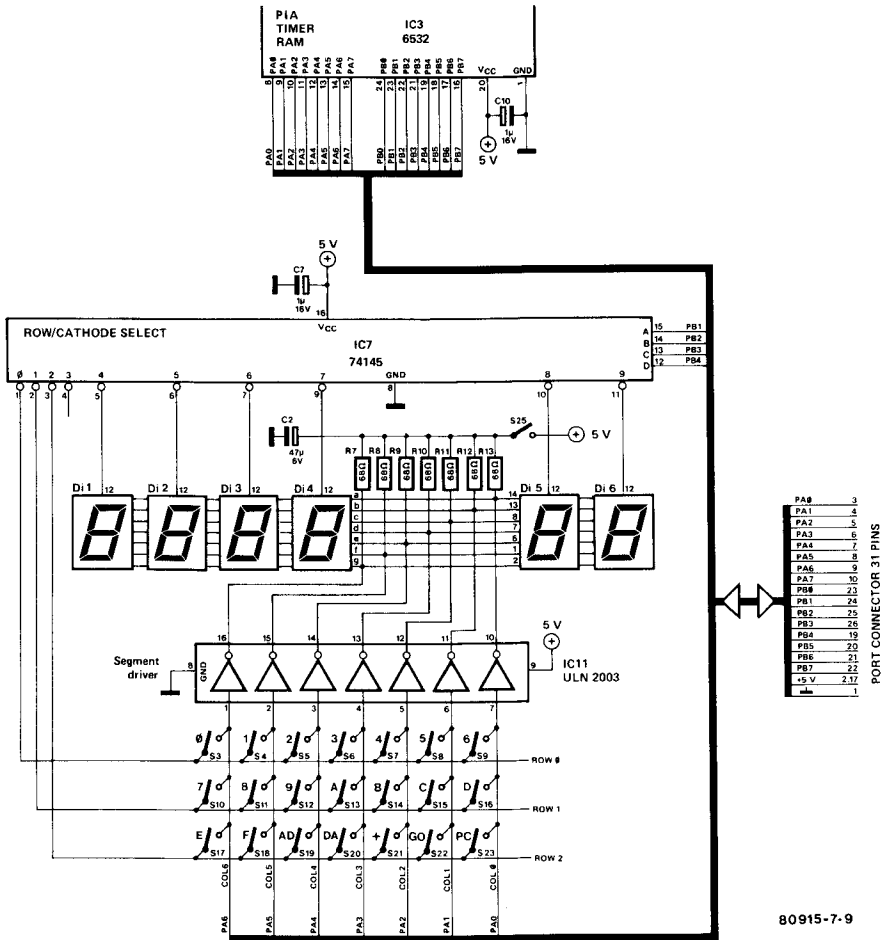
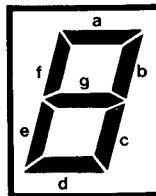


Figure 9. A closer look at the keyboard and display section of the Junior Computer circuit diagram.



80915-7-10

Figure 10. The segments of a display are identified by the letters a . . . g. The bit pattern on port lines PA0 . . . PA6 light up the various segments required to produce a particular character on the display.



In order to detect whether a key has been depressed, port A lines have to be programmed as inputs. If it is correctly controlled via port B, one of the first three outputs of IC7 (0, 1 or 2) will be low. When a key is depressed the same logic zero will appear at one of the (output) lines of port A. In this manner a depressed key can be detected and identified without error. The matrix row is determined by the bit pattern on port lines PB0 . . . PB4 and the column by whichever of port A lines goes low. The two tables below sum up the situation:

**Table 3**

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	row of depressed key
			D	C	B	A		
x	x	x	0	0	0	0	x	row 0 (keys 0,1,2,3,4,5,6)
x	x	x	0	0	0	1	x	row 1 (keys 7,8,9,A,B,C,D)
x	x	x	0	0	1	0	x	row 2 (keys E,F,AD,DA,+, GO and PC)

This establishes which row the depressed key is situated in. The column of the particular key can be determined from:

**Table 4**

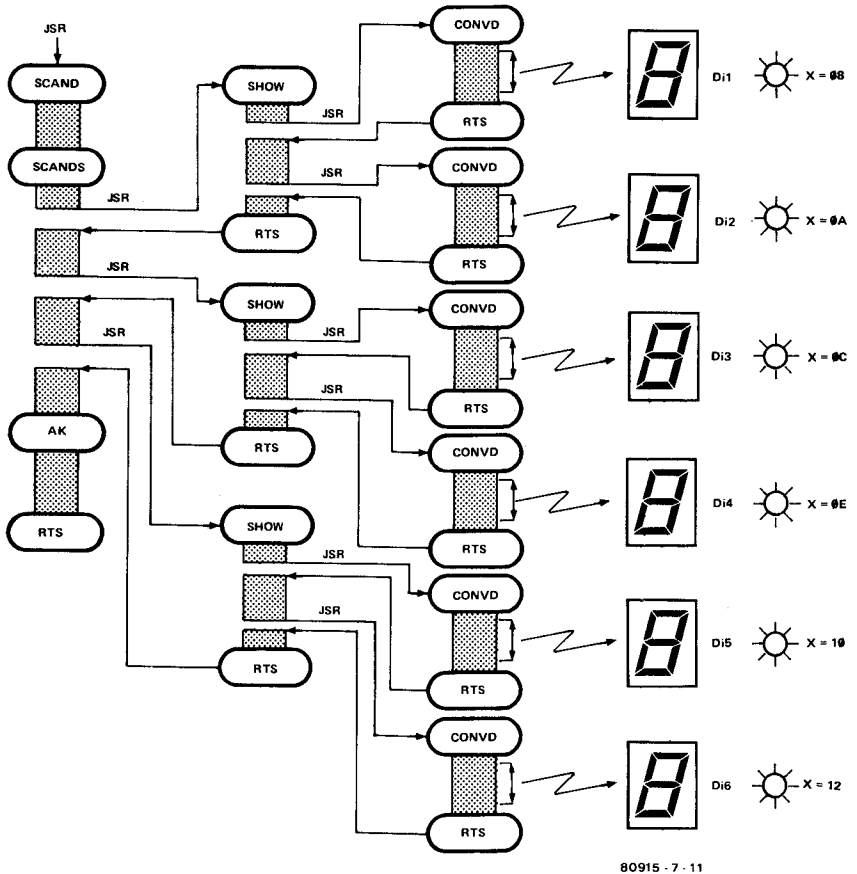
	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	column
keys 0, 7 and E	x	0	1	1	1	1	1	1	
keys 1, 8 and F	x	1	0	1	1	1	1	1	
keys 2, 9 and AD	x	1	1	0	1	1	1	1	
keys 3, A and DA	x	1	1	1	0	1	1	1	
keys 4, B and +	x	1	1	1	1	0	1	1	
keys 5, C and GO	x	1	1	1	1	1	0	1	
keys 6, D and PC	x	1	1	1	1	1	1	0	

*Now back to the software.*

### The subroutine SCAND – minus AK

Before going through each instruction in this subroutine, it might be a good idea to summarize the sequence of events first. Inside the SCAND there is another subroutine called SHOW, which also contains a subroutine called CONVD. The subroutine nesting thus formed is shown in figure 11. It involves the display section of SCAND, or rather, everything that happens before AK, the key detection portion of SCAND.

The contents of the two address buffers, POINTH and POINTL, together with the contents of the data buffer INH, must be displayed. Each display is controlled by half of the data byte contained in each of the three buffers. In the subroutine CONVD, the displays are activated in turn for a certain period of time. During the SHOW subroutine, the processor establishes which part of the buffers is to be displayed – which display is to become active after the jump to CONVD. There are two jumps per display buffer from SHOW to CONVD. As there are three display buffers, this means that there are three jumps from SCAND(S) to SHOW.



**Figure 11.** The subroutine nesting that occurs during the SCAND section of the monitor program. The instructions pertaining to the various subroutines are shown as shaded rectangles.

Thus, during the execution of the SCAND routine all six displays will light. Since there is a short delay loop (the second jump to SCAND plus the associated BEQ instruction in figure 7) the displays are in fact accessed periodically. This is termed **software controlled display multiplexing**.

The instructions used in the subroutine SCAND are given in figure 12. The first of these cause location INH to be used as a data buffer: the contents of the memory locations indicated by POINTH and POINTL are entered into INH. (In the editing mode the three display buffers are used for other purposes). The program then moves on to the next section, SCANDS. This starts by programming port A lines (which will act as a segment switch) as outputs. The X index register is then loaded with 08.

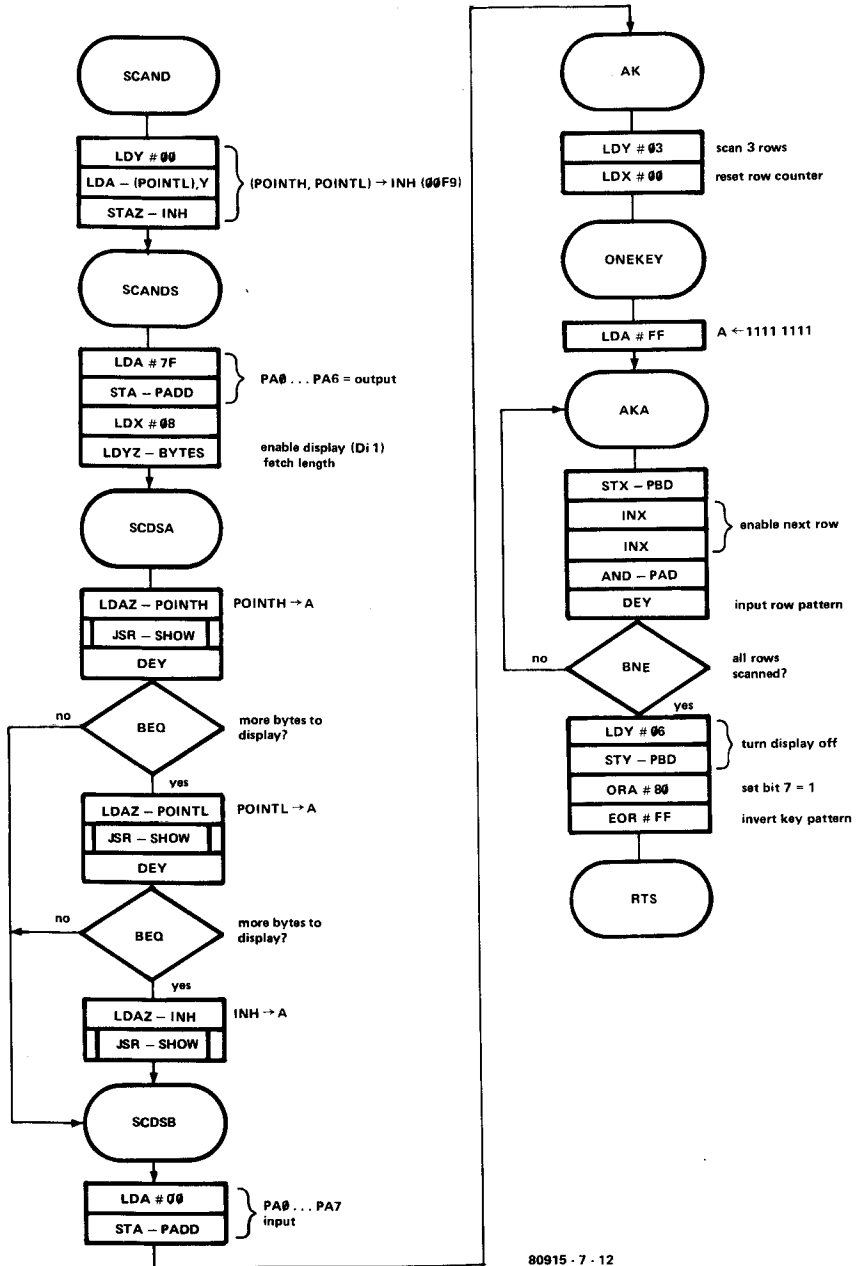


Figure 12. The detailed flowchart of the SCAND subroutine. It is structured so that the second half of it, AK, can be used as a separate routine by the programmer.

This value is transferred to the display control, IC7, via port B during the CONVD subroutine. The X index register thus controls the display digits via port B and IC7 and is modified during subroutine SHOW and CONVD so that it always indicates the next display to be switched on.

Next, the Y index register is loaded with the contents of location BYTES. This determines how many of the displays are to be lit. When the RST button is depressed, location BYTES is loaded with 03. This means that the contents of all three data buffers (POINTH, POINTL and INH) are to be displayed. Then the accumulator is loaded with the contents of POINTH and the processor jumps to the SHOW subroutine. The data contained in location POINTH is split into groups of four bits each and these are presented to the corresponding displays during the SHOW and CONVD subroutines, that is, to displays Di1 and Di2 (see figure 11). After this the Y index register is decremented by one and the processor returns to the SHOW subroutine to display the contents of POINTL in displays Di3 and Di4. The Y index register is then decremented and the contents of INH are transferred to displays Di5 and Di6. All that remains is to program port A lines as inputs in preparation for the program section AK to follow later.

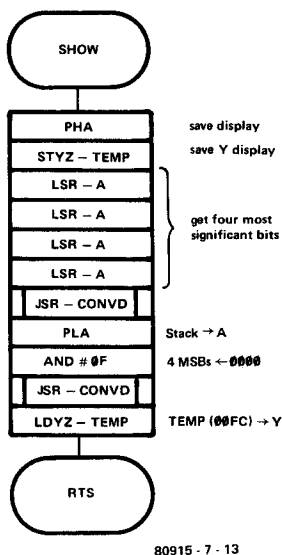
The use of the Y index register as a display buffer counter may seem superfluous here, seeing as the number of buffers to be displayed has already been determined. However, the SCANDS subroutine is also used for editing purposes when the length of the display varies. Thus it is necessary to use the Y index register as a counter in this subroutine.

## The subroutine SHOW

At the start of this subroutine (see figure 13) the contents of the buffer to be displayed are held in the accumulator. Each of the two halves of a buffer data byte has its own display digit. This means that there are always two jumps involved from SHOW to CONVD. The latter subroutine operates by using the accumulator as an index to obtain the correct seven segment information to be displayed. This means the most significant bits of the data byte held in the accumulator must be zero.

Since the multiplex operation controls the six displays from left to right, the four most significant bits need to be presented first followed by the four least significant bits. Therefore, by repeating the instruction LSR-A four times in succession the four least significant bits are replaced by the four most significant bits and the four most significant bits become zero. In order that the four least significant bits can be displayed afterwards the accumulator contents are saved on the stack before the shift operation by means of the instruction PHA. The stack will then be used as a temporary store. The buffer counter contents are also saved as the Y index register is required for other tasks during the subroutine CONVD. Now, the CONVD subroutine can be called up to convert the (previous) four most significant bits of the data byte into a seven segment bit pattern to be shown on the display. Exactly how this conversion takes place will be seen later.

After the return from subroutine CONVD to subroutine SHOW the original state of the accumulator is restored. It will again contain every-



**Figure 13. The detailed flowchart of the SHOW subroutine which assembles the data to be displayed in the accumulator.**

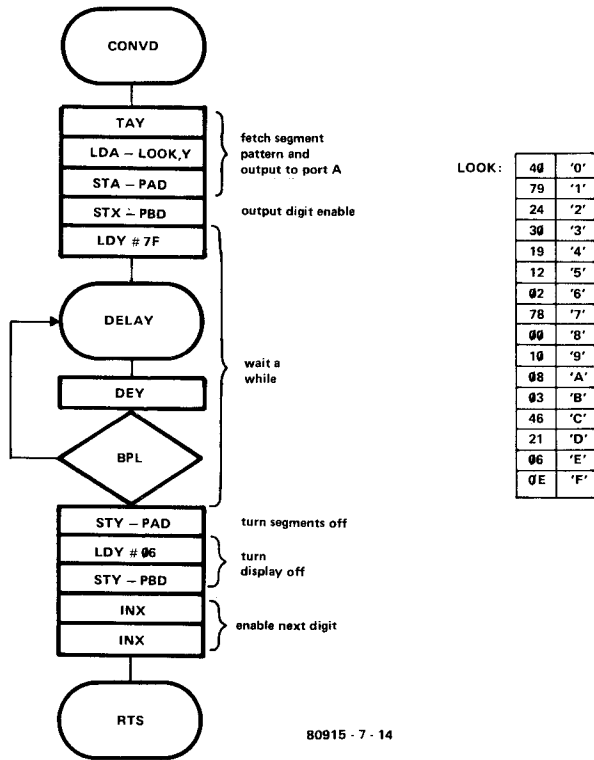
thing to be displayed on the two relative displays. Now, the four least significant bits are to be displayed and the four most significant bits removed. This is accomplished by ANDing the contents of the accumulator with 0F. A further jump to the CONVD subroutine causes the four least significant bits of the data byte to be presented to the appropriate display.

Upon the processor's return from CONVD to SHOW the original state of the buffer counter is restored (LDY-TEMP), so that the computer is able to take care of the next data buffer to be displayed.

### The subroutine CONVD

As mentioned previously, this subroutine obtains the seven segment code for the particular four data bits to be displayed. During the process a hexadecimal number must be converted into a corresponding seven segment code. Before the CONVD subroutine can be called the four least significant bits of the data byte in the accumulator must correspond to the value of the depressed hexadecimal key and the four most significant bits must be zero. The complete subroutine is illustrated in figure 14.

Initially, the hexadecimal value to be displayed is transferred into the Y index register. Post-indexed indirect addressing is then used to load the accumulator with the corresponding value from the look-up table. If, for example, the figure 'D' is to be displayed, the accumulator must contain the seven segment code of 21. The accumulator contents are then



80915 - 7 - 14

**Figure 14.** The detailed flowchart of the CONVD subroutine which transfers the data in the accumulator to the Y index register and thus obtains the correct seven segment code from the look-up table.

transferred to port A and the contents of the X index register to port B, which now functions as a display digit selector. The display digit will light according to the bit pattern obtained from the look-up table. The question now is: for how long does it remain lit? The answer being: for the duration of the loop DELAY-BPL-DELAY. The processor remains in this loop until the N flag is set. Thus, a display is switched on for a little more than 630  $\mu$ s. When the delay loop is over the Y index register will contain the value FF. The processor transfers this bit pattern to the port A lines thereby switching off all the display segments. The instruction for this is STY-PAD. The following instructions (LDY # 06 and STY-BPD) place the 'display selector' into the neutral position and the unconnected output of IC7 (pin 4) becomes active and none of the display cathodes will be low. At the end of the CONVD subroutine the X index register is incremented twice in order to enable the next digit to be displayed.

Note: The four most significant bits of the accumulator contents must be equal to zero before the jump to CONVD, as otherwise the contents of

the Y index register would become greater than 0F and that would mean exceeding the highest address of the look-up table (1F1E). As there is still memory following 1F1E, segments would light at random and give an incorrect indication.

### Section AK of the SCAND subroutine

The AK section completes the SCAND subroutine and serves to detect whether a key is depressed. As we know, the keys are arranged in a matrix of three rows and seven columns (see figures 9 and 16). The bit pattern presented to the port B lines determines which display is 'on' and which row is to be scanned for a depressed key at a particular moment. Port B is controlled by means of the X index register. As a reminder, the contents of the accumulator at the end of the SCAND subroutine will not be equal to zero if a key is depressed.

At the start of the AK routine all port A lines are pulled 'high' by internal resistors (these lines were programmed as inputs immediately before the AK routine). The Y index register is then loaded with 03 and is used as a row counter. There are three rows to scan: ROW0, ROW1 and ROW2. The contents of the X index register determine which of the three is selected via port B. The hexadecimal value FF is then loaded into the accumulator (ONEKEY). The contents of the X index register are then transferred to port B and ROW0 is taken low (see table 3). The value in the X index register is then incremented twice in preparation for the following row. The contents of the accumulator are then ANDed with the bit pattern presented to the port A input lines. The value in the Y index register is then decremented by one. In the end, after the loop AKA-BNE-AKA three times, the number of zeros in the accumulator will correspond to keys that are depressed simultaneously in any column (COL0 . . . COL6). Normally, however, only one key will be depressed at a time (and the processor is so fast that if two keys were depressed at the same time only one would be 'seen'). If, for example, one of the keys 3, A or DA in column 3 were depressed, bit 3 in the accumulator would be zero after the AND instruction. The display/row selector (IC7) is then placed in the neutral position (as in the CONVD subroutine) with the instructions LDY # 06, STY-PBD. Further details of this can be found in tables 1 and 3.

The penultimate instruction (ORA 80 – not counting the RTS instruction) causes bit 7 in the accumulator to become one. This is important as port line PA7 could well be used as an interrupt input in conjunction with a peripheral device such as a printer. This means that bit 7 could be either high or low at any time. Preparation is now complete for the final instruction (EOR # FF) which inverts all the bits in the accumulator. Any of the bits b0 . . . b6 in the accumulator could be zero after the loop AKA-BNE-AKA if a key was depressed. After the inversion that bit will be one and the rest zero. This means that the contents of the accumulator will not be equal to zero if no key is depressed. Thus, the sole purpose of the AK routine is to test to see if a key was depressed or not. The following subroutine to be discussed calculates the actual value of the depressed key.

## The subroutine GETKEY

Once the processor has determined that a key has indeed been depressed the next task is to calculate the value of the key. This is accomplished during the subroutine GETKEY. Figure 16 shows the layout of the keys in the matrix and also indicates the values allocated to the particular keys.

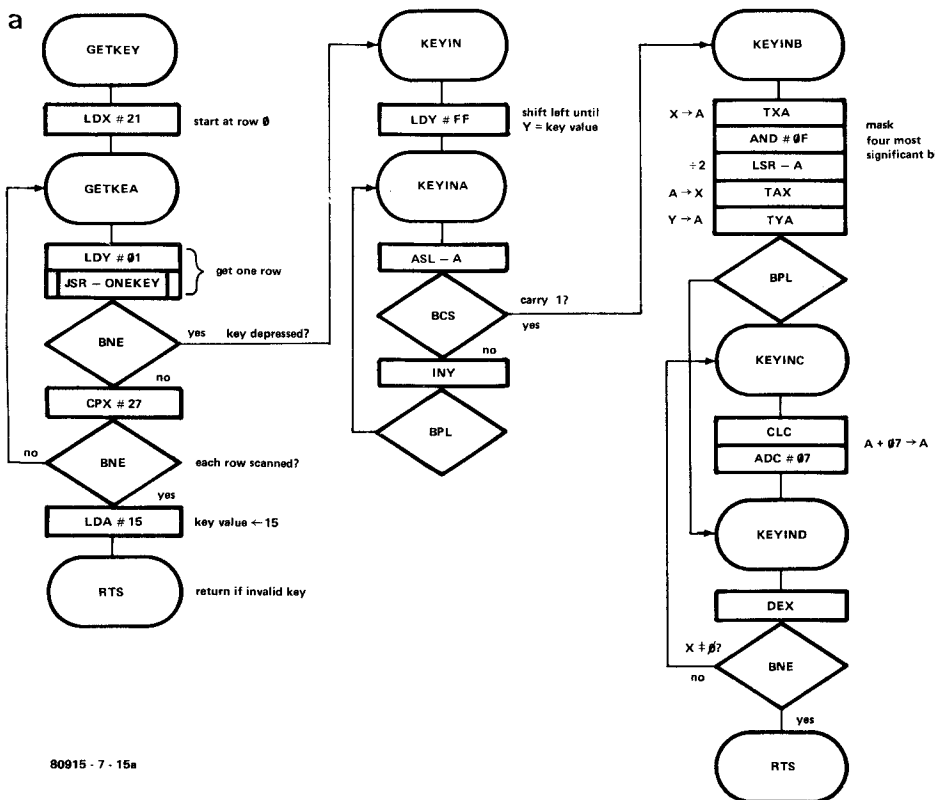
The series of instructions contained in the GETKEY subroutine are shown in figures 15a and 15b. To make things clearer, the section of program labelled ONEKEY (see figure 12) is shown again. At the start of the GETKEY subroutine the hexadecimal value 21 is loaded into the X index register. As a result, ROW0 will be scanned at the next jump to ONEKEY. The Y index register is loaded with 01, as ONEKEY must only deal with one row. If a key was depressed after the return from ONEKEY, the contents of the accumulator will not be equal to zero. The following BNE instruction will then enable the program to branch to KEYIN. If no such key was discovered, the value contained in the X index register is tested. If this value is not equal to 27 the program will branch back to label GETKEA and the subroutine ONEKEY will be called once more.

Why must the contents of the X index register be compared with 27? At the start of the GETKEY subroutine the value in the X register is made equal to 21 and so ROW0 will be accessed. During the subroutine ONEKEY the X index register will be incremented twice, making its value 23. As this does not equal 27, subroutine ONEKEY is run again, although this time ROW1 will be accessed. After the return from ONEKEY the value in the X index register will be 25. If no key in ROW1 was depressed, the processor must again jump to ONEKEY after which the value in the X index register will be 27.

Now all the rows have been scanned and if still no key is found to have been depressed there must be an error somewhere. In that case, the computer must not branch to KEYIN and start calculating the key value. Instead, after the compare instruction, the accumulator is loaded with 15 to instruct the computer to ignore the 'depressed' key and to go back to scanning the keyboard.

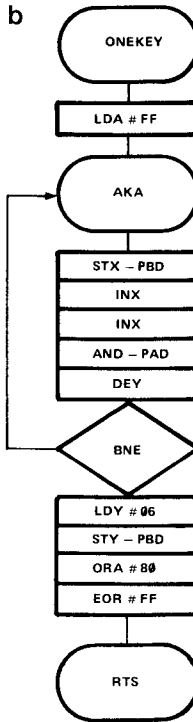
From label KEYIN onwards the computer calculates the value of a valid depressed key. The coordinates of the various keys are given in figure 16. The KEYIN routine starts by loading the Y index register with the value FF. The processor then shifts the contents of the accumulator to the left at least twice. Prior to this the accumulator contained seven zeros and a one (the one indicating the column of the depressed key). The contents of the accumulator are shifted left until the carry flag is set. After each shift the state of the carry flag is tested and if it is not set the value in the Y index register is incremented. Therefore, the value in the Y register now indicates the column of the depressed key. With respect to the keys in ROW0, the value in the Y index register will be equal to the value of the key. However, with respect to ROW1 and ROW2 the values 07 and 0E respectively have to be added. Hexadecimal 0E is twice the value of 07. These additions are taken care of in the program section labelled KEYINC in figure 15a. However, before the program has reached that point, the contents of the X index register, the row information, is trans-





**Figure 15.** The detailed flowchart of the GETKEY subroutine which determines whether a valid key has been depressed and if so assigns the correct value to it. The program section labelled ONEKEY (see figure 12) has been repeated here for the sake of clarity.

ferred to the accumulator. Its value will be 23 for a key in ROW0, 25 for a key in ROW1 and 27 for a key in ROW2. This value is then masked by the instruction AND # 0F to remove the four most significant bits of the byte. The contents of the accumulator are shifted right one bit position, which effectively divides the value in the accumulator by 2. Thus, it follows that the contents of the accumulator now become 01 for a depressed key in ROW0, 02 for a key in ROW1 and 03 for a key in ROW2. This value is then transferred to the X index register and the contents of the Y index register are transferred to the accumulator. After the label KEYINC either the value 07 or 0E is added to the column number, depending on the row of the depressed key. The processor does not enter the loop KEYINC-BNE-KEYINC if the key was in ROW0, as the value in the X index register (after label KEYIND) will already be zero. When



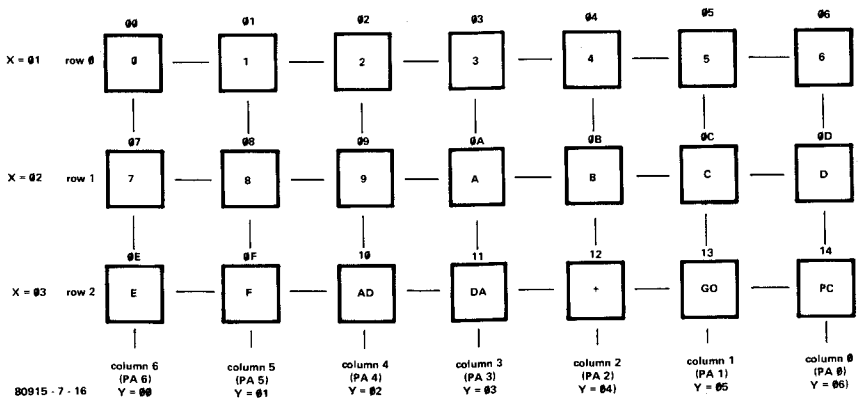
80915 - 7 - 15b

the processor returns from the GETKEY subroutine the value of the depressed key will be contained in the accumulator.

To fully understand how the subroutine works let us see what happens when key 'C' is depressed. To start with the X index register is loaded with the value 21 and the Y index register with the value 01. Then we jump to subroutine ONEKEY. Here the accumulator is loaded with FF and the value contained in the X index register (= 21) is transferred to the (output) lines of port B. This effectively grounds one side of all the switches connected to ROW0 via IC7. Key 'C' however, does not belong to ROW0, but to ROW1. Thus the following situation will occur when the keys in ROW0 are read via port A:

11111111	accumulator (A-FF)
x11111111	port A
x11111111	AND with FF
11111111	OR with 80
00000000	EOR with FF

Since the contents of the accumulator are zero, the processor does not



**Figure 16. The keyboard matrix showing the key values and the corresponding row and column data.**

branch to label KEYIN. During the subroutine ONEKEY the contents of the X index register will also have changed:

001000011 = 21 (initially)  
 001000010 = 22 (after first INX)  
 001000011 = 23 (after second INX)

Upon the return from ONEKEY the BNE instruction causes the processor to react to the last register contents, or rather to the last state of the Z flag. The last instruction to affect the Z flag was EOR #FF in the ONEKEY subroutine. Since the 'C' was depressed, the processor will not branch to KEYIN as the Z flag is high. The processor then tests to see whether the contents of the X index register are equal to 27 yet. But as we have only jumped to the ONEKEY subroutine once so far, the contents of the X index register will be 23. The processor therefore branches back to GETKEA, the Y index register is loaded with 01 once more and we jump back to ONEKEY. The accumulator is once again loaded with FF and the new contents of the X index register are transferred to the port B (output) lines. The common connection of all the keys in ROW1 will now be low. This is where key 'C' is situated and the bit pattern on port A will therefore be:

x1111101  
 PA1 = column 1 (see figure 16)

and after ANDing with PAD the accumulator contents will be:

x1111101

After ORing this with 80 the contents of the accumulator will be:

11111101

Finally, after the Exclusive OR with FF the accumulator will contain:

00000010

Meanwhile, the contents of the X index register have become 25. When the processor returns from ONEKEY the BNE instruction is encountered once again. Since the contents of the accumulator are no longer zero (00000010) the processor will branch to KEYIN. Here the accumulator contents are shifted left until the carry flag (C) is set:

carry flag		contents of Y register after ASL-A and before BCS	
x	00000010	FF	start
0	00000100	FF	after the 1st ASL-A
0	00001000	00	after the 2nd ASL-A
0	00010000	01	after the 3rd ASL-A
0	00100000	02	after the 4th ASL-A
0	01000000	03	after the 5th ASL-A
0	10000000	04	after the 6th ASL-A
1	00000000	05	after the 7th ASL-A

After seven shift instructions the carry flag has been set and the program branches to label KEYINB. Since the 'C' key is the one depressed, the contents of the X index register will be 23 and the contents of the Y index register will be 5.

The contents of the X index register are then transferred to the accumulator:

00100101	accumulator contents before ANDing with
00001111	0F
00000101	accumulator contents after ANDing
00000010	accumulator contents after LSR-A (= 02)

The contents of the accumulator (02) are again transferred to the X index register and the contents of the Y index register (05) are transferred to the accumulator. As the accumulator contents are not negative the processor branches to label KEYIND (via the BPL instruction). Here, the X index register is decremented by one, making its contents one. Since this is not zero the processor branches back to KEYINC, where the value 07 is added to the contents of the accumulator (the key column). This means that the value now contained in the accumulator is 05 + 07 = 0C, which is the final value of key 'C'. The contents of the X index register are then decremented once more to become zero, so the processor does not branch

back to KEYINC, but continues until it reaches the RTS instruction. The correct value of the depressed key will, therefore, be held in the accumulator after the return to main program execution.

*This brings us to the end of chapter 7. We have discussed a total of 181 bytes of the main monitor program together with 152 bytes of monitor subroutines. There are, of course, 1024 bytes of monitor program contained in the EPROM. The remaining subroutines will be discussed in the following two chapters concerning the editor and the assembler.*

# The Editor program

## The 'intelligence' behind simple program entry

**This chapter is devoted to the subroutines used by the editor section of the monitor program. What happens inside the Junior Computer when information is entered into the memory in the editor mode via the command keys INSERT, INPUT, DELETE etc.? How does the editor store a program so that it can be processed by the assembler? All these questions will be answered with the aid of flowcharts during the course of this chapter.**

As mentioned previously, the EPROM contains three separate sections: the main monitor program which controls address and data display, the editor and the assembler. Of these three it is the editor which takes up most memory space. Chapter 5 showed us how effective the editor routine can be when rather lengthy programs are entered into the computer. Now it is time to examine the editor section itself, which consists of a large number of subroutines, in much greater detail.

These subroutines are structured so that they can be incorporated into user programs if required, thereby saving a lot of time and effort when developing programs. One subroutine has already been introduced in Book I: the subroutine GETBYT, which 'reads' two hexadecimal keys and enters their value into the accumulator.

The editor responds to the five command keys SEARCH, INSERT, INPUT, SKIP and DELETE and to the data keys 0 . . . F. When one of the keys SEARCH, INSERT or DELETE is depressed, the processor expects either two, four or six data keys to follow. This is because the 6502 micro-processor instructions vary in length. They can be one, two or three bytes long. In the case of the SEARCH command, the Junior Computer requires four data keys to be depressed, as a specific 16 bit (two bytes) pattern must be traced in the (programmable) memory.

Before we deal with the main section of the editor and its associated subroutines, let us take a look at a general survey of the special characteristics

inherent to the editor which were not discussed in chapter 5. There we saw how the editor could be used, whereas this chapter will illustrate how the processor performs all the operations required by the editor routine(s).

### Characteristics of the editor

Before the editor can be run, the start and end addresses of the memory range in which the program is to be stored must be keyed in to the memory locations BEGADH, BEGADL and ENDADH, ENDADL respectively. It is possible to enter instructions into the Junior Computer by using the editor without having to activate the assembler afterwards. In this instance, however, the program must not contain any labels, as the microprocessor itself is not able to decode the label identifier FF. It follows, therefore, that if there are any labels in the entered program, the assembler will have to be started once editing is complete. As you know, the assembler removes all the labels from a program entered via the editor.

The editor also makes use of another pseudo-instruction that the 6502 microprocessor would otherwise ignore. This is the 'End Of File' (EOF) character 77. This is automatically entered after all the other program instructions. When using the editor to enter a program it is a good idea to leave at least six memory locations clear between the end of the program and the end address of the available memory (ENDADH, ENDADL). Otherwise it will be very difficult to start the assembler, as the symbol table prepared by the assembler could overwrite the tail end of the user program (see chapter 9 for further details).

Once the editor has been started, we can enter instructions into the Junior Computer with the command keys INSERT and INPUT. Whenever a complete instruction is keyed in, the occupied memory space will 'grow' by one, two or three memory locations. The exact amount is determined by the opcode of the entered instruction. At the same time, the EOF character will be shifted down by the same number of spaces. If, for example, a two byte instruction is keyed in, the EOF character will move down by two memory locations.

However, if the DELETE key is depressed, quite the opposite happens. The instruction shown on the display will be erased from the Junior Computer's memory, leaving a gap which the editor will fill by moving up the data block following the deleted instruction by the requisite number of address locations. Here again, the position of the EOF character will be adjusted: it will be moved up by the same number of bytes.

The SEARCH function is used to track down a particular two-byte 'pattern' within the defined memory area of the Junior Computer. This could be either an opcode plus first operand byte, or a label with the pseudo opcode FF plus the label number. The search starts off at the beginning of the defined memory area (BEGADH, BEGADL) and ends at the address containing the opcode of the double byte in question.

This also happens during the SKIP function, which traces the instruction immediately following the one currently on display. Both the SEARCH and SKIP functions can report an error. If, for example, the former is used to try to find a non-existent double-byte pattern, the display will show

EEEEEE until the SEARCH key is released. Then a random instruction will be shown on the display, which is where the Junior Computer stopped searching. The SKIP function also has this facility. The EOF character indicates the end of an entered data block made up from instructions and labels. If the processor discovers more instructions behind the EOF character, the message EEEEE will again be shown on the display until the SKIP key is released.

### The address pointers in the editor

The memory range which is used by the editor to store data and which runs from BEGAD to ENDAD is called a 'file'. This can be either a complete page or a data block. Four address pointers are required to take care of file management. They are:

1) **BEGAD**: This is stored in memory locations 00E3 and 00E2 – BEGADH and BEGADL respectively. This pointer thus establishes the start address of the file in which the editor is to store the program data.

2) **ENDAD**: This is stored in memory locations 00E5 and 00E4 – ENDADH and ENDADL respectively. This pointer therefore defines the end address of the file in which the editor is to store the program data.

3) **CURAD**: This is stored in memory locations 00E7 and 00E6 – CURADH and CURADL respectively. CURAD is an abbreviation for 'CURrent Address'. The editor requires this pointer to control the seven segment display on the Junior Computer. Since all the instructions in the file are shown on the display, the address pointer CURAD will change constantly during editing. It will always point to the instruction currently being displayed.

4) **CEND**: This is stored in memory locations 00E9 and 00E8 – CENDH and CENDL respectively. CEND is an abbreviation for 'Current END address'. Like CURAD, it will change constantly during the editing procedure. It will change whenever the command keys INSERT, INPUT or DELETE are used to enter or erase program instructions. The address pointer CEND is used to point to the address location immediately following the EOF character 77. If, for instance, the EOF character is located at address 0259, CEND will be pointing to location 025A.

The current address pointer CURAD and the end of program pointer CEND are initialised as soon as the editor is activated. In the case of a 'cold start entry' (see chapter 5) CURAD will point to the start address of the program file. This is where the EOF character 77 is stored initially, which is what is seen on the display each time a cold start entry into the editor is made. As no instructions have, as yet, been entered, CEND will point to BEGAD + 1. The initial situation as described is illustrated in figure 2a.

In addition, the editor features a 'warm start entry'. When the editor is started in this mode it operates as normal, but without affecting the current address pointer CURAD, the end of program pointer CEND or the EOF character. As discussed in chapter 5, the editor can be activated by depressing the GO key or the ST key. The start address for a cold start entry is ICD5. This address must be stored in the NMI vector if the editor is to be started with the ST key. The same is of course true of the warm



start entry. In this mode the editor is started from address 1CCA. The editor can be left with the aid of either the RST key or the ST key. The latter method will involve modifying the contents of the NMI vector (see chapter 5).

In the editing mode the display buffers will no longer contain address information and corresponding data, but instructions to be executed. These buffers are still called POINTH, POINTL and INH although they perform an entirely different function to that described in chapter 7. Data contained in POINTH will still be displayed on Di1 and Di2, data contained in POINTL will be displayed on Di3 and Di4 and the data contained in INH will be displayed on Di5 and Di6. The functions of the display buffers during the editing mode are as follows:

- \* During data entry the display is filled from left to right.
- \* The computer accepts data one byte at a time, so that two data keys have to be depressed before the entered byte appears on the display.
- \* When entering instructions, the computer will first evaluate the opcode to determine the length of the instruction. This is to see whether the instruction is one, two or three bytes long. Once the opcode has been evaluated therefore, the computer will know how many displays to enable (Di1 . . . Di2, Di1 . . . Di4 or Di1 . . . Di6).

During the editing routines the display buffers will contain the following:

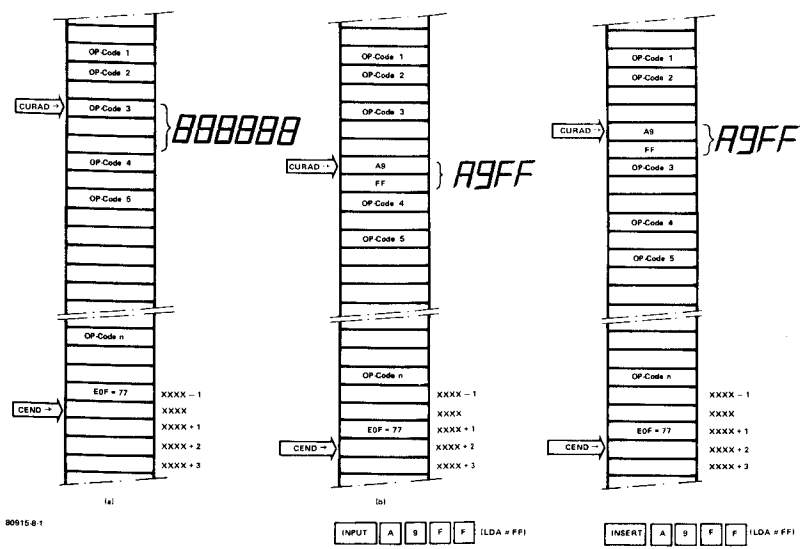
**POINTH:** will always contain the opcode of an instruction or the pseudo opcode of a label.

**POINTL:** will always contain the first operand byte (if present) or the label number.

**INH:** will always contain the second operand byte (if present) or the limiter byte of a label.

Before considering the editor software in detail, it may be as well to say a few words about the functions INSERT, INPUT and DELETE. The initial situation is illustrated in figure 1a. Here the address pointer CURAD is pointing to opcode 3. This opcode, together with the two operand bytes following it, is shown on the display. The EOF character is located at address  $xxxx - 1$  and the end of program pointer is pointing to address location  $xxxx$ . If the programmer wishes to enter a new instruction such as LDA #FF, for instance, immediately after the one currently on display, the INPUT key must be used (see figure 1b). The hexadecimal code A9FF will then be loaded into the address locations behind the instruction that was previously on display (opcode 3 plus the two operand bytes) and the processor will move all subsequent bytes down by two memory locations. The EOF character will also move down two places as the instruction LDA #FF is two bytes long. If, however, the programmer wishes to enter the new instruction immediately before the one currently on display, the INSERT key must be used. This is shown in figure 1c. Here the hexadecimal code for the instruction LDA #FF is placed in front of the instruction that was previously on display. Again, all subsequent bytes (and the EOF character) are moved down two address locations. In both instances, the EOF character will be located at address  $xxxx + 1$  and CEND will point to location  $xxxx + 2$ .

In chapter 5 we mentioned the fact that during cold start entry the first instruction must always be entered by means of the INSERT key. The



**Figure 1. The difference between the INPUT and INSERT functions. When an instruction is entered into the file using the INPUT command (1b) it is placed behind the instruction currently on display. When the INSERT command is used (1c) it is placed in front of the current instruction. Figure 1a shows the initial situation, where CURAD is pointing to opcode 3.**

reason for this is illustrated in figure 2. The initial situation is given in figure 2a. From figure 2b it can be seen what happens when the label FF 15 00 is entered into the Junior Computer memory using the INPUT key. The EOF character will remain where it was and the label FF 15 00 will end up behind it – with disastrous results!

The correct procedure for a cold start entry is given in figure 2c. Here the programmer uses the INSERT key before keying in the label. This means that the label is correctly positioned in front of the EOF character. As the label is three bytes long, the EOF character has moved down three address locations. At this moment in time the complete file consists of the label FF 15 00 and the EOF character 77, starting at BEGAD and ending at CEND. The more instructions that are stored in the file, the further down the EOF character will drop. The CEND pointer will then also be moved further away from the BEGAD pointer, indicating that the file is growing all the time.

The effect of the DELETE function is illustrated in figure 3. Let us suppose that the instruction LDA # FF is situated somewhere in the file and is followed by the label FF 3B 00. We now wish to erase the instruction LDA # FF (hexadecimal code A9FF) from the file. Once the instruction has been located, using the SEARCH function for instance, the display will show the code A9FF and the DELETE key can be depressed. This has the opposite effect of the INPUT and INSERT keys in that the

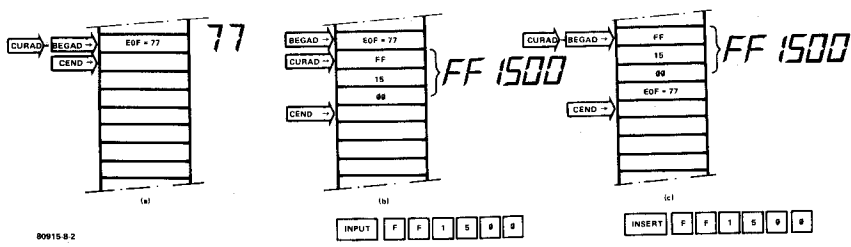


Figure 2. When the editor is started via a cold start entry, the pointers CURAD and BEGAD will both indicate the EOF character 77 (2a). If the INPUT function is used to enter the label FF 15 00 (2b) it will be stored after the EOF character, which is, of course, incorrect. The correct procedure is to use the INSERT function (2c) so that the label is stored before the EOF character.

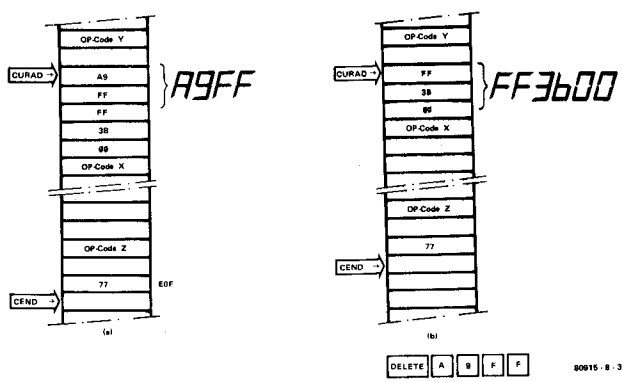


Figure 3. When an instruction is to be erased from the file, the DELETE function is used. This effectively shifts the entire data block upwards in the file so that the unwanted instruction is overwritten.

instruction currently on display is erased from memory and the complete section of file following it is moved up two locations. This means, of course, that the EOF character and the CEND pointer also move up two places. The file will now be two bytes shorter than it was previously.

### The basic flowchart of the editor

The basic flowchart of the editor section of the monitor program is shown in figure 4. This can be compared, to a certain extent, to that given in figure 1 of chapter 7. As can be seen, there are various similarities and various differences between the two.

Taking the similarities first: the 'warming up session' required before the computer can operate in the editor mode can be likened to the RESET

routine which initialises the main section of the monitor. Furthermore, the central label CMND in the editor resembles the START label in the main monitor section. Even the individual operations that the processor performs between the labels CMND and SEARCH can be compared to block C in figure 1 of chapter 7: in both sections the computer scans the keyboard and multiplexes the display. After the SEARCH label in figure 4 the processor scans each of the command keys (SEARCH, INSERT, INPUT, SKIP and DELETE) in turn to ascertain which of the functions the programmer wishes to execute.

This covers the similarities between the two basic flowcharts. What about the differences? In contrast to the main monitor routine, the editor has no clear exit point from which it can be left. Once the editor is up and running by way of a cold or warm start entry, it cannot be left by depressing one of the command keys as in the case of the GO key in the main monitor routine. As far as the computer is concerned, the editor is an infinite loop which can only be exited from by depressing one of the hardware keys RST or ST.

The label ERRA constitutes a further difference. It belongs to a section of program in the editor which causes the 'text' EEEEEEE to be displayed whenever the programmer is guilty of an operational error. The operation of the main monitor routine is so elementary that there is no real need for any error messages. However, when instructions are entered in the editing mode, it is very easy to make mistakes which the computer must bring to the attention of the programmer.

Errors will be reported in the following instances:

- \* The programmer uses the SEARCH function to trace a certain two-byte pattern which is supposedly situated somewhere in the file. If this particular pattern turns out to be non-existent, the Junior Computer will display the error message EEEEEEE until the SEARCH key is released.

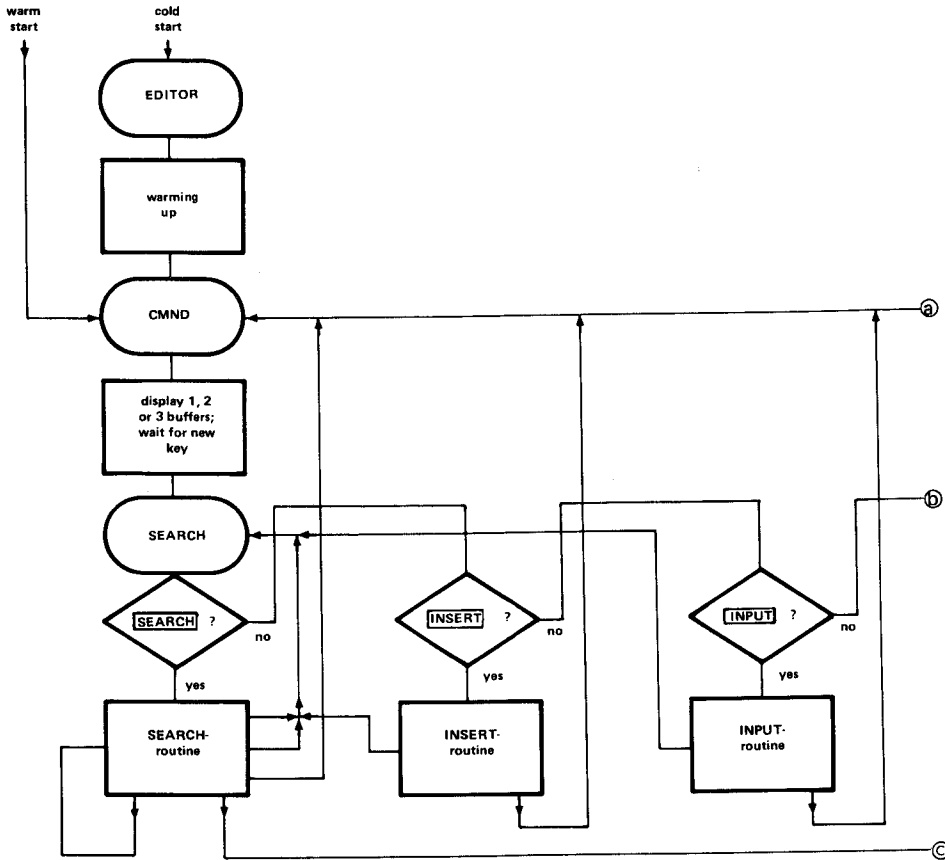
- \* The programmer uses the SKIP function to quickly run through a file. Eventually the programmer reaches the EOF character, but inadvertently depresses the SKIP key once more. As we know, the file must end after the EOF character. Therefore, the computer will once again display EEEEEEE as an indication that the programmer has exceeded the boundaries of the file which extends from BEGAD to CEND.

- \* The computer will also report an error whenever the programmer depresses the wrong key. An example of this is when the programmer depresses a data key when the processor is waiting for a command key.

There is one final difference between the two basic flowcharts. It is clear from figure 4 how the various command keys are tested, but where does the computer process the data keys 0 . . . F when the machine is in the editing mode? It is common knowledge by now that the data keys are intended for the purpose of entering instructions, in other words opcodes and operand bytes. In fact, the hexadecimal keys are taken care of during the SEARCH, INSERT and INPUT routines, which is the reason for all the branches back to the SEARCH label in the basic flowchart (figure 4). During these routines the GETBYT subroutine, which is familiar to us from Book I, is used to fetch the information from two data keys depressed in succession and to combine the data to form a single byte. The GETBYT subroutine will only accept hexadecimal keys and will ignore

command keys. As a result, it is possible for the programmer to switch over to command entry during data entry (by depressing one of the keys SEARCH, INPUT, INSERT, SKIP or DELETE) without the wrong data being stored in the file.

The contents of the three display buffers will not be copied from the display into the file until the programmer has entered the entire instruction. If the programmer therefore depresses a command key while entering an instruction, the editor program will branch to the SEARCH label and will then take care of the new command. If, however, the programmer has typed in the whole instruction, the program will branch to the centre label CMND. The editor will then wait for a new command key to be depressed and will once again branch to the relative command key routine.

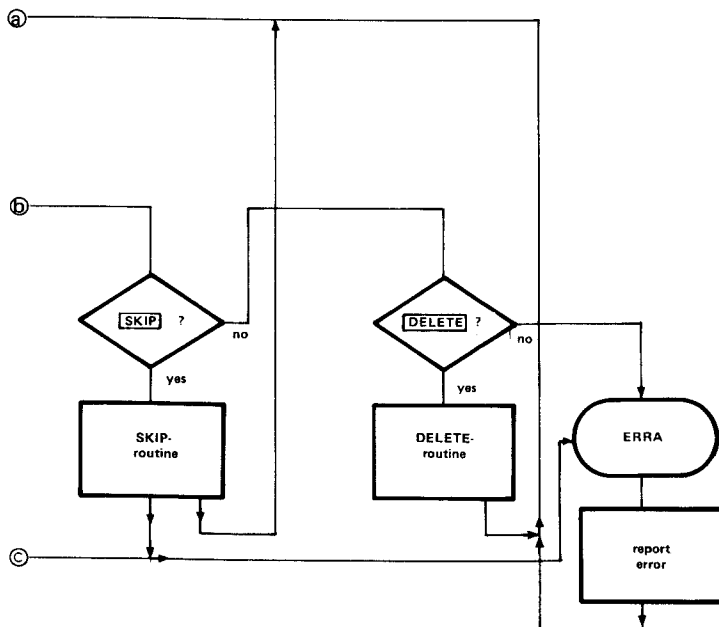


## The detailed flowchart of the editor

Now that we have discussed the basic flowchart of the editor, it is time to take a much closer look at the routines etc. involved. The detailed flowchart of the editor program is given in figure 5, albeit without the initialisation routine required for a cold start entry. This routine, which prepares the memory of the Junior Computer for data entry is given in figure 6.

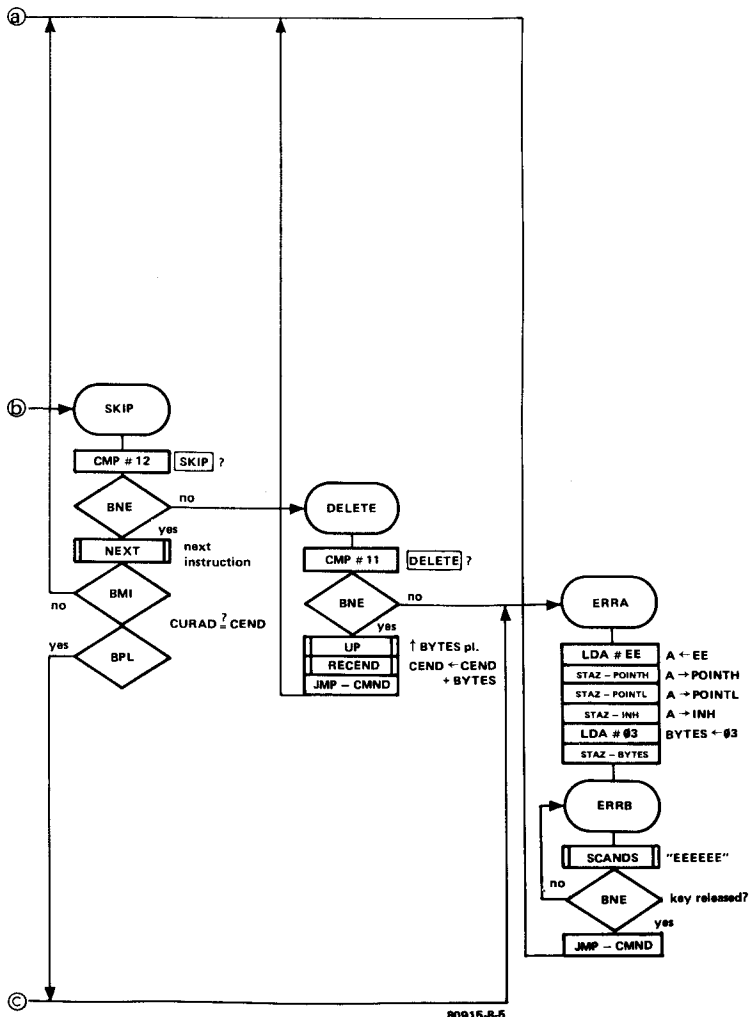
As can be seen from the two figures, the editor is made up from a number of subroutines which we will get to understand more fully when discussing the overall program. By the way, as with most of the subroutines contained in the monitor EPROM, it is possible to incorporate these routines in your own programs. This can, in fact, save a great deal of time and effort when developing programs.

Figure 4. The basic flowchart of the editor section of the monitor program.



80915-8-4





80915-8-5



## Cold start entry

The detailed flowchart of the section of program that is executed when the editor is activated via a cold start entry is given in figure 6. The editor program starts with the subroutine BEGIN, which is shown in detail in figure 7. This consists of only four instructions, but it is extremely important for both the editor and the assembler. The BEGIN subroutine ensures that the contents of the display pointer CURAD are the same as the contents of the address pointer BEGAD. As you know, BEGAD points to the start address of the file where the programmer intends to store his/her series of instructions and labels.

After returning from the BEGIN subroutine, the processor makes the contents of the variable end address pointer CEND equal to BEGAD + 1 with the aid of the X and Y index registers. This is quite logical when you

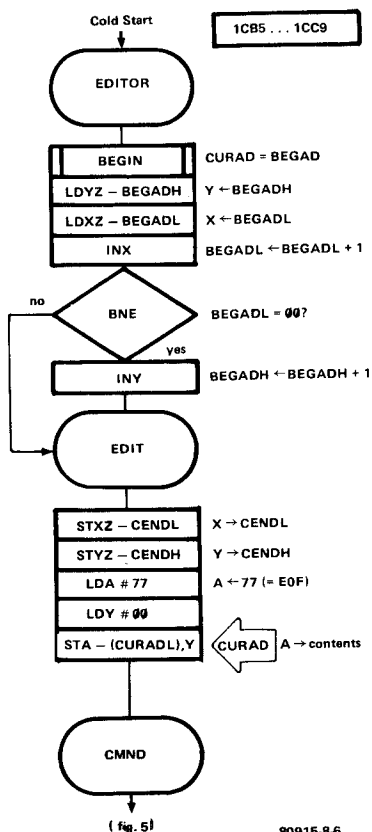
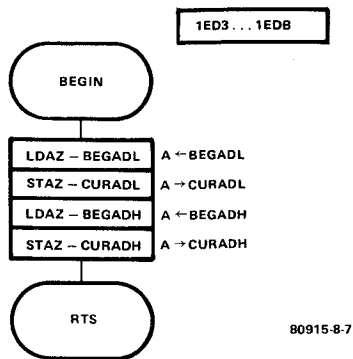


Figure 6. The detailed flowchart of the section of program that is executed when the editor is started via a cold start entry.



**Figure 7.** The subroutine **BEGIN** simply ensures that the contents of the pointers **CURAD** and **BEGAD** are equal.

consider that the memory location being pointed to by **CURAD** is to be loaded with the EOF character 77 and that **CEND** is to point to the memory location following the EOF character.

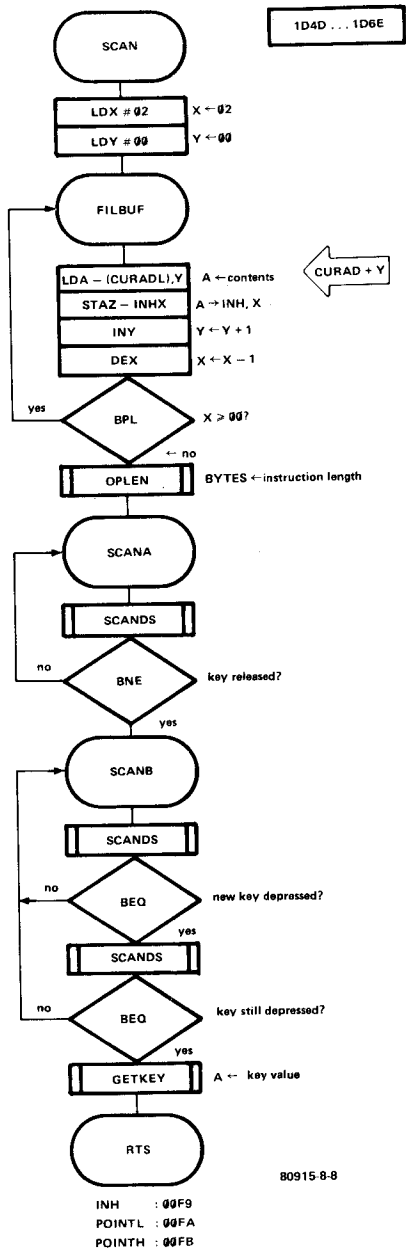
How is **CEND** made equal to **BEGAD + 1**? First of all, the contents of the X index register (= **BEGADL**) are incremented by one. If the contents of **BEGADL** were FF prior to this, the contents afterwards will be 00. This, of course, means that the contents of **BEGADH** will also have to be increased by one. The processor checks out this situation by means of a **BNE** instruction and if true the contents of the Y index register (= **BEGADH**) are also incremented. Since the pointer **BEGAD** is not allowed to change during the entire editor routine, the two index registers are used as temporary storage locations. The two subsequent instructions (**STXZ-CENDL** and **STYZ-CENDH**) ensure that **CEND = BEGAD + 1**. The three final instructions in this section of program simply store the EOF character 77 in the memory location being pointed to by **CURAD**. Now the initial situation as given in figure 2a will have been achieved.

## Keyboard and display

### *the SCAN subroutine*

The program section between the labels **CMND** and **SEARCH** in figures 4 and 5 is restricted to scanning the keyboard and multiplexing the displays. This is where the processor determines whether one, two or all three display buffers are to be enabled and whether any key in the Junior Computer keyboard matrix has been depressed. Once a depressed key has been detected, the processor will calculate its value in the usual manner. These tasks are carried out in the subroutine **SCAN**, the detailed flowchart of which is given in figure 8.

At the beginning of this subroutine, the X index register is loaded with 02 and the Y index register is loaded with 00. As a result, from the label **FILBUF** onwards, the contents of three consecutive memory locations



**Figure 8.** The SCAN subroutine is responsible for scanning the keyboard for a depressed key and multiplexing the display. At the end of this subroutine the value of the depressed key will be held in the accumulator.

contained in the file are copied into the three display buffers POINTH, POINTL and INH as follows:

1. X = 02 Y = 00: contents of CURAD → POINTH (00FB)
2. X = 01 Y = 01: contents of CURAD + 1 → POINTL (00FA)
3. X = 00 Y = 02: contents of CURAD + 2 → INH (00F9)
4. X = FF; the subsequent BPL instruction does not lead to another branch and the processor reaches the subroutine OPLEN. This subroutine is almost identical to the LENACC subroutine described in Book I. The OPLEN subroutine calculates the length of the instruction from the opcode that the display pointer CURAD is pointing to. After returning from OPLEN the length of the instruction to be displayed will be stored in the RAM location BYTES (00F6).

Once the length of the instruction is known, the processor can determine whether it should only display the contents of POINTH, the contents of POINTH and POINTL or the contents of POINTH, POINTL and INH.

As mentioned previously, the subroutine SCAND looks after the scanning of the keyboard and the multiplexing of the display. The subroutine SCANDS is also familiar to us from chapter 7, for it is the subroutine SCAND minus the first program section. In that section of program the display buffer INH is loaded with the contents of the memory location being indicated by the address pointers POINTH and POINTL. The processor does not require this particular section in the editor mode. Since SCANDS and the associated subroutines were all described in chapter 7, there is no need to go into them again. You can always refer back to that chapter to refresh your memory.

Once the subroutine OPLEN has been completed the processor moves on to label SCANA. This section of the program up to the final (RTS) instruction has also been described in chapter 7. Just to recap, this section of the program waits for a key to be depressed, debounces the depressed key and finally calculates the values of the key via the subroutine GETKEY. Upon the return from GETKEY the value of the depressed key will be held in the accumulator. The editor command keys have the following values:

SEARCH : key value 14 (same as the PC key)  
INSERT : key value 10 (same as the AD key)  
INPUT : key value 13 (same as the GO key)  
SKIP : key value 12 (same as the + key)  
DELETE : key value 11 (same as the DA key).

The testing of the various keys is accomplished with the aid of a CMP instruction followed by a BNE instruction. When a command key is detected, the processor will not branch at the next BNE instruction and the computer will concentrate on dealing with the particular task pertaining to the depressed key (see figures 4 and 5).

## Reporting an error

*EEEEEE!*

In the event of an operational error, the program section following the label ERRA in figure 5 will be executed. We have already discussed the

possible causes of an error report. The first group of instructions in this section of the program simply enters the hexadecimal value FF into the three display buffers POINTH, POINTL and INH. The memory location BYTES is then loaded with the value 03, as when an error is reported all six digits of the display must be enabled. The program section continuing on from label ERFB consists of a wait loop where the keyboard and display subroutine, SCANDS, is executed. The display will continue to show EEEEE until the key which caused the error to be reported is released. Once this has been done the processor will jump back to the central label CMND.

### The SEARCH routine

As would be expected, whenever the SEARCH key is depressed the routine labelled SEARCH is executed (see figure 5). We already know that the value of the depressed key is held in the accumulator when the processor returns from the SCAN routine. The computer then determines whether the SEARCH key was depressed with the aid of two instructions immediately following the SEARCH label (CMP # 14, BNE). The program section pertaining to the SEARCH function has to meet the following requirements:

1. It must be able to trace any two-byte pattern situated anywhere in the file.
2. The search should start at BEGAD and end as soon as the required bit pattern has been found.
3. If the required two-byte pattern can not be traced between BEGAD and CEND, the computer must report an error.

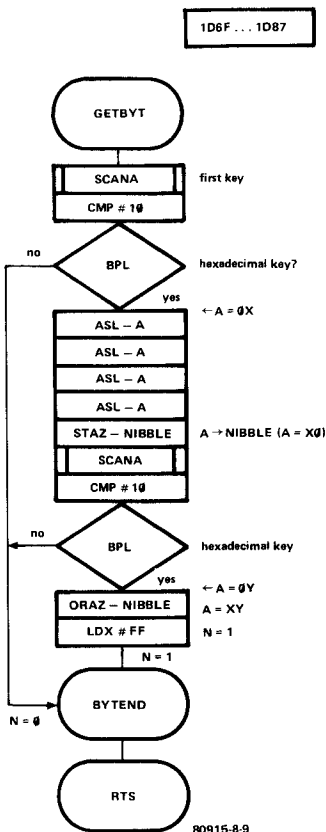
The SEARCH routine starts by calling the subroutine GETBYT twice. Although this subroutine is familiar to us from Book 1, the finer details have yet to be discussed.

### The subroutine GETBYT

The detailed flowchart of the GETBYT subroutine is given in figure 9. Its task is to combine the values of two depressed data keys into a single byte and hold the result in the accumulator. This subroutine will only accept hexadecimal keys and will ignore command keys.

The values of the two depressed keys are stored as one byte in the accumulator by shifting the value of the first one into the four most significant bits of the accumulator and ORing it with the value of the second. At the beginning of the GETBYT routine, the subroutine SCANA is called up. This is the second section of the subroutine SCAN illustrated in figure 8, which a. establishes whether the old key has been released; b. establishes whether a new key has been depressed; c. calculates the value of the new key.

Command keys have a value of 10 or higher, whereas hexadecimal keys have a maximum value of 0F. Therefore, the two instructions (CMP # 10, BPL) at the start of the GETBYT routine determine which type of key was depressed. If a command key was depressed, the value of the N flag will be zero and the processor will branch back to the SEARCH routine



**Figure 9.** The subroutine GETBYT combines the values of two depressed keys into a single byte and holds the result in the accumulator.

via the RTS instruction. If, on the other hand, a hexadecimal key was depressed, the processor will return from SCANA with the value of that key stored in the accumulator. If this was the first data key, the contents of the accumulator will be changed from  $0x$  to  $x0$  (where  $x$  denotes the value of the depressed key) after the four shift operations (ASL-A). This value is then stored in the temporary location NIBBLE ( $00FE$ ).

In order to read the second data key the subroutine SCANA is called once more. (This is why the contents of the accumulator were saved beforehand). When the processor returns from SCANA the second time a further test is carried out (CMP#10, BPL) to determine whether the second depressed key is a command key or a hexadecimal key. In the case of the latter, the routine will ignore it and the processor will again branch back to the SEARCH routine.

If the second key happens to be a data key, with an assumed value of  $\emptyset y$ , the data byte in the accumulator will become  $xy$  after the instruction ORAZ-NIBBLE. Here  $x$  represents the value of the first hexadecimal key and  $y$  represents the value of the second data key. The final instruction in the GETBYT subroutine loads the value FF into the X index register, which sets the N flag. Now it can easily be determined whether a command key or a data key was depressed during GETBYT by examining the N flag.

It should be added at this point that the subroutine GETBYT is very versatile. The programmer will soon see just how easily this subroutine can be incorporated into his/her own programs.

Now to get back to the SEARCH routine shown in figure 5. After the subroutine GETBYT is called the first time, the N flag is tested to see whether a command key was depressed. This lets the computer know whether or not to branch back to the label SEARCH by way of the BPL instruction.

If two data keys were depressed during the GETBYT subroutine, they will both form the part of the two-byte pattern to be searched for. This will then be transferred to the display buffer POINTH and will be displayed the second time that GETBYT is called. While the computer is servicing the GETBYT routine the second time it will wait for the remaining half of the two-byte pattern to be entered. If, during this routine, a command key is depressed, the processor will once again branch back to SEARCH and will wait for a completely new two-byte pattern to be keyed in.

However, if two data keys are depressed, they will constitute the second half of the bit pattern to be looked for. The combined value of these two keys will then be stored in the display buffer POINTL. This means that the two display buffers, POINTH and POINTL, have become the temporary storage locations for the particular two-byte pattern to be traced.

At this stage the processor knows what pattern to look for between the labels BEGAD and CEND and the search can commence. To start with, the subroutine BEGIN is called, which makes the contents of pointer CURAD equal to those of BEGAD. The following section of the program, starting at label SELOOP, is executed until the required bit pattern turns up in the file. Once this has occurred, the processor will branch back to the central point CMND and the pattern will appear on the display during the following subroutine SCAN. Should the pattern in question prove to be non-existent, the processor will branch from SELOOP to label ERRA to report an error.

## The search

How is the search for the particular two-byte pattern carried out? What happens is that the processor compares all the instructions and labels in the file to the bit pattern stored in the two display buffers POINTH and POINTL.

The section of program concerned commences at label SELOOP. Here the contents of the Y index register are made to equal  $\emptyset\emptyset$ . During this routine the Y register is used as an index to access the various instructions and/or labels contained in the file. The accumulator is then loaded with the contents of the location indicated by the current address pointer

(CURAD) and compared with the contents of the display buffer POINTH. If the contents of POINTH are **not equal** to the opcode or label identifier pointed to by CURAD, then the length of the instruction (or label) must be calculated and the current address pointer updated accordingly. This is accomplished by the subroutines OPLEN and NEXT respectively. During the subroutine OPLEN address location BYTES is loaded with the length of the instruction pointed to by CURAD. During the subroutine NEXT (see figure 19) the contents of BYTES are added to the contents of CURAD and a check is made to see whether the value of CURAD is already greater than the value of CEND. If this is so, the processor will branch to ERRA to report an error. This will indicate that the bit pattern being searched for does not exist in the file.

If the contents of POINTH are **equal** to the opcode or label identifier pointed to by CURAD, the contents of the Y index register are incremented by one. Then the contents of the location indicated by CURAD + 1 is compared to the contents of the display buffer POINTL. Should the data contained in the address location pointed to by CURAD + 1 be equal to that contained in POINTL, then the required bit pattern will have been found. The processor will then branch back to CMND to display the bit pattern. Thus, the SEARCH function allows labels or instructions of any length to be tracked down anywhere in the file.

If the contents of POINTL are not equal to the contents of the location indicated by CURAD + 1, the search must continue. Once again the length of the label or instruction is determined by the subroutine OPLEN and the contents of the current address pointer updated accordingly. Provided the contents of CURAD are less than those of CEND the processor will branch back to SELOOP to continue the search for the required bit pattern.

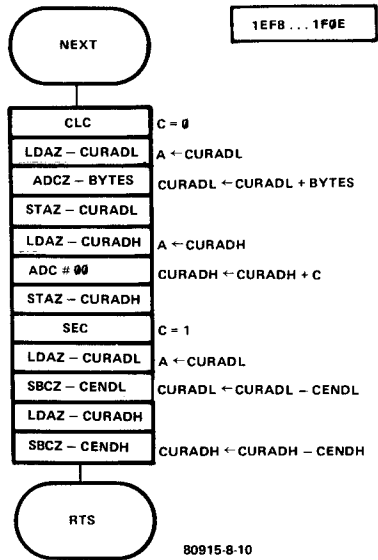
Thus, the SEARCH function compares two pairs of bytes with each other. The two stored in POINTH and POINTL are compared to those stored in the locations indicated by CURAD and CURAD + 1 respectively. In the subroutine NEXT the CURAD pointer is updated so that it now points to the location containing the next byte of the instruction (or label number) to be examined. Although what happens during the NEXT subroutine should be perfectly clear, it may be a good idea to examine the details a bit closer.

## The subroutine NEXT

The detailed flowchart of the NEXT subroutine is given in figure 10. After the carry flag has been reset, the low order byte of the CURAD pointer is increased by the value contained in location BYTES. The latter location contains the length of the instruction currently on display. Following this the value 00 plus the contents of the carry flag are added to the high order byte of CURAD. This ensures that the 16 bit addition of CURAD + BYTES results in the pointer indicating the next instruction (or label) to be examined.

The second part of the NEXT subroutine is devoted to testing whether or not the current address pointer has exceeded the file boundary imposed by





**Figure 10.** The NEXT subroutine moves the current address pointer down the file by the number of memory locations contained in location BYTES. In addition, it checks whether the CURAD pointer is greater than the CEND pointer.

the contents of the current end address pointer (CEND). This involves a 16 bit subtraction, which is virtually the opposite of the previous addition. Firstly, the carry flag is set. Then the low order byte of CEND is subtracted from the low order byte of CURAD. Finally, the high order byte of CEND is subtracted from the high order byte of CURAD. The result of the complete subtraction will affect the N flag in the status register. If, after the return instruction, the N flag is reset ( $N = 0$ ), the contents of CURAD are greater than those of CEND and the processor must report an error. If, however, the N flag is set ( $N = 1$ ), the contents of CURAD are less than those of CEND and all is well. The processor will then return to SELOOP and continue the search. The state of the N flag is tested upon the return from NEXT with the aid of a BMI (Branch if Minus) instruction which causes the processor to branch back to SELOOP if the N flag is set. If the N flag is not set it must be reset and therefore the BPL instruction directs the processor to ERRA.

### The INSERT routine

To understand how the INSERT routine works we need to go back to figure 5, the editor flowchart, once more. The first two instructions (CMP # 10 and BNE) determine whether or not the INSERT key was depressed. If the computer returns from the SCAN routine with the value 10 in the accumulator, the INSERT key will have been depressed and the

INSERT function must be carried out. The INSERT routine performs the following task:

1. It reads an instruction entered from the keyboard and places it in the display buffer. At the same time, the length of the instruction is determined.
2. Once the entire instruction has been entered into the display buffer, it must be copied into the work memory immediately before the instruction currently on display. This obviously involves moving the remainder of the file downwards by the corresponding number of bytes.

Two subroutines have been provided in the editor program to meet the above requirements: the subroutine RDINST (= ReaD INSTRUCTION) and FILLWS (= FILL Work Space). What exactly do these subroutines do?

### The subroutine RDINST

The detailed flowchart of the RDINST subroutine is given in figure 11. The purpose of this subroutine is to enter a one, two or three byte instruction into the display buffer of the computer. It starts by calling up another subroutine, GETBYT. This, as we know, enables two hexadecimal keys to be read into the computer. These will correspond to the opcode of the instruction to be entered. Upon the return from GETBYT, the opcode will be held in the accumulator and from there will be transferred into the display buffer POINTH. If the programmer depressed a command key during the GETBYT routine, the processor will return to the editor main routine in the usual manner with the N flag in the status register reset ( $N = \emptyset$ ).

Once the opcode of the instruction has been transferred to POINTH, the processor is able to determine the length of the instruction. For this purpose, the subroutine LENACC is called, which is part of the subroutine OPLEN we mentioned before. (These two subroutines will be discussed in greater detail towards the end of this chapter). Once the length of the instruction has been obtained, it is stored in three memory locations: BYTES, COUNT and TEMPX.

After decrementing memory location COUNT, the processor tests to see whether the opcode is followed by any operand bytes which are also to be read into the display buffer. If the instruction is only one byte long, the processor will branch to the label RDA where the X index register is loaded with the value FF. This instruction causes the N flag to be set.

If the computer returns from the subroutine RDINST with the N flag set, the entire instruction will have been entered into the display buffer. This can now be copied into the work memory in front of the instruction previously on display.

If the processor returns to the editor main program from the subroutine RDINST with the N flag reset, the programmer must have depressed a command key when entering the instruction. Part of the instruction will now be in the display buffer, but this must not be transferred to the computer work memory.

By setting or resetting the N flag during the subroutine RDINST the processor is able to check on its return to the main editor routine whether the



entered instruction is to be copied from the display buffer into the file, or whether the computer should deal with the new command function.

Now then, back to what happens when an instruction of more than one byte is read in. If after the first time the contents of memory location COUNT are decremented they are not equal to zero, the instruction will either be two or three bytes long. For this reason the subroutine GETBYT is called once more, during which the first operand byte of the instruction is read into the accumulator. This is then stored in the display buffer POINTL after which the contents of location COUNT are again decremented. If this becomes equal to zero at this point, the instruction to be entered will be two bytes long and the processor will return to the editor main routine with the N flag set. If, however, the contents of COUNT are still not equal to zero, the instruction concerned will be three bytes long. The subroutine GETBYT will, therefore, be called a third time. The processor is then able to read in the second operand byte of the instruction. Once this is accomplished, it is stored in the display buffer INH. The processor then returns to the main editor routine with the N flag set.

Upon the return from the RDINST subroutine the state of the N flag is tested with the aid of the BPL instruction, which causes a branch to SEARCH if the N flag is reset. If the N flag is set at this point in the INSERT routine the processor is directed to the subroutine FILLWS.

### **Subroutines FILLWS and ADCEND**

The task of the FILLWS subroutine is to transfer the contents of the display buffers into the computer file and update the current end address pointer CEND. The latter is in fact accomplished during the subroutine ADCEND — more about this later.

The detailed flowchart of the FILLWS subroutine is shown in figure 12. At the beginning of this routine the subroutine DOWN is called (see figure 15). The DOWN subroutine will come up for discussion later as it is a relatively complicated subroutine. For the moment it is sufficient to know that the DOWN subroutine makes space available in the computer work memory for the new instruction to be inserted. We already know that the current address pointer, CURAD, indicates the address of the instruction currently on display. Since a new instruction can be one, two or three bytes long and must be inserted before the instruction shown, the subroutine DOWN must move the entire data block between CURAD and CEND down by the corresponding number of bytes. As soon as space has been made, the new instruction can be copied from the display buffer into the file.

Prior to the subroutine FILLWS being called (see figure 5), the processor will have calculated the length of the instruction and stored the result in location BYTES. When the processor jumps from FILLWS to DOWN, the data block between the pointers CURAD and CEND is then moved down by the number of memory locations contained in BYTES. If, for instance, the entered instruction is two bytes long, the hexadecimal number 02 will be contained in location BYTES. The subroutine DOWN will then move the block of data between the pointer CURAD and CEND down by two

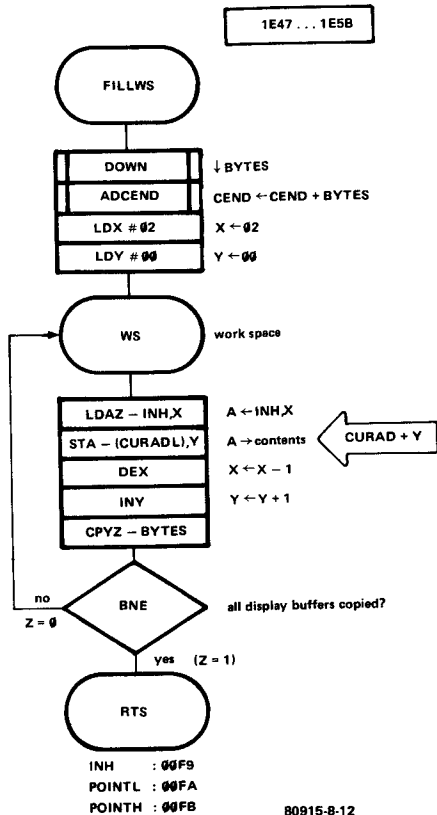
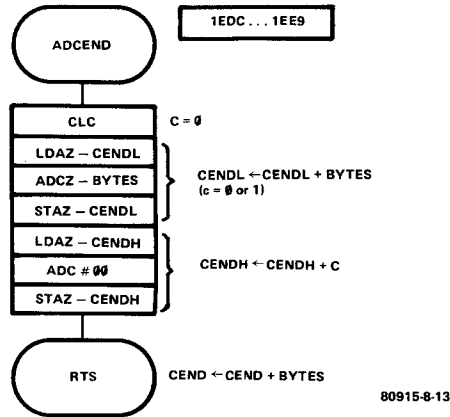


Figure 12. The subroutine FILLWS makes room for a new instruction to be copied from the display buffers into the file.

memory locations.

It should now be fairly clear how the subroutine DOWN helps to make room for a new instruction to be copied from the display buffer into the file. Upon returning to the subroutine FILLWS (figure 12), the next stage is to update the current end address pointer, CEND. This is the task for the subroutine ADCEND (= ADvance Current END address pointer = move CEND pointer down by the value of BYTES). The detailed flowchart of the ADCEND subroutine is given in figure 13. This routine simply adds the contents of location BYTES to the address pointed to by CEND. The 16 bit addition is performed in exactly the same way as for CURAD in the NEXT subroutine. Firstly, the carry flag is cleared. The value contained in location BYTES is then added to the low order byte of CEND. Finally, the value 00 plus the contents of the carry flag are added to the high order byte of CEND.



**Figure 13.** The ADCEND subroutine is called by the subroutine FILLWS (figure 12). Its function is to increase the value of CEND by the contents of location BYTES — move the CEND pointer down by the number of bytes corresponding to the length of the new instruction.

As soon as the processor returns from the ADCEND subroutine, the X index register is loaded with the value 02 and the Y index register with the value 00. The process that follows, after the label WS, simply copies the contents of the display buffers into the file starting from CURAD and ending when the value in the Y index register is equal to that of BYTES. Both pre-indexed and post-indexed indirect addressing are used in this section of the program. The procedure can be summarised as follows:

1. The contents of  $INH + 02$  (= POINTH = the opcode of the instruction to be entered) are transferred to the contents of  $CURAD + 00$  ( $X = 02$ ,  $Y = 00$ ). The contents of the X index register are then decremented and those of the Y index register are incremented. The value in the Y index register is then compared with the contents of BYTES to check whether there are any further operand bytes to be transferred. If there are:
2. The contents of  $INH = 01$  (= POINTL = the first operand byte of the instruction to be entered) are transferred to the contents of  $CURAD + 01$  ( $X = 01$ ,  $Y = 01$ ). Again, the contents of the X index register are decremented and those of the Y index register incremented. Then the value contained in the Y index register is once more compared to the contents of location BYTES to check whether a further operand byte is to be transferred. If so:
3. The contents of  $INH + 00$  (= INH = the second operand byte of the instruction to be entered) are transferred to the contents of  $CURAD + 02$  ( $X = 00$ ,  $Y = 02$ ). The contents of the X index register are again decremented and those of the Y index register incremented. When the contents of the Y index register are compared to those of location BYTES this time they will have to be equal as the contents of BYTES can not exceed 03. This means that the processor will not effect a branch this time, but will continue to the return instruction.

Once the contents of all the display buffers have been inserted into the file between CURAD and CEND, the processor will leave the FILLWS routine and go back to the INSERT routine (see figure 5). The Z flag will always be set at this stage and so at the end of the INSERT routine the processor branches back to the central point CMND via a BEQ instruction. The editor will then continue to wait for a new command key to be entered.

### The INPUT routine

The INPUT routine is very similar to the INSERT routine, therefore, apart from a few minor differences which we'll examine here. The purpose of the routine was explained in chapter 5 and is: to read in a new instruction from the keyboard into the display buffer and copy the new instruction into the file immediately after the one currently on display.

The INPUT routine is also illustrated in figure 5. The first two instructions (CMP # 13 and BNE) determine whether or not the INPUT key was depressed. If so, the subroutine RDINST is executed which, as we know, reads an instruction entered from the keyboard and copies it into the display buffer. If a command key is depressed during this subroutine, the processor will branch back to the central label CMND via the following BPL instruction.

If, on the other hand, the entire instruction has been entered, the subroutine OPLEN will be called. At this moment the current address pointer will still be pointing to the 'old' instruction in the file. The computer then determines the length of this instruction with the aid of the OPLEN subroutine. Following this the subroutine NEXT is called, which moves the current address pointer, CURAD, down by the equivalent number of bytes to the length of the old instruction shown on the display.

The pointer CURAD will now indicate the address where the new instruction should be stored in the file. After the return from the NEXT subroutine, the contents of location TEMPX are transferred to location BYTES. As you know, the contents of location BYTES represent the length of an instruction. What, however, does the location TEMPX contain? To find out, let us reconsider part of the RDINST subroutine (figure 11). During this subroutine the instruction length information is stored in location TEMPX after the return from LENACC. The processor uses this information in the INPUT subroutine before jumping to the FILLWS subroutine. During the latter routine the new instruction is inserted immediately after the old one. As the current address pointer, CURAD, has not altered since the NEXT subroutine, it will already be pointing to the newly entered instruction upon the processor's return to the central CMND point. The SCAN subroutine proceeds to display the new instruction and the computer will wait for a further command key to be depressed.

### The SKIP routine

The purpose of the SKIP subroutine (as mentioned in chapter 5) is to check through a program that has been entered with the aid of the editor. The SKIP function simply steps through the entered program one com-

plete instruction at a time whenever the SKIP key is depressed. The entire instruction, be it one, two or three bytes long, will appear on the display. Once more, the detailed flowchart of the SKIP subroutine is shown in figure 5.

The two initial instructions (CMP #12 and BNE) determine whether or not the SKIP key was depressed. If so, the only thing to be altered is the position of the current address pointer, CURAD. We already know which subroutine moves the CURAD pointer down by one, two or three bytes: the NEXT subroutine (figure 10).

During this subroutine the processor checks to see whether the CURAD pointer has already exceeded the current end address pointer, CEND. If so, an error will have to be reported. The program will branch to label ERRA via the BPL instruction. The display will show the 'message' EEEEEEE until the SKIP key has been released. Otherwise the processor will branch back to the central CMND label via the BMI instruction and the processor will wait for the next command key to be depressed.

### The DELETE routine

The final command key routine of figure 5 to be described is the DELETE routine. As we know from chapter 5, its purpose is to erase a particular instruction from the computer memory. The DELETE function has to effectively move the complete data block starting at the instruction immediately following the one currently on display and ending at the pointer CEND up by one, two or three bytes. The actual number of bytes the block is moved up depends on the length of the instruction to be erased (the one currently on display). As a result of the data block being moved up, the current instruction will be overwritten.

As can be seen from the detailed flowchart in figure 5, the DELETE key is detected in the usual manner (with the aid of the instructions CMP #11 and BNE). Once the DELETE key has been depressed, the processor jumps to the subroutine UP. As expected, this is the subroutine which moves the data block up by the required number of bytes. (This subroutine will be described in detail later).

Since deleting an instruction will cause the file to be shortened by one, two or three memory locations, the current end address pointer CEND will also have to be shifted upwards. This is accomplished by the subroutine RECEND (= Reduce Current END address = move CEND pointer up by value of BYTES). The detailed flowchart of the RECEND subroutine is shown in figure 14. It simply performs a 16 bit subtraction. In fact, it is the exact opposite of the subroutine ADCEND shown in figure 13 (the 16 bit addition). The operation of the RECEND subroutine should hardly need explaining by now. All that happens is that the contents of address location BYTES are subtracted from the contents of the current end address pointer CEND. When the processor returns from the RECEND routine a jump is made back to the central label CMND where the computer once again waits for a new command key to be depressed.

This completes the description of all five command routines contained in the editor program. However, three subroutines were only mentioned



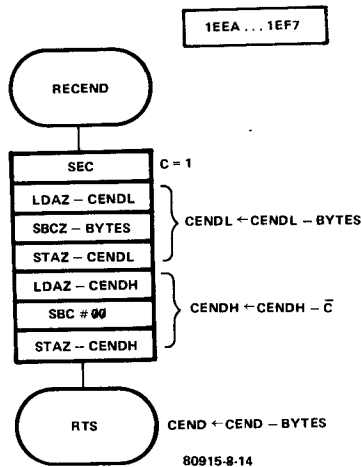


Figure 14. The subroutine RESEND is the opposite of ADCEND (figure 13). Its function is to decrease the value of CEND by the contents of location BYTES — move the CEND pointer up by the number of bytes corresponding to the length of the instruction to be erased.

briefly and it is time to examine these in greater detail. The three subroutines in question are: DOWN, UP and OPLEN/LENACC.

### The subroutine DOWN

As mentioned previously, the DOWN subroutine is called by the FILLWS subroutine. The latter was used for the INSERT and INPUT functions, when space for the new instructions had to be made in the file. In other words, the processor had to move a data block down by one, two or three bytes depending on the length of the instruction to be entered. Essentially, this is the task of the DOWN subroutine. The detailed flowchart of the DOWN subroutine is shown in figure 15, while the operation is illustrated in figure 16.

During the DOWN subroutine the processor makes use of a new address pointer MOVAD (= MOVE Address). This pointer will always indicate a byte in the file that is to be moved down by one, two or three memory locations. The first four instructions in the DOWN subroutine therefore make the contents of MOVAD equal to those of CEND. The move address pointer will indicate the same address location as the CEND pointer, which is the memory location following the EOF character 77.

By now the processor has reached the label DNLOOP. This is where the actual moving of the data block takes place. Firstly, the value contained in the Y index register is made equal to zero. The contents of the address location indicated by the MOVAD pointer are then loaded into the accumulator (Y = 00). The contents of location BYTES are then transferred to the Y index register. The value in the accumulator is then stored

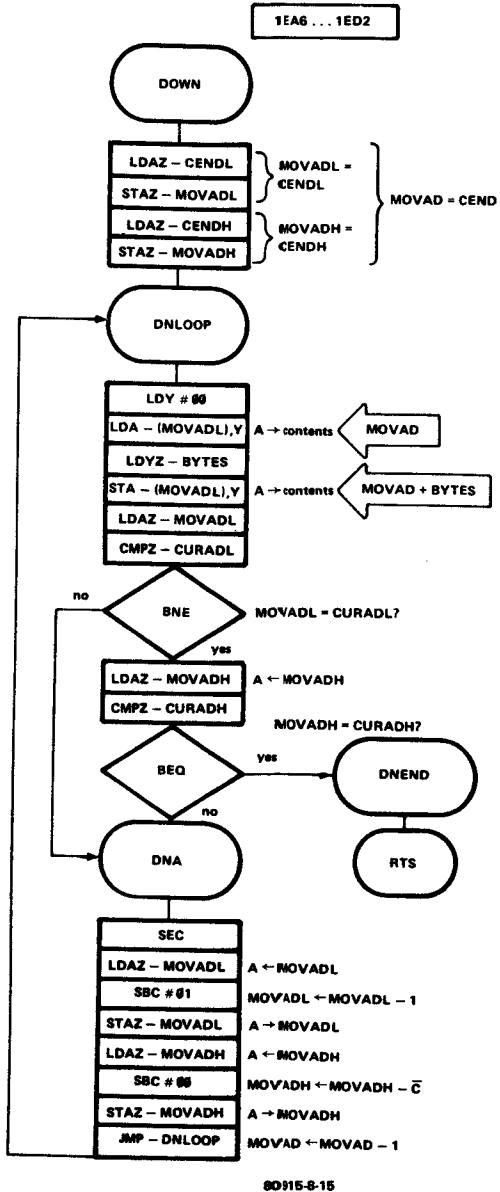
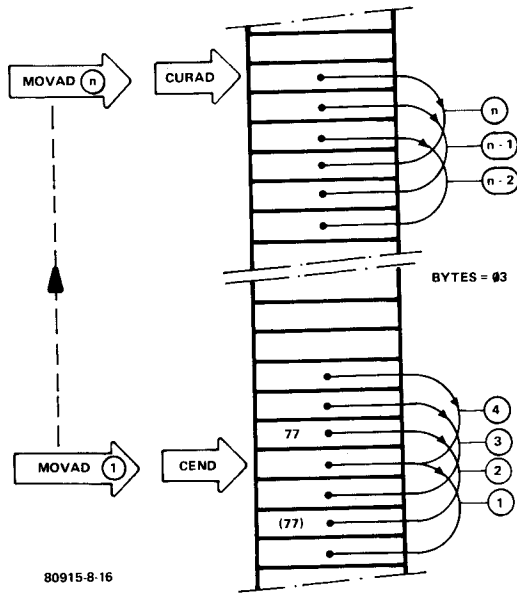


Figure 15. The DOWN subroutine shifts a data block in the file down by one, two or three memory locations. This makes room for a new instruction to be copied from the display buffer.



80915-8-16

**Figure 16. The operation of the DOWN subroutine. In the given example the data block is moved down three places as the contents of memory location BYTES equals 03.**

in the address location indicated by the MOVAD pointer plus location BYTES ( $Y = \text{BYTES}$ ).

Subsequently, a comparison is made between the two pointers CURAD and MOVAD. If the pointer MOVAD is not equal in value to that of CURAD, there are more bytes to be moved down in the file. The processor will then decrement the value of the pointer MOVAD after the label DNA, meaning that MOVAD will move up one memory location in the file. After jumping back to the label DNLOOP the processor will also move the data byte pointed to by MOVAD down by the number of locations indicated by the value contained in location BYTES. Thus, the procedure described above is repeated until the two pointers MOVAD and CURAD are equal. Once this is so, all the bytes in the data block will have been moved down in the file by the requisite number of memory locations. The processor is then able to insert the new instruction in the (now) vacant space immediately following the address indicated by the pointer CURAD.

A graphical representation of the way in which the DOWN subroutine operates is given in figure 16. The MOVAD pointer will always indicate the data byte to be moved down in the file. After every 'shift' operation the processor will decrement the pointer MOVAD, so that it gradually moves up in the file. This means that its starting value will be equal to CEND and its final value will be equal to CURAD. In figure 16 the contents of

location BYTES are equal to 03. Therefore, three consecutive memory locations will be vacated (following CURAD) in which the new instruction can be entered.

In the standard version of the Junior Computer a file will be no longer than 512 bytes. Such short files will only require the use of the DOWN subroutine for very brief periods, since few data bytes are to be shifted. When the computer is expanded, however, a file could well be two to four kilobytes long. If an instruction is to be entered near the beginning of such a long file via the INSERT or INPUT routines, the Junior Computer will spend several seconds in the DOWN subroutine. This can be seen on the display which will go dark for some time!

### The subroutine UP

The UP subroutine is the reverse of the DOWN subroutine and, as we know, is called during the DELETE function. The latter is used to erase certain instructions from the file. This means that the data block in the file will have to be shifted up by one, two or three bytes, depending on the length of the instruction to be deleted, to compensate for the erasure. This is, of course, the task of the UP subroutine. The detailed flowchart of the UP subroutine is given in figure 17, while the operation is illustrated in figure 18.

The address pointer MOVAD is again made use of during the UP subroutine. Again, this pointer will indicate the byte in the file that is to be repositioned. The first four instructions in the UP subroutine therefore make the contents of MOVAD equal to those of CURAD. Whereas the DOWN subroutine started the transfer from the bottom of the file, the UP subroutine starts at the top.

By now, the processor will have arrived at the label UPLOOP. This is where the actual movement of the data block takes place. Firstly, the contents of location BYTES are loaded into the Y index register. The contents of the address location pointed to by MOVAD + BYTES is then loaded into the accumulator. The value in the Y index register is then made zero and the value in the accumulator stored in the address location indicated by MOVAD. The value of the move address pointer is then incremented so that it is ready for the next data byte location. If the low order byte of the MOVAD pointer is zero, the high order byte will also have to be incremented.

This brings us to label UPA. From here on the processor simply determines whether or not the contents of the move address pointer have exceeded the value of the current end address pointer. If not, the processor will branch back to label UPLOOP and move the next data byte in the file up by the number of spaces corresponding to the value held in location BYTES until such time as the two pointers (MOVAD and CEND) are equal.

Once this is so, all the bytes in data block will have been moved up in the file by the requisite number of memory locations. The processor will then have overwritten the instruction previously on display thereby deleting it from the file.

Little needs to be said about the graphical representation of the UP subroutine (see figure 18) as it is virtually identical to that of the DOWN

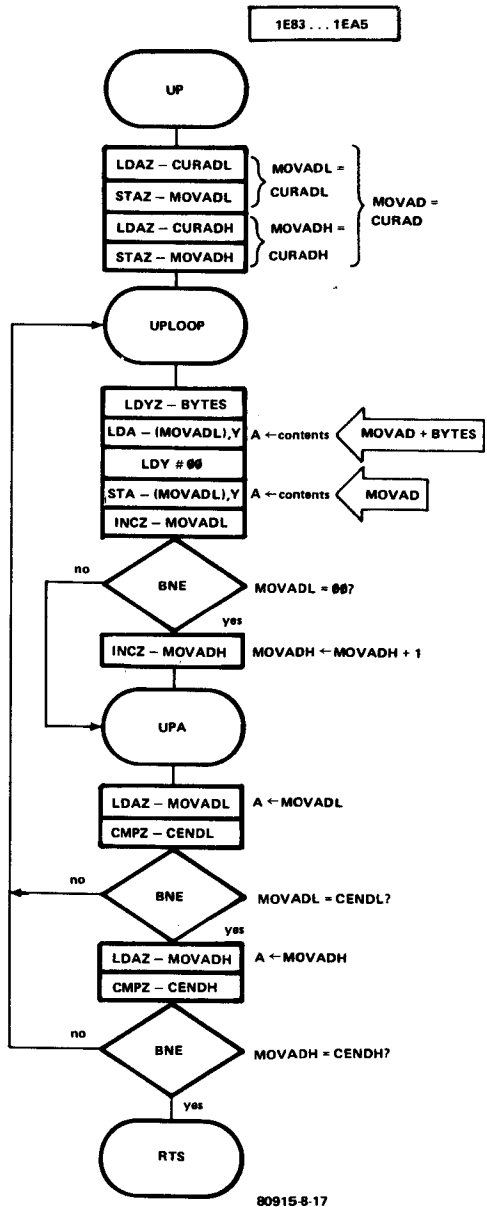
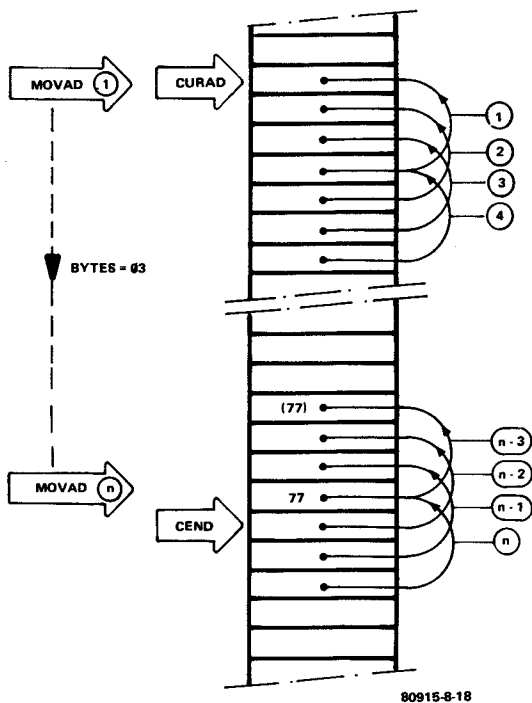


Figure 17. The UP subroutine is the reverse of the DOWN subroutine in that it shifts a data block in the file UP by one, two or three memory locations. This enables unwanted instructions to be erased from the file.

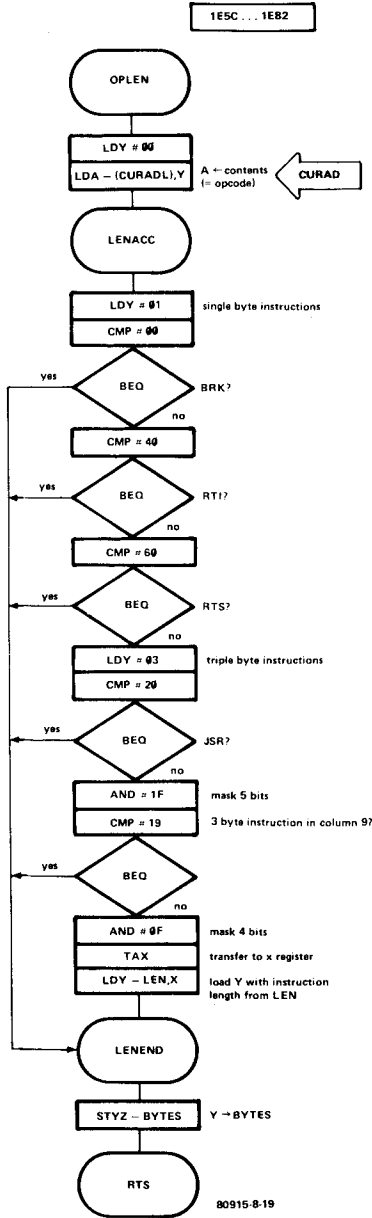


**Figure 18.** The operation of the UP subroutine. In the given example the data block is moved up three places as the contents of location BYTES equals 03.

subroutine (figure 16) – albeit in reverse. Again, the contents of location BYTES are 03. Initially, the MOVAD pointer and the CURAD pointer are equal. The MOVAD pointer always indicates the address of the data byte to be moved up three places in the file. After every shift operation the processor will increment the contents of the MOVAD pointer, so that it gradually moves down the file. This means that its starting value will be equal to CURAD and its final value will be equal to CEND.

### The subroutine OPLEN/LENACC

Part of the OPLEN subroutine should be familiar to us from Book I, chapter 4: the subroutine LENACC. During this subroutine the processor calculates the length of an instruction. Before LENACC can be called the instruction opcode must be held in the accumulator, for without the opcode the length of the instruction can not be calculated. When the subroutine OPLEN is called, therefore, the instruction indicated by the address pointer CURAD is (automatically) entered into the accumulator.



LEN		
1F1F	02	Y = 0
.	02	Y = 1
.	02	Y = 2
.	01	Y = 3
.	02	Y = 4
.	02	Y = 5
.	02	Y = 6
.	01	Y = 7
.	01	Y = 8
.	02	Y = 9
.	01	Y = A
.	01	Y = B
.	03	Y = C
.	03	Y = D
.	03	Y = E
1F2E	03	Y = F

**Figure 19. The OPLEN subroutine calculates the length of a particular instruction, the opcode of which is indicated by the current address pointer. The actual calculation starts after the label LENACC.**

The detailed flowchart of the OPLEN subroutine is given in figure 19. There is a slight difference between the LENACC section of the subroutine described in Book I, chapter 4, page 132, and the LENACC subroutine as contained in the EPROM of the Junior Computer. The two index registers (X and Y) have been interchanged — this has no effect on the operation of the routine.

As you will remember from Book I, the lengths of the various 6502 micro-processor instructions can be determined from a look-up table. This table was previously called LENTBL, but is now called, quite simply, LEN (see table 1).

Table 1	LEN			
1F1F	02	X = 00;	column 0	(mainly 2 byte instructions)
1F20	02	X = 01;	column 1	(only 2 byte instructions)
1F21	02	X = 02;	column 2	(LDX IMM)
1F22	01	X = 03;	column 3	(empty '1 byte instruction')
1F23	02	X = 04;	column 4	(mainly 2 byte instructions)
1F24	02	X = 05;	column 5	(only 2 byte instructions)
1F25	02	X = 06;	column 6	(only 2 byte instructions)
1F26	01	X = 07;	column 7	(EOF character '1 byte instruction')
1F27	01	X = 08;	column 8	(only 1 byte instructions)
1F28	02	X = 09;	column 9	(mainly 2 byte instructions)
1F29	01	X = 0A;	column A	(mainly 1 byte instructions)
1F2A	01	X = 0B;	column B	(empty '1 byte instructions')
1F2B	03	X = 0C;	column C	(mainly 3 byte instructions)
1F2C	03	X = 0D;	column D	(only 3 byte instructions)
1F2D	03	X = 0E;	column E	(mainly 3 byte instructions)
1F2E	03	X = 0F;	column F	(label FF '3 byte instruction')

The look-up table, LEN, was compiled with the aid of the condensed instruction code table shown in figure 20 (this can also be found on pages 130 and 131 of Book I). Note that columns 7 and F are no longer empty. They contain the EOF character 77 and the label identifier FF respectively. Full details of this table and the LENACC subroutine were given in Book I, therefore there is no need to describe them fully here. However, since the subroutine OPLEN/LENACC is used quite often, a summary of the main points would not be amiss:

1. After the subroutine OPLEN is called, the accumulator is loaded with the opcode of the instruction which is being pointed to by the current address pointer CURAD. This pointer will indicate a normal instruction opcode, the label identifier (pseudo-opcode) FF or the EOF character 77.
2. From LENACC on, the Y index register is loaded with 01 and the instructions BRK, RTI and RTS (single byte instructions) are filtered out with the relative compare instructions. If the opcode of one of these three instructions is held in the accumulator, the contents of the Y index register (01) will be stored in location BYTES (= instruction length).
3. The next instruction to be filtered out is the JSR instruction. As this instruction is three bytes long, the Y index register is first loaded with 03. The following compare and branch instructions direct the processor to store the value 03 (Y register) in location BYTES if the opcode held in the accumulator is that of the JSR instruction.



		least significant four bits							
		0	1	2	3	4	5	6	7
most significant four bits	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)	STX Z (2)	
	9	BCC (2)	STA (IND),Y (2)				STY Z,X (2)	STX Z,X (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)

		least significant four bits							
		8	9	A	B	C	D	E	F
most significant four bits	0	PHP (1)	ORA # (2)	ASL A (1)			ORA ABS (3)	ASL ABS (3)	
	1	CLC (1)	ORA ABS,Y (3)				ORA ABS,X (3)	ASL ABS,X (3)	
	2	PLP (1)	AND # (2)	ROL A (1)		BIT ABS (3)	AND ABS (3)	ROL ABS (3)	
	3	SEC (1)	AND ABS,Y (3)				AND ABS,X (3)	ROL ABS,X (3)	
	4	PHA (1)	EOR # (2)	LSR A (1)		JMP ABS (3)	EOR ABS (3)	LSR ABS (3)	
	5	CLI (1)	EOR ABS,Y (3)				EOR ABS,X (3)	LSR ABS,X (3)	
	6	PLA (1)	ADC # (2)	ROR A (1)		JMP IND (3)	ADC ABS (3)	ROR ABS (3)	
	7	SEI (1)	ADC ABS,Y (3)				ADC ABS,X (3)	ROR ABS,X (3)	
	8	DEY (1)		TXA (1)			STY ABS (3)	STA ABS (3)	STX ABS (3)
	9	TYA (1)	STA ABS,Y (3)	TXS (1)				STA ABS,X (3)	
	A	TAY (1)	LDA # (2)	TAX (1)			LDY ABS (3)	LDA ABS (3)	LDX ABS (3)
	B	CLV (1)	LDA ABS,Y (3)	TSX (1)			LDY ABS,X (3)	LDA ABS,X (3)	LDX ABS,Y (3)
	C	INY (1)	CMP # (2)	DEX (1)			CPY ABS (3)	CMP ABS (3)	DEC ABS (3)
	D	CLD (1)	CMP ABS,Y (3)					CMP ABS,X (3)	DEC ABS,X (3)
	E	INX (1)	SBC # (2)	NOP (1)			CPX ABS (3)	SBC ABS (3)	INC ABS (3)
	F	SED (1)	SBC ABS,Y (3)					SBC ABS,X (3)	INC ABS,X (3)

80915-20

**Figure 20. Both the EOF character 77 and the label identifier FF have been included in the condensed instruction table. This table is used to compile the look-up table LEN (table 1).**

4. The next step in the OPLEN subroutine is to filter out all three byte instructions of column 9 in figure 20. This is accomplished by masking the five least significant bits (AND #1F) and comparing the result with the value 19. If the opcode contained in the accumulator belongs to one of these instructions the value 03 (Y index register still contains 03) will be placed in location BYTES.

5. Finally, if none of the above points apply, the length of the instruction must be obtained from the look-up table LEN. This is accomplished by masking the four least significant bits of the opcode (AND #0F) and transferring the result into the X index register. The Y index register is then loaded with the value obtained from the table, which is then transferred to location BYTES.

We should now be totally familiar with the operation of the editor section of the monitor program. The various editor subroutines will prove to be a great help to the programmer when developing future programs. The complete source listing pertaining to the editor can be found in the appendix to this book.

# The Assembler program

## Tidying up the edited program

Once a program has been entered into the Junior Computer by means of the editor routine, it has to be 'reshaped' into a form that the 6502 microprocessor can understand. This is the task of all the subroutines used by the assembler section of the monitor program. When a program is entered with the aid of the editor, the operand of a jump or branch instruction will usually be a label number, in other words, a symbolic address. Even the labels themselves are pseudo-instructions which the processor is unable to understand. For these reasons, the assembler must remove all labels from the program, replace the symbolic addresses of jump instructions with their 'real' addresses and replace the label numbers of branch instructions with their actual displacement values. This chapter therefore describes the way in which all this is dealt with by the various subroutines used during the assembler program.

Chapter 5 introduced us to the benefits to be derived from using the editor and the assembler. When entering a program with the aid of the editor it is no longer necessary to know the start addresses of the subroutines to be called or the displacement values of the branch instructions. For now we can use labels, or so-called symbolic instructions. By using labels instead of actual addresses, the computer can be made to relieve us of a considerable amount of tedious calculations (what else are computers for?). Programs can be keyed in to the memory banks of the Junior Computer with the aid of the editor and without any difficulty. Once the program has been assembled we can be sure that all the subroutine start addresses and displacement values of branch instructions are correct. This means that the time consuming job of stepping through programs to look for errors can be reduced to a swift, last minute

check. Another advantage is that the editor and assembler enable programs to be stored in the available memory as compactly as possible.

## The basic flowchart of the assembler

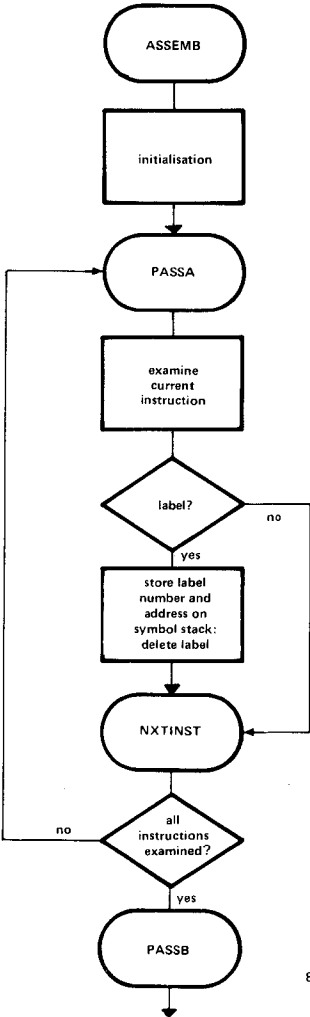
It was mentioned in chapter 5 that the assembler contained in the monitor program of the Junior Computer is a 'two pass' assembler. This means that the actual assembly procedure is carried out in two stages. The basic flowchart of the assembler is given in figures 1 and 2. In order to appreciate the two phases involved, the basic flowchart has been divided into two sections.

As the obvious place to start is at the beginning, let us first take a look at figure 1. The label ASSEMB (start address 1F51) heads a section of program which carries out all the necessary preparations before the actual assembler main routine can proceed. This involves setting the various address pointers to the correct positions.

Next we come to label PASSA which is where the first phase of the main assembler program starts. Here, the processor will only concentrate on labels entered by the programmer with the aid of the editor, all other instructions will be ignored. How does the processor track down each of the labels? This is a relatively simple task. As we already know, the processor is able to determine the length of any given instruction with the aid of the OPLEN subroutine. We also know that the NEXT subroutine can be used to 'skip' from instruction to instruction. By combining these two subroutines the processor simply runs through the entire program (section) and examines the first byte of each instruction encountered.

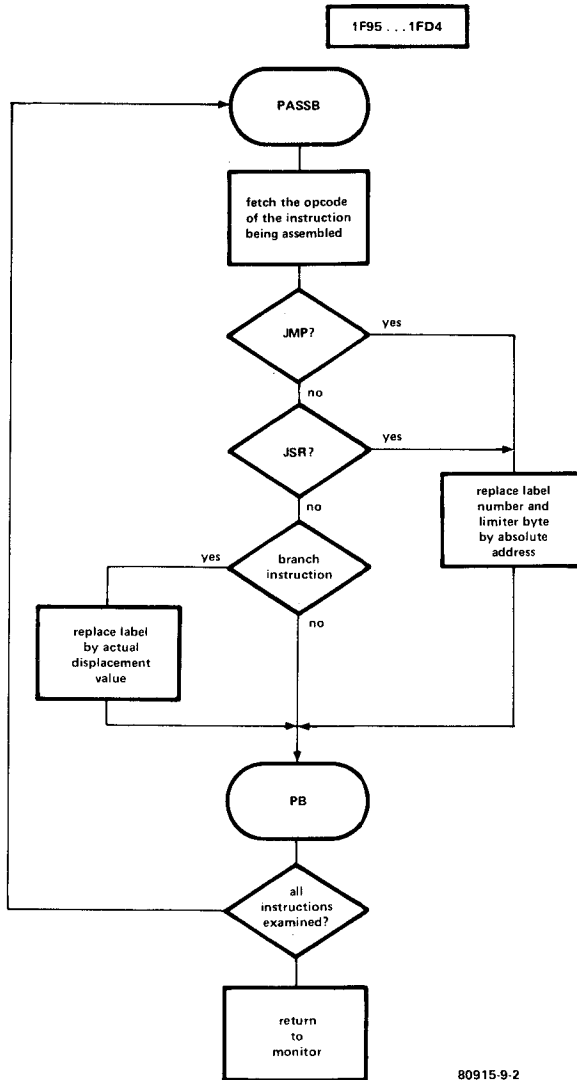
If a label is involved, the value of this first byte will be FF (the label identifier). The processor will then take note of the address which contains this pseudo-instruction. In addition, it will store the label number following the identifier. Both the address and the label number are stored in what is called the 'symbol stack'. This is virtually nothing more than a scratch pad and can be compared to the computer's normal stack situated on page 1. The only real difference between them is that the stack pointer is controlled by the microprocessor itself (hardware controlled) and the pointer belonging to the symbol stack is controlled by the program (software controlled). Nevertheless, they both have one thing in common and that is that they 'expand' upwards from the bottom.

Now to get back to the first phase of the assembly process. Once the processor has found a label in the file and the address of the pseudo-instruction together with the label number have been stored on the symbol stack, the label must be deleted from the file. The subroutine which takes care of this was introduced in the previous chapter, namely the UP subroutine which was called during the DELETE function. Since a label is always three bytes long, the data block starting immediately after the label and ending at the address indicated by the CEND pointer will be moved up by three bytes. This deletes the label from the file which is now three bytes shorter. Obviously, the end address pointer, CEND, will now have to be readjusted so that it also moves up three places in the file. Now the opcode of the instruction following the deleted label will have moved



80915-9-1

**Figure 1. The basic flowchart of the first phase of the assembler. During this phase, the processor removes all the labels from the file and stores the label number and its associated address on the symbol stack.**



80915-9-2

**Figure 2. The basic flowchart of the second phase of the assembler. During this phase, the label number and limiter byte following jump instruction opcodes are replaced by the actual jump address. At the same time, the label number following a branch instruction is replaced by the actual displacement value.**

to the address where the label identifier was situated previously. The above procedure is clearly illustrated in figures 3a and 3b.

Once the label has been erased from the file and all the relevant information concerning it has been stored on the symbol stack, the processor will search for the next label. This means that the processor continues to jump from instruction to instruction while testing each one to see if the first byte has the value FF. Whenever the label identifier, FF, is encountered, the same procedure takes place:

1. The address location where the label identifier is found is stored on the symbol stack (first the high order byte followed by the low order byte).
2. The label number following the identifier is obtained and is also stored on the symbol stack. This means that three important parameters concerning the label are stored on the symbol stack: the high and low order bytes of the address where it was found and the label number. The exact structure of the symbol stack will be discussed in detail later.
3. The label is removed from the file. The instruction immediately following the label will now be located at the address where the label was previously.
4. The file is examined for more labels and if none are found the process will be stopped.

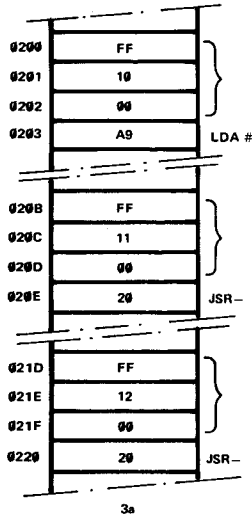
This procedure is also illustrated in figure 3. Four stages of a label search operation during the first phase of assembly are shown. In figure 3a the processor encounters a label at address 0200. Once all the required details concerning the label have been stored on the symbol stack, the label can be erased from the file. This situation is shown in figure 3b. As a label is always three bytes long, the data block following it has been shifted up three places. Thus the opcode of the instruction following the label will now be situated at the old address of the label. At the same time, the file will have become three bytes shorter.

The processor then continues the search for another label, which it finds at address location 0208. The required parameters are again stored on the symbol stack and the label is removed from the file (figure 3c). The next label will be found at address 0217 and the above procedure is repeated to produce the final result as shown in figure 3d.

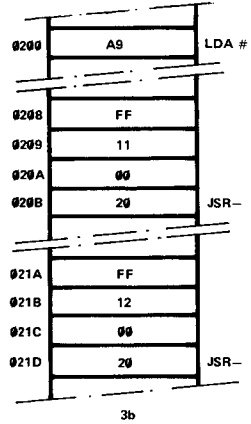
This covers just about all there is to say initially about the first phase of the assembly routine. Now it is time to take a look at the second phase (PASSB). As stated previously, the basic flowchart of the second phase of the assembler operation is shown in figure 2. By this time all the labels will have been removed from the file. After the label PASSB the processor will examine all the instructions remaining in the file, starting with the first one. However, the only instructions the processor is interested in are those referring to a label number. We know which instructions these are from chapter 5:

- \* the JMP instruction with the opcode 4c
- \* the JSR instruction with the opcode 20
- \* the branch instructions BCC, BCS, BEQ, BMI, BNE, BPL, BVC and BVS.

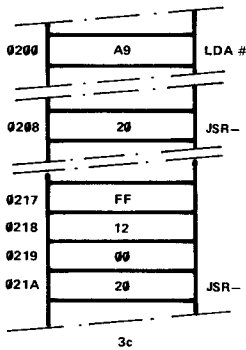
During the second assembly phase, the processor assigns the correct addresses and displacement values to the above instructions. For this reason the three different types of instruction are filtered out of the file



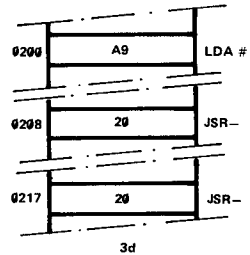
(a)



(b)



(c)



(d)

80915-9-3

**Figure 3. Four stages of a label search operation during the first phase of the assembler. Each time a label is encountered in the file the following data block is shifted up three locations, after the label number and its address have been stored on the symbol stack.**

one after the other. The processor is quite capable of determining the type of instruction from its opcode. If the processor encounters the opcode 4C (the JMP instruction) it 'knows' that there is a label number following. The address of the label was stored on the symbol stack together with the label number during the first phase of assembly (PASSA).

The computer will now interrogate the symbol stack until it finds a label number identical to the one following the opcode 4C. When this has been found, the processor will fetch the corresponding address from the symbol stack and place it behind the opcode 4C. The address consists of two bytes, therefore the label number and the limiter byte which were previously behind the opcode will now be replaced by the actual jump address in the process. The JMP instruction (with the opcode 4C) has now been assembled.

The next instruction the computer checks for is the JSR instruction. If the computer comes across the opcode 20 the symbol stack is once more interrogated for the address corresponding to the label number following the JSR instruction. As before, the processor places this address behind the instruction and thereby assembles it. The procedure is identical to that for the JMP instruction.

As we know, the 6502 microprocessor features eight branch instructions. If the processor encounters the opcode of a branch instruction, the label number following it must be replaced by the actual displacement value. A branch instruction is assembled as follows:

- \* The processor checks to see which label number follows the opcode of the branch instruction. An absolute address corresponding to that label is stored in the symbol stack.
- \* The processor searches for the label number in the symbol stack. Once this has been found, the corresponding absolute address will also be known.
- \* The computer will now know the address containing the opcode of the branch instruction and the absolute address of the label number. From this the processor can calculate the required displacement value for the branch instruction.
- \* The processor places the displacement value it has calculated immediately behind the branch instruction to be assembled.

In certain instances the processor may come across a JMP, JSR or branch instruction that does not require assembly. As mentioned in chapter 5, there are certain times when the jump instruction refers to a particular address that is already known during editing. If this is the case, the instruction will not be followed by the label number and the limiter byte 00. Also, the label will not have been stored in the symbol stack during the first phase of assembly as the label identifier FF will be absent. In other words, the label is non-existent in the file and so the processor was unable to store the label number and its corresponding absolute address on the symbol stack during PASSA. If the Junior Computer encounters such an instruction during the second phase of the assembly procedure, the instruction will not be assembled. This means that no data is altered after the opcode. For the sake of clarity, the assembly block has been left out of figure 2.

Up to now, we have discussed the general procedure during assembly. We



have discovered that the assembler processes the file produced by the editor in two stages. During the first phase the processor removes all the labels from the file and stores all the label numbers and associated addresses on the symbol stack. During PASSA the processor is only interested in those instructions that start with the pseudo-opcode FF. Once all the labels have been deleted and all the label numbers and their absolute addresses have been stored on the symbol stack, the computer will start the second phase of the assembly operation (PASSB).

During this second phase the processor is only interested in the opcodes 4C, 20 and those belonging to the various branch instructions. These are still followed by label numbers (and in the case of the jump instructions by the limiter bytes). The processor will then track down the label number situated behind the opcode of the instruction being assembled in the symbol stack. Each label number on the symbol stack has a corresponding absolute address, which will be placed in the file directly after the opcode as far as jump instructions are concerned. With respect to branch instructions, the processor will calculate the displacement value from this address. The organisation of the symbol stack still remains to be considered. Now that we know the basics of the assembly procedure, we can go in to it in greater detail and discover more about the symbol stack in the process.

### **The detailed flowchart of the assembler**

The detailed flowchart of the assembler routine will again be discussed in two phases. The first phase of the assembler (PASSA) is shown in figure 4. This is the section that deletes the labels from the file and stores their numbers and associated absolute addresses on the symbol stack.

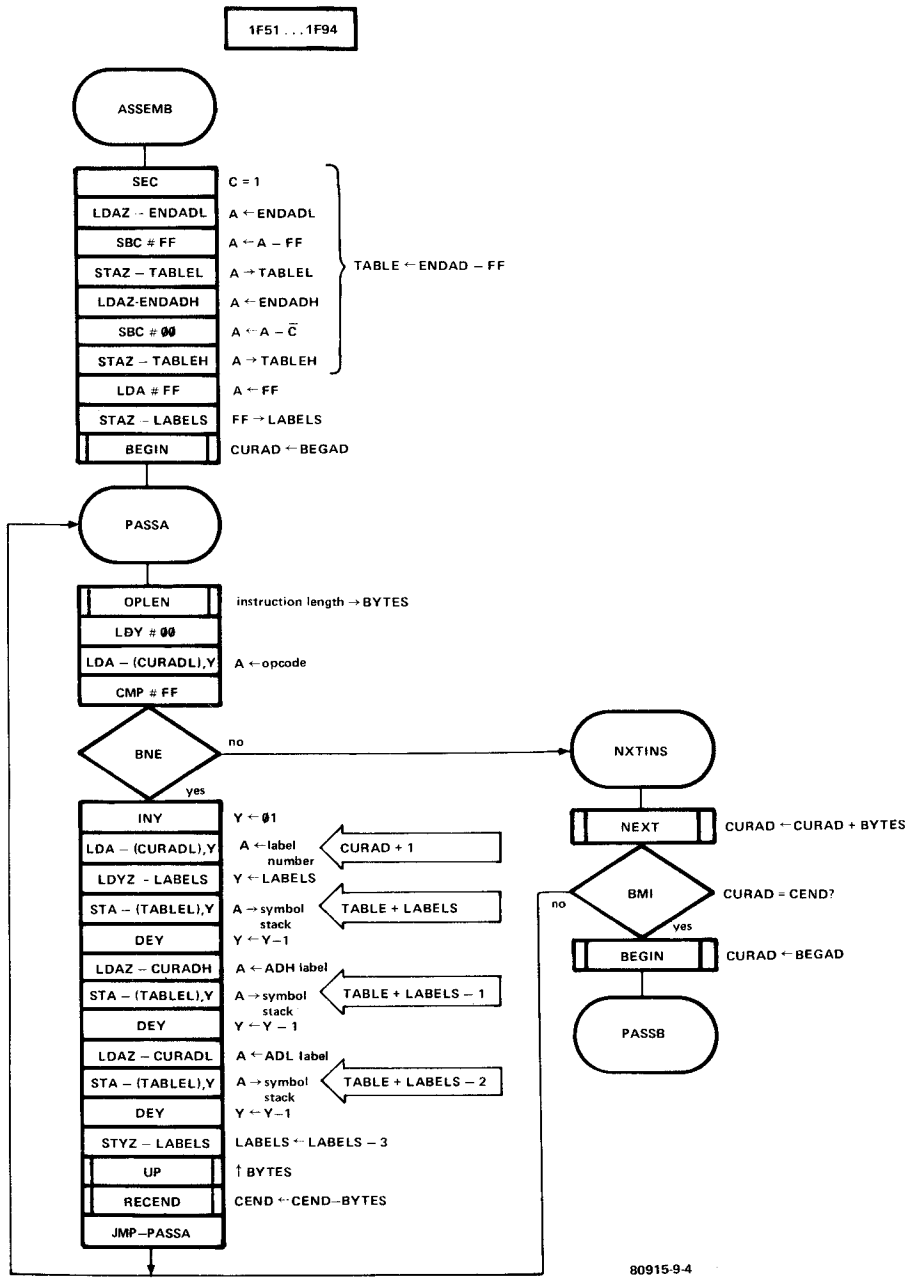
The assembler starts from the label ASSEMB at address location 1F51. During the first part of this program section the processor sets various address pointers and initialises the symbol stack.

Before continuing with this description, a few general remarks should be made. The assembler program is closely related to the editor program. In other words, the assembler makes use of many of the subroutines used by the editor. For this reason, it would be useful to study chapter 8 again at this stage, if the way in which the editor subroutines operate is not fully understood.

The assembler uses an address pointer that has not yet been mentioned, namely: TABLE. This is stored in address locations 00ED and 00EC (TABLEH and TABLEL respectively).

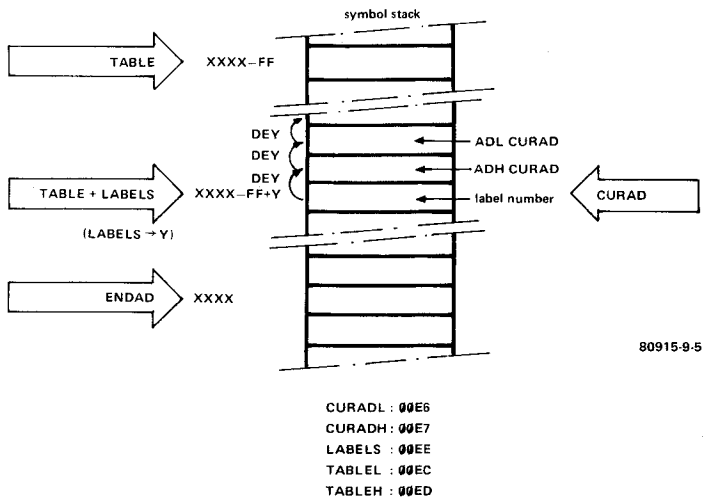
These two memory locations contain the address indicated by the TABLE pointer. This address pointer always indicates the highest address on the symbol stack. The lowest address will be TABLE + FF, meaning that the symbol stack is 256 bytes long. The first seven instructions following the label ASSEMB ensure that the address pointer TABLE is initially loaded with an address which is 256 (= FF) memory locations in front of the ENDAD (end address) pointer. This is also illustrated graphically in figure 5.

Next, the assembler routine uses another new memory location called LABELS. This is situated at address 00EE. When the contents of LABELS



80915-9.4

Figure 4. The detailed flowchart of the first phase of assembly.



80915-9-5

**Figure 5. The pointer  $TABLE + LABELS$  enables the processor to control the data transfer to or from the symbol stack. The symbol stack is 'filled' from bottom to top during the assembly process. First the label numbers and then the contents of the current address pointer  $CURAD$  are stored on the stack.**

are added to those of the address pointer  $TABLE$ , the result will be the actual address to which the symbol stack pointer is directed at that moment. This will therefore be equal to  $TABLE + LABELS$ . During the first phase of assembly the symbol stack pointer will always indicate the first available vacant location in the symbol stack. If a label is found during this first phase of assembly, the label number will be stored in the first spare location of the symbol stack followed by the low order address byte in the next spare location followed by the high order address byte in the third spare location.

By now it should be fairly clear as to how the symbol stack is organised. Once all the pointers have been initialised, the symbol stack pointer will indicate the same address location as the  $ENDAD$  pointer (see figure 5 for details). Immediately before the label  $PASSA$  the subroutine  $BEGIN$  is called. This is a short subroutine which has already been described in chapter 8, figure 7. It simply ensures that the contents of the pointers  $CURAD$  and  $BEGAD$  are the same. As you know, the  $BEGAD$  pointer indicates the start address of the file.

## PASSA

Now the processor has reached the label  $PASSA$ , thereby initiating the first phase of the actual assembly process. To start with, the processor jumps to the subroutine  $OPLEN$  (see chapter 8, figure 19). This subroutine calculates the length of an instruction and stores the result in memory location  $BYTES$ . This informs the processor how many places the current

address pointer (CURAD) should be shifted for the next instruction to be examined.

The Y index register is then loaded with 00 so that it can be used as an index to load the accumulator with the opcode of the instruction indicated by the current address pointer CURAD. The two subsequent instructions (CMP #FF and BNE) test to see whether this instruction is actually the pseudo-opcode of the label (the label identifier FF). If not, the processor will branch to the section of program headed by the label NXTINS (= NeXT INStRuction) as shown in figure 4.

During the NEXT subroutine (chapter 8, figure 10), the processor directs the current address pointer to the following instruction by adding the contents of location BYTES to the previous contents of CURAD. This subroutine also tests to see whether the current address pointer, which should be still pointing to the opcode of an instruction, is still situated between the pointers BEGAD and CEND, or whether it has already exceeded the CEND pointer. If it is still within the file boundaries, the processor will branch back to label PASSA and go on to examine the next instruction for the label identifier FF. The length of this instruction will again be calculated during the OPLEN subroutine. The above procedure is repeated until all the labels have been removed from the file.

If, however, the processor encounters the label identifier FF, it will not branch to label NXTINS, but will deal with the label in question. It is now time to find out exactly what this entails. To do this we need to look at figures 4 and 5. Figure 4 shows the details of the first assembly phase while figure 5 shows part of the symbol stack. Following the (non) branch instruction BNE, the process is as follows:

1. The instruction INY increases the contents of the Y index register from 00 to 01. The following instruction (LDA-(CURADL), Y) causes the accumulator to be loaded with the contents of the location indicated by CURAD + 1. This means that the label number following the identifier is now held in the accumulator.

2. The contents of location LABELS are stored in the Y index register. The ensuing instruction (STA-(TABLEL), Y) transfers the number of the label being examined to the symbol stack. The actual address in the stack in which it is stored will be indicated by the temporary pointer TABLE + LABELS. Since the contents of location LABELS were made equal to FF at the beginning of the assembler routine, which was when the symbol stack pointer TABLE was made equal to the contents of ENDAD-FF, the first label number will be stored in the location indicated by the end address pointer ENDAD.

3. DEY

```
LDA-CURADH
STA-(TABLEL), Y)
```

The processor has encountered the label identifier FF. The current address pointer CURAD will still be directed at the address location where this was found. Once the Y index register has been decremented, the high order byte of this address will be loaded into the accumulator. The processor now stores this on the symbol stack at the location indicated by TABLE + LABELS - 1. The high order byte of the label address will now be stored on the symbol stack one location above where the label

number was previously stored.

4. DEY

LDA-CURADL  
STA-(TABLEL),Y)

The processor now loads the low order address byte of the location containing the label into the accumulator in the same manner as before. Next it is stored on the symbol stack at the address  $TABLE + LABELS - 2$ . The low order byte of the label address will therefore be stored one memory location above the high order byte of the label address previously stored.

5. DEY

STY-LABELS

The Y index register has now been decremented three times in succession, or rather, the symbol stack pointer has been shifted up by three places. The pointer  $TABLE + LABELS$  is again pointing to an address in the symbol stack where the next label number is to be stored, provided of course that there is another label with the pseudo-opcode FF present in the file.

6. JSR-UP

JSR-RECORD

All the necessary information regarding the label and its address have now been stored on the symbol stack. Thus, the label is no longer required and can be deleted from the file. It is the subroutine UP (chapter 8, figure 17) which, as you know, performs this task by shifting the remainder of the data block up by three memory locations, starting at the opcode of the next instruction and ending at CEND. After this shift operation, the file will have been shortened by three bytes and so the CEND pointer will also have to be shifted up three locations. This is carried out by the subroutine RECORD (chapter 8, figure 14).

7. JMP-PASSA

At this stage, the Junior Computer examines the remaining instructions in the file to see whether any of them possess the pseudo-opcode FF. Upon encountering further labels in the file, the label number and the address where it was found is stored on the symbol stack. Again, the labels can be erased from the file. If the instruction turns out to be of a different kind, the assembler program (see figure 4) will branch to the section of program labelled NXTINS and skip from instruction to instruction until it finds a label.

During the NEXT subroutine, which repositions the current address pointer to indicate the following instruction opcode, the processor checks whether or not the CURAD pointer has overstepped its mark, by exceeding CEND. If the processor has reached the end of the file, the first phase of assembly will be complete and the processor will proceed to PASSB after the subroutine BEGIN has been executed. This starts the second phase of assembly.

In chapter 5 it was mentioned that at least six memory locations must be kept clear between the pointers ENDAD and CEND. If the programmer is not sure whether his/her program is the right length, he/she can check to see whether there are six free locations between the two pointers (before the assembler is started). This can be accomplished by examining the contents of the various pointers in page zero:

the address of CENDL is 00E8  
the address of CENDH is 00E9  
the address of ENDADL is 00E4  
the address of ENDADH is 00E5

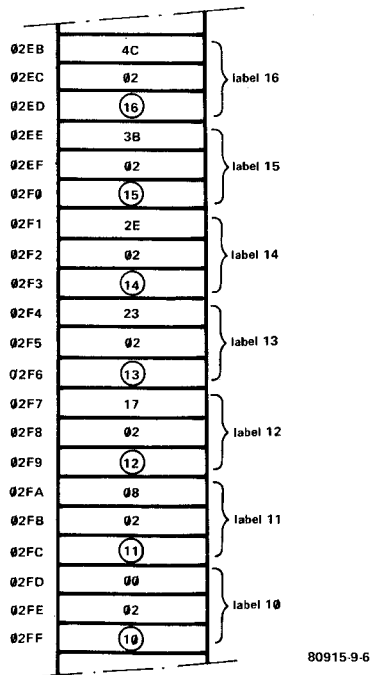
What about the six memory locations that seem so vital? As you may remember, the label number and the absolute address belonging to it are stored on the symbol stack, therefore occupying a total of three locations. The symbol stack starts at the address indicated by ENDAD. When a label is removed from the file, the file becomes three bytes shorter, in other words, the CEND pointer moves up three locations. When another label is removed from the file there appears to be adequate room for it and its address to be stored on the symbol stack in the locations just vacated. Therefore, the question is, why insist on six memory locations, when three seem to be ample? To find out, let us take another look at the UP subroutine in chapter 8, figure 17. This is where the labels are removed from the file. When one is deleted, the data block following it is moved up three bytes. This is also illustrated in figure 18 of chapter 8. When the data block moves up, the UP subroutine also moves up another three bytes behind the pointer CEND. These are superfluous to the assembler, so that if only three spaces were left available between the two pointers CEND and ENDAD, the last data at the end of the file before the CEND pointer would be overwritten by new data.

Part of the symbol stack after the first phase of program assembly using the example from chapter 5 (figures 1 . . . 4) is given in figure 6. All seven label numbers and their associated addresses are stored on the symbol stack. It can be clearly seen that the symbol stack is filled with data from bottom to top, the lowest label being the first to be deleted from the file. The first label number is 10 and it is situated at address 0200. The last label to be assembled has the number 16 and can be found at the absolute address 024C. This was the last label in the file before the EOF character and so it will be stored at the very top of the symbol stack. At this point in time the symbol stack pointer TABLE + LABELS will indicate the address 02EA.

## PASSB

This label heads the second phase of assembly. The detailed flowchart of this section of the assembler program is given in figure 7. To start with, the current address pointer is in the same position as the BEGAD pointer. This was the last operation of the first phase of assembly (see figure 4). During the second phase the processor checks through all the remaining instructions in the file, for initially (during pass one) the computer was only interested in labels featuring the pseudo-opcode FF (the label identifier). Now that these have all been removed from the file, the processor can concentrate on instructions which need to be assembled. These will all have a label number after the opcode. The instructions the processor is now interested in are:

- \* JMP instructions
- \* JSR instructions and
- \* branch instructions.



**Figure 6. Part of the symbol stack after the first phase of program assembly using the example from chapter 5. The end address pointer ENDAD indicates address location 02FF.**

However, these could also be followed by absolute addresses or actual displacement values, in which case they must be ignored. Nevertheless, conditional and unconditional jump instructions have to be dealt with separately during this phase, which we shall describe right away.

Just as in the program section headed by the label PASSA, the section labelled PASSB starts by calling the OPLEN subroutine to ascertain the length of the instruction and therefore the number of memory locations that the CURAD pointer must be shifted down in the file in order to indicate the next instruction. After the Y index register has been loaded with 00, the next instruction (LDA- (CURADL), Y) transfers the opcode of the instruction under examination to the accumulator. The processor then filters out the two jump instructions, JMP and JSR, and then moves on to filter out the various branch instructions:

1. CMP #4C  
BEQ

filters out the JMP instruction which has the opcode 4C.

2. CMP #20  
BEQ

filters out the JSR instruction which has the opcode 20.

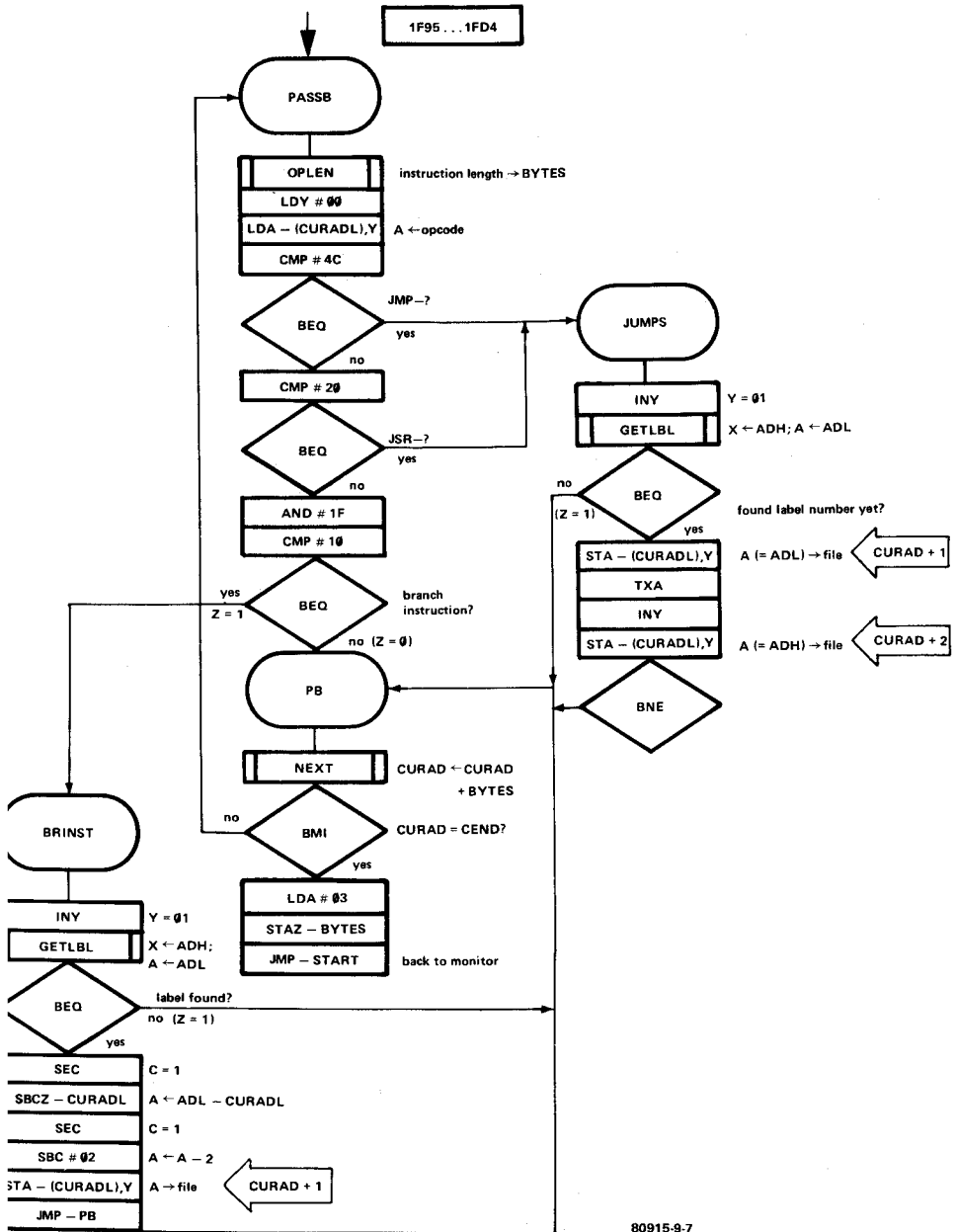


Figure 7. The detailed flowchart of the second phase of the assembly procedure.



3. AND #1F  
CMP #10  
BEQ

filters out all the branch instructions which have the opcodes 10, 30, 50, 70, 90, B0, D0 and F0. These opcodes correspond to the instructions BPL, BMI, BVC, BVS, BCC, BCS, BNE and BEQ respectively. Because the contents of the accumulator (the opcode) are masked by the value 1F, the five least significant bits of the data byte will remain unchanged, but the three most significant bits will be zero. If the opcodes of the branch instructions are considered, it will be apparent that the four least significant bits are always zero. Thus, by masking the five least significant bits of the opcode with 1F the result will invariably be 10. By comparing the result with the value 10, the processor can establish whether the opcode being examined belongs to a branch instruction or not.

If neither a jump instruction nor a branch instruction is involved the opcode will belong to an instruction that does not require assembling. The processor can then deal with the next instruction in the file after it reaches the label PB. As we know, the NEXT subroutine points the current address pointer CURAD to the address containing the opcode of the following instruction.

If the processor has not yet reached the end of the file, the assembler program will branch back to label PASSB to calculate the length of the next instruction via the subroutine OPLEN before dealing with it. The procedure just described above will then be repeated.

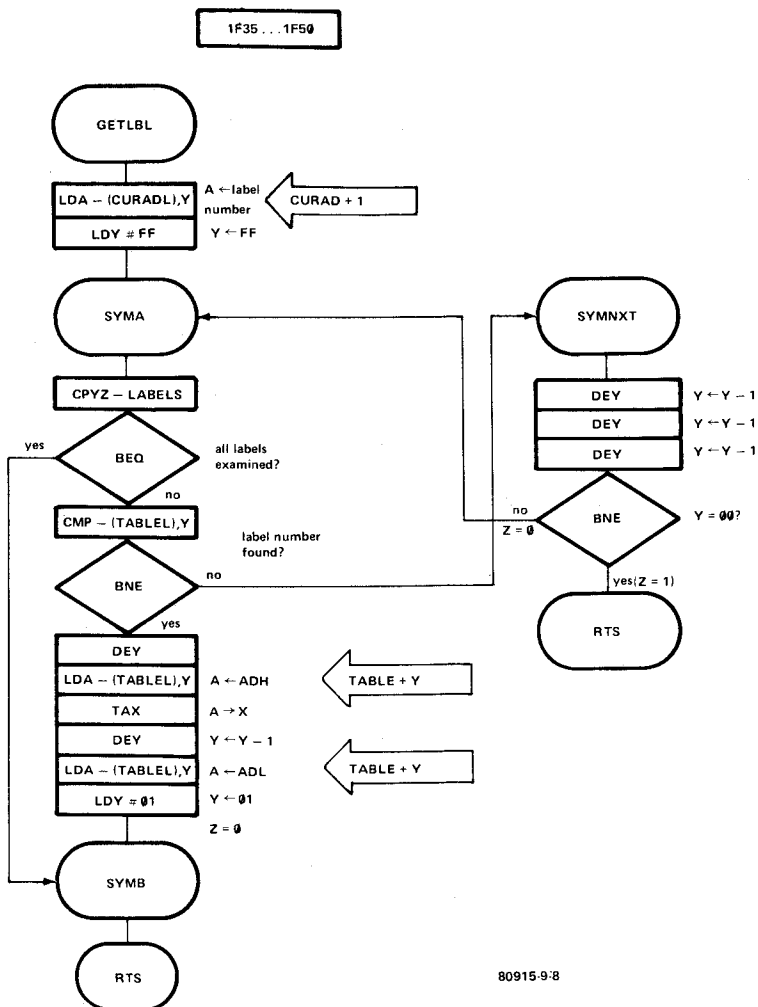
If, however, a jump or branch instruction is involved, it will have to be assembled. For this the Junior Computer will have to place the absolute addresses (previously stored on the symbol stack) behind the jump instructions. With regard to the branch instructions, on the other hand, the displacement values will have to be calculated and then placed behind the opcodes. The way in which the computer carries out these tasks with the aid of the symbol stack can be seen as follows:

### JMP and JSR instructions

All the unconditional jump instructions are assembled from the label JUMPS (figure 7) on. Here the absolute jump address is traced from the symbol stack and then placed behind the opcode of the actual jump instruction.

After incrementing the Y index register (the value therein will now be 01) the subroutine GETLBL is called. This is shown in detail in figure 8 and will be described fully later on. All we have to know about this subroutine at this stage can be expressed in a few words:

1. The subroutine GETLBL searches for the label number behind the opcode of the jump instruction being assembled on the symbol stack. Once this is found on the symbol stack, the processor has access to the absolute address corresponding to it.
2. The address concerned is fetched from the symbol stack. Before returning to the main assembler routine, the subroutine GETLBL stores the two bytes of the label address in two internal CPU registers:



80915-9:8

**Figure 8.** The detailed flowchart of the GETLBL subroutine. This subroutine obtains the label number and associated address from the symbol stack. The high order byte of the label address will be held in the X index register and the low order byte in the accumulator upon the return from this subroutine.

- \* the high order byte of the label address will be stored in the X index register
  - \* the low order byte of the label address will be stored in the accumulator.
3. If the processor can find the required label number on the symbol stack it will return from the GETLBL subroutine with the Z flag reset.

4. If the computer can not find the label in question it will return to the main assembler routine with the Z flag set.

Let us assume that the processor has found the label number on the symbol stack during the GETLBL subroutine. It will then return to the main assembler routine with the high order byte of the label address in the X index register and the low order byte of the label address in the accumulator. The Z flag will have been reset, therefore there will be no branch to the label PB at the next BEQ instruction. The Y index register will still contain the value 01. The following instruction (STA- (CURADL), Y) ensures that the low order byte of the label address is stored immediately behind the opcode of the instruction being assembled. Then the high order byte of the label address is transferred from the X index register into the accumulator and the Y index register is incremented once more. The next instruction (STA- (CURADL), Y) places the high order byte of the label address immediately behind the low order byte in the file.

Before assembly, the jump instruction was stored in the file like this:

Opcode, xx, 00. Where the opcode is either 20 or 4C, xx is a random label number and the value 00 represents the limiter byte. After assembly the jump instruction is stored in the file with the two correct operand bytes:

Opcode, ADL, ADH. The opcode will be the same, but the operand bytes ADL (= low order byte of the label address = low order byte of the absolute jump address) and ADH (= high order byte of the label address = high order byte of the absolute jump address) have replaced the label number and the limiter byte. The entire jump instruction has now been assembled. The correct absolute address for the jump instruction has been fetched from the symbol stack and stored behind the opcode of the jump instruction.

## Branch instructions

If, during the second phase of assembly, the processor comes across a branch instruction in the file, it will have to calculate the required displacement value corresponding to that instruction. This will mean removing the label number from behind the opcode of the branch instruction. This type of instruction is assembled from the label BRINST onwards. Firstly the Y index register is incremented (the value in the Y index register will now be 01) and the processor jumps to the GETLBL subroutine. This is where the processor obtains the absolute address belonging to the label number and to which the branch must take place from the symbol stack. If the computer can locate the required label number on the symbol stack it will return from the subroutine GETLBL to the assembler main routine with the Z flag reset. As before, the high and low order label address bytes will be held in the X index register and the accumulator respectively. Now how can the processor calculate the actual displacement value of the branch instruction and store it behind the opcode? To answer this question we must assess exactly what the computer 'knows' about the branch instruction being assembled at this moment in time.

1. Upon its return from the GETLBL subroutine, the computer knows the label address to which the proposed branch is to take place. The

low order byte of the label address is stored in the X index register and the high order byte in the accumulator. These two address bytes constitute the destination address, in other words, the location to which the processor must branch.

2. The current address pointer CURAD informs the computer of the address where the opcode of the branch instruction being assembled is actually situated. This will be the source address, in other words, the location from which the processor is to effect the branch. This will consist of the contents of the current address pointer, stored in memory locations 00E6 and 00E7.

3. From the source and destination addresses the computer is able to calculate the actual displacement value:

Displacement = destination address - source address - 02.

The reason for subtracting the extra two units is because after a branch instruction has been decoded the program counter of the CPU will be pointing to the opcode of the next instruction. Once the displacement value has been calculated, this will have to be inserted behind the branch instruction being assembled. The penultimate instruction of the section of program labelled BRINST (STA- (CURADL), Y) replaces the label number behind the branch instruction with the displacement value. At the same time this will complete assembly of the branch instruction.

Now that we have become familiar with the main assembler routine, let us examine the main points of the two phases involved once more:

#### **Phase one:**

The absolute addresses of the labels together with the actual label numbers are stored on the symbol stack. Then the processor deletes the label from the file and shifts the data block following it up by three memory locations. Thus the label is overwritten. During the shift operation the next label in the file moves up three positions. In the first phase of assembly, therefore, the opcodes of the labels not yet removed from the file keep their locations. The symbol stack contains all the information required concerning the labels erased from the file. Label numbers and addresses remain unchanged during the entire assembly procedure.

#### **Phase two:**

Once all the labels have been removed from the file, all the required information concerning them will have been stored on the symbol stack and the second phase of the assembly procedure can begin. This is where the computer checks through the file for all the instruction opcodes followed by a label number. The opcodes for the JMP, JSR and branch instructions are filtered out of the file. In the case of jump instructions, the processor searches the symbol stack for the label number situated behind the instruction opcode being assembled. This gives the processor access to the address of the label, which is the absolute jump address and which is inserted directly behind the opcode of the jump instruction in the file.

Where branch instructions are concerned, the displacement value still has to be calculated. It is found by subtracting the source address from the destination address and the result is stored in the file directly behind the opcode of the branch instruction.

## The subroutine GETLBL

The central subroutine of the assembler program is the GETLBL subroutine. During this subroutine the processor obtains the address of the particular label from the symbol stack – provided of course there is a JMP, JSR or branch instruction to be assembled in the file. The function of this subroutine can be described as follows:

1. Search for the label number on the symbol stack.
2. Once this has been found, fetch the label address from the symbol stack. Place the high order byte of the label address in the X index register and the low order byte of the label address in the accumulator.
3. If the label number is not present in the symbol stack, return to the main assembler routine with the Z flag in the status register set.

As can be seen from the detailed flowchart of the GETLBL subroutine in figure 8, the first instruction loads the label number into the accumulator (at this time the contents of the Y index register are 01). The contents of the Y index register are then made equal to FF. Before we continue with the description of the GETLBL subroutine, the following points should be kept in mind:

\* The address indicated by the symbol stack pointer TABLE + LABELS is the highest vacant location in the symbol stack. During the second phase of assembly, the symbol stack increases from address TABLE + FF (= ENDAD) to address TABLE + LABELS.

\* TABLE + FF constitutes the 'bottom' of the symbol stack, whereas the top of the stack is formed by TABLE + LABELS. All the relevant table information is stored between these two ('piled on' during PASSA).

\* The search for a label number on the symbol stack starts at the bottom, at address TABLE + FF, and ends at the top, at address TABLE + LABELS.

Back to the program. After the label SYMA the processor checks whether or not the symbol stack pointer has already reached the top of the symbol stack. The instruction used to find this out is CPY-LABELS. The subsequent comparison (CMP-(TABLEL), Y) compares a label number on the symbol stack to one in the file. What exactly is the procedure for this? Before this comparison takes place the situation is as follows:

1. During PASSB the processor has encountered the opcode of an instruction which is followed by a label number. This instructions needs to be assembled, therefore the GETLBL subroutine is called.
2. The label number situated behind the opcode of the instruction must then be loaded into the accumulator.
3. At the moment the symbol stack pointer is pointing to the address location TABLE + FF. This is where the first label number was stored on the symbol stack. The processor now compares the label number in the accumulator to the one on the symbol stack by means of the instruction CMP-(TABLEL), Y). If the label numbers are different, the processor will branch back to the label SYMNXT, where the Y index register is decremented three times in succession.
4. Since the contents of the Y index register will not be zero at this point, the processor will branch to the label SYMA. Following this the processor compares the label number in the accumulator with the next label number on the symbol stack.

5. If the two label numbers are still different, the Y index register is again decremented three times so that the symbol stack pointer indicates the position where the next label is stored. If, however, the two label numbers are the same, the label number concerned will have been found.

6. The label number has now been found on the symbol stack. The address belonging to it can now be obtained from the symbol stack. First, the high order byte of the label address is retrieved from the symbol stack. The instructions for this purpose are:

```
DEY  
LDA- (TABLEL), Y  
TAX
```

As the Y index register is decremented by one, the symbol stack pointer will indicate the location where the high order byte of the label address is stored. This is then loaded in the accumulator and from there transferred to the X index register. Following this the contents of the Y index register are decremented once more, which causes the symbol stack pointer to indicate the location where the low order byte of the label address is stored. This is then loaded into the accumulator. The two instructions which perform this operation are:

```
DEY  
LDA- (TABLEL), Y
```

At this stage in the proceedings the high order byte of the label address will be held in the X index register and the low order byte of the label address will be held in the accumulator.

7. The subroutine GETLBL has yet another purpose:

- \* If the label number on the symbol stack was found to be the same as that of the instruction being assembled, the Z flag in the status register will have to be reset.
- \* If the required label number was not found on the symbol stack, the Z flag in the status register will have to be set.

To reset the Z flag the processor uses the instruction LDY # 01 directly before the label SYMB. This instruction can only be carried out if the computer has found the required label number on the symbol stack.

If, however, the processor has searched through the entire symbol stack in vain, the Z flag will have to be set. If the label number does not happen to be present on the symbol stack, the search will be discontinued when the symbol stack pointer has reached the top of the symbol stack. This is the only time that the pointer TABLE + Y will indicate the same address as TABLE + LABELS. By comparing the pointer LABELS and the contents of the Y index register, the processor is able to determine whether or not the required label number is present on the symbol stack. This comparison takes place after the label SYMA:

CPY-LABELS.

If indeed the label number turns out to be non-existent the result of this comparison will be zero. This in turn means that the Z flag will be set as mentioned and the processor will return to the main assembler routine immediately by way of the label SYMB.

Now that we have discussed the assembler routine, we can appreciate why it is so closely related to the editor routine. They both utilise the same subroutines. This is a great advantage for the programmer, for once he/she

has mastered their operation, they can be incorporated into his/her own programs. The source listing of all the subroutines described is given in the back of this book. This is very practical, regardless of whether the subroutines are used by the main monitor, editor or assembler routines. It is very useful for the novice programmer to understand how the same subroutine(s) can be used for different tasks. In addition, the source listings for the sample programs given in chapters 5 and 6 are also provided at the back of the book.

### The BRANCH routine

On several occasions in Book I it was mentioned that the EPROM contained a program which allows the calculation of displacement values pertaining to branch instructions. This program is called BRANCH and starts at address location 1FD5. The program is in fact an infinite loop (see figure 9) which can only be left by depressing either of the keys RST or ST, or by way of an external interrupt (NMI or IRQ).

Once the BRANCH routine has been started, the display will show 00 00 00. The programmer is now able to enter the low order byte of the address containing the opcode of the branch instruction. This is then shown on the two left hand digits of the display.

Following this the programmer is able to enter the low order byte of the address where the processor is to branch to. This will then be shown on the two centre digits of the display. Finally, the two right hand digits will show the actual displacement value.

When data is entered, only the low order bytes of the source and destination addresses have to be keyed in, as the 6502 processor can only branch 127 bytes forwards and 128 bytes backwards.

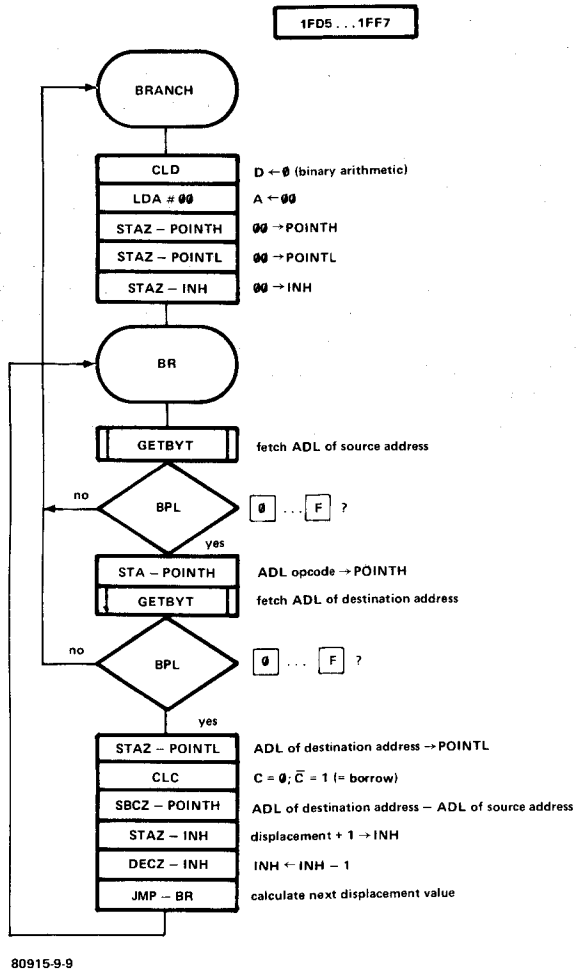
Once a displacement value has been calculated, the display will again show 00 00 00 if any of the command keys are depressed. The same will occur during the entry of the next two address bytes.

With regard to the BRANCH routine we can be fairly brief, since the displacement values are of course calculated along the same lines as described in figure 7:

To start with, the computer is placed in the binary arithmetic mode and the display buffers POINTH, POINTL and INH are loaded with the value 00. The subroutine GETBYT is then called after the label BR. As we know, this subroutine combines the values of two hexadecimal keys into one byte and holds the result in the accumulator and returns to the BRANCH routine with the N flag set.

If during the GETBYT routine a command key is depressed, the N flag will be reset and the processor will branch back to the label BRANCH upon the return from subroutine GETBYT. This also causes the display to be reset.

Once the two key values have been stored in the accumulator, the latter's contents are transferred to the display buffer POINTH. This means that POINTH now contains the low order byte of the address at which the opcode of the branch instruction is located. The subroutine GETBYT is then called a second time to obtain the low order byte of the destination address. This is then transferred to the display buffer POINTL in the



**Figure 9. The detailed flowchart of the BRANCH subroutine. This routine enables the programmer to calculate the displacement values of branch instructions.**

same manner as before. The Junior Computer can then go ahead and calculate the displacement value.

At this time the accumulator still holds the low order byte of the destination address. After clearing the carry flag the processor subtracts the low order byte of the source address (POINTH) from the contents of the accumulator. Since the carry flag was reset before the subtraction took place (which incidentally is strictly not allowed!!!) the result will be one location short. Although the wrong result is therefore stored in the display buffer INH, the processor decrements 1 the contents of INH so



fast that it is never noticed. The value contained in the display buffer INH will now be correct. The program then branches back to label BR and there the subroutine GETBYT shows the low order bytes of the source and destination addresses and the calculated displacement value on the display.

This brings us to the end of Book II. In both Books I and II we have tried to show the reader how to develop useful programs as simply as possible on a small computer. This in fact covers the description of the standard version of the Junior Computer. This does not mean, however, that it is the end of the Junior Computer. This book terminates with an appendix containing a summary of the subroutines described and a source listing of the entire monitor program with extensive commentary, a real treasure for software enthusiasts. Those of you who have successfully come to terms with Book I and II may look forward to reading Book III, for here the Junior Computer develops into a powerful personal computer by means of an elaborate, yet inexpensive hardware expansion system. Happy programming!

# The program listing of the EPROM

### *The monitor, the editor and the assembler*

The next ten pages contain a survey of the EPROM contents. It is the 'expanded' version of Appendix 3 in Book 1, which showed the contents in the form of a hex dump (in bytes only).

The listing includes the following:

1. A survey of all the RAM memory locations in page 00 (temporaries) and in page 1A (PIA addressing and locations for the NMI and IRQ jump vector).
2. The main monitor routine (1C00 ... 1CB4).
3. The main editor routine (1CB5 ... 1D4C).
4. Subroutines as part of:
  - a. the editor: SCAN (1D5C ... 1D6E);
  - b. the editor and the BRANCH routine: GETBYT (1D6F ... 1D87);
  - c. the editor and the monitor: SCAND(S) (1D88 ... 1E1F, including SHOW, CONVD and GETKEY);
  - d. the editor: RDINST (1E20 ... 1E46) and FILLWS (1E47 ... 1E5B);
  - e. the editor and the assembler: OPLEN/LENACC (1E5C ... 1E82);
  - f. the editor and the assembler: UP (1E83 ... 1EA5);
  - g. the editor: ADCEND (1EDC ... 1ED2);
  - h. the editor and the assembler: BEGIN;
  - i. the editor: ADCEND (1EDC ... 1EE9);
  - j. the editor and the assembler: RECEND (1EEA ... 1EF7) and NEXT (1EF8 ... 1F0E).
5. Look-up table LOOK, used by the monitor and the editor (subroutine CONVD) (1F0F ... 1F1E).
6. Look-up table LEN, used by the editor and the assembler (subroutine OPLEN/LENACC (1F1F ... 1F2E).
7. Subroutine GETLBL, used in the assembler (1F35 ... 1F50).
8. The main assembler routine (1F51 ... 1FD2).
9. The displacement calculation routine BRANCH (1FD5 ... 1FF7).
10. Six EPROM locations to establish the NMI, RES and IRQ vectors (1FFA ... 1FFF) and six locations for the two JMP-IND instructions involved (see chapter 3 in Book 1) (1F2F ... 1F34).  
N.B. Locations 1FF8 and 1FF9 are not used. When the EPROM is programmed they are filled with FF.
11. All the labels and names of memory locations that are used in the monitor, the editor and the assembler, in alphabetical order, plus the corresponding address.

```

0000: 1C00      LOYS  ORG  SIC00  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      TABLE *      $00EC
0310: 1C00      TABLEH *     $00ED
0320: 1C00      LABELS *      $00EE
0330: 1C00      BYTES  *      $00EF  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  = FLAGZ
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  *      $1AF8  CLK1T
0880: 1C00      CNTF  *      $1AF9  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:
0950: 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 STVZ 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      SET AD-MODE
1230: 1C1A 4C 33 1C JMP START
1240:
1250: 1C1D A9 1E    RESET LDAIM $1E PBI---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      LDAIM $03
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 00 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 4B          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 48          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    always
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    always
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 STEPA JMP
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 ADDRESS
1930: 1C96 B1 FA LDAIY POINTL GET DATA SPECIFIED
1940: 1C98 8A ASLA ASLA BY POINT
1950: 1C99 8A ASLA ASLA SHIFT LOW ORDER
1960: 1C9A 8A ASLA ASLA NIBBLE INTO HIGH ORDER NIBBLE
1970: 1C9B 8A ASLA ASLA
1980: 1C9C 85 E1 ORAZ KEY DATA WITH KEY
1990: 1C9E 91 FA STAIY POINTL RESTORE DATA
2000: 1CA0 4C 7A 1C JMP STEPA
2010:
2020: 1CA3 A2 84 ADDRESS LDYIM $84 4 SHIFTS
2030: 1CA5 86 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 85 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: CURADL IS OUT OF RANGE
2330:
2340: 1CB5 20 D3 1E EDITOR JSR BEGIN CURADL=-BEGAD
2350: 1CB8 A4 E3 LDYZ BEGADH
2360: 1CBA A6 E2 LDYZ 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: 1CD8 10 F0 BPL SEARCH COM. KEY?
2550: 1CDD 85 FA STAZ POINTL DISCARD DATA
2560: 1CDF 20 D3 1E JSR BEGIN CURADL=-BEGAD
2570: 1CE2 A0 00 SELOOP LDYIM $00
2580: 1CE4 B1 E6 LDAIY CURADL COMPARE INSTRUCTION
2590: 1CE6 C5 FB CMPZ FOINTH 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 20 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 CURADL 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: 1CFE 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 CURADL IS LESS THAN CEND
2830: 1D18 A5 FD LDAZ TEMPX LENGTH OF INSTR. TO BE INSERTED
2840: 1D1A 85 F6 STAZ BYTES

```

```

2850: 1D1C 20 47 1E      JSR  FILLWS MOVE DATA IN WS DOWNWARD BY THE AM. IN BYTES
2860: 1D1F F0 A9          BEQ  CMND  RETURN TO DISPLAY THE INSERTED DATA
2870:
2880: 1D21 C9 12          SKIP  CMPIM $12  SKIP COMMAND?
2890: 1D23 D0 07          BNE  DELETE
2900: 1D25 20 F8 1E      JSR  NEXT  SKIP TO NEXT INSTRUCTION. CURAD LESS THAN CEND?
2910: 1D28 30 A0          BMI  CMND
2920: 1D2A 10 0D          BPL  ERRA
2930:
2940: 1D2C C9 11          DELETE CMPIM $11  DELETE COMMAND?
2950: 1D2E D0 09          BNE  ERRA
2960: 1D30 20 B3 1E      JSR  UP     DELETE CURRENT INSTR. BY MOVING UP THE WS
2970: 1D33 20 EA 1E      JSR  RECDN ADJUST CURRENT END ADDRESS
2980: 1D36 4C CA 1C      JMP  CMND
2990:
3000: 1D39 A9 EE          ERRA  LDAIM SEE
3010: 1D3B 85 FB          STAZ  POINTH
3020: 1D3D 85 FA          STAZ  POINTL
3030: 1D3F 85 F9          STAZ  INH
3040: 1D41 A9 03          LDAIM $03
3050: 1D43 85 F6          STAZ  BYTES
3060: 1D45 20 8E 1D      ERRB JSR  SCANDS DISPLAY EEEEE UNTIL KEY IS RELEASED
3070: 1D48 D0 FB          BNE  ERRB
3080: 1D4A 4C CA 1C      JMP  CMND
3090:
3100:
3110:
3120:
3130:
3140:
3150:
3160:
3170:
3180:
3190:
3200:
3210:
3220:
3230:
3240:
3250:
3260:
3270: 1D4D A2 02          SCAN  LDXIM $02  FILL UP THE DISPLAY BUFFER
3280: 1D4F A0 00          LDYIM $00
3290: 1D51 B1 E6          FILBUF LDAIY CURADL START FILLING AT OP CODE
3300: 1D53 95 F9          STAX  INH
3310: 1D55 C8             IMY
3320: 1D56 CA             DEX
3330: 1D57 10 F8          BPL  FILBUF
3340: 1D59 20 5C 1E      JSR  OPLEN STORE INSTRUCTION LENGTH IN BYTES
3350: 1D5C 20 8E 1D      SCANA JSR  SCANDS DISPLAY CURRENT INSTRUCTION
3360: 1D5F D0 FB          BNE  SCANA KEY RELEASED?
3370: 1D61 20 8E 1D      SCANB JSR  SCANDS DISPLAY CURRENT INSTRUCTION
3380: 1D64 F0 FB          BEQ  SCANB ANY KEY DEPRESSED?
3390: 1D66 20 8E 1D      JSR  SCANDS DISPLAY CURRENT INSTRUCTION
3400: 1D69 F0 F6          BEQ  SCANB ANY KEY STILL DEPRESSED?
3410: 1D6B 20 F9 1D      JSR  GETKEY IF YES, RETURN WITH KEY IN ACCU
3420: 1D6E 60             RTS
3430:
3440:
3450:
3460:
3470:
3480:
3490:
3500: 1D6F 20 5C 1D      GETBYT JSR  SCANA READ HIGH ORDER NIBBLE
3510: 1D72 C9 10          CMPIM $10
3520: 1D74 10 11          BPL  BYTEND COMMAND KEY?
3530: 1D76 0A             ASLA
3540: 1D77 0A             ASLA  IF NOT, SAVE HIGH ORDER NIBBLE
3550: 1D78 0A             ASLA
3560: 1D79 0A             ASLA
3570: 1D7A 85 FE          STA  NIBBLE
3580: 1D7C 20 5C 1D      JSR  SCANA READ LOW ORDER NIBBLE
3590: 1D7F C9 10          CMPIM $10
3600: 1D81 10 04          BPL  BYTEND COMMAND KEY?
3610: 1D83 85 FE          ORA  NIBBLE IF NOT, COMPOSE BYTE
3620: 1D85 A2 FF          LDXIM $FF  SET N=1
3630: 1D87 60             BYTEND RTS
3640:
3650:
3660:
3670:
3680:
3690:
3700:
3710:
3720:
3730:
3740:
3750:
3760:
3770: 1D88 A0 00          SCAND  LDYIM $00
3780: 1D8A B1 FA          LDAIY POINTL GET DATA SPECIFIED BY POINT
3790: 1D8C 85 F9          STAZ  INH

```

```

3800: LD8E A9 7F      SCANDS LDAIM $7F
3810: ID90 8D 81 1A   STA  PADD  PA0...PA6 IS OUTPUT
3820: ID93 A2 08      LDXIM $08  ENABLE DISPLAY
3830: ID95 A4 F6      LDYZ  BYTES  FETCH LENGTH FROM BYTES
3840: ID97 A5 FB      LDAZ  POINTH OUTPUT 1ST BYTE
3850: ID99 20 CC 1D   SCDSA JSR  SHOW
3860: ID9C 88         DEY
3870: ID9D F0 0D      BEQ  SCDSB  MORE BYTES?
3880: ID9F A5 FA      LDAZ  POINTL
3890: IDA1 20 CC 1D   JSR  SHOW  IF YES, OUTPUT 2ND BYTE
3900: IDA4 88         DEY
3910: IDA5 F0 05      BEQ  SCDSB  MORE BYTES?
3920: IDA7 A5 F9      LDAZ  INH
3930: IDA9 20 CC 1D   JSR  SHOW  IF YES, OUTPUT 3RD BYTE
3940: IDAC A9 08      SCDSB LDAIM $08
3950: IDAE 8D 81 1A   STA  PADD  PA0...PA7 IS INPUT
3960:
3970: IDB1 A0 03      AK   LDYIM $03  SCAN 3 ROWS
3980: IDB3 A2 00      LDXIM $00  RESET ROW COUNTER
3990:
4000: IDB5 A9 FF      ONEKEY LDAIM SFF
4010: IDB7 8E 82 1A   AKA  STX  PBD  OUTPUT ROW NUMBER
4020: IDBA E8         INX         ENABLE FOLLOWING ROW
4030: IDBB E8         INX
4040: IDBC 2D 80 1A   AND  PAD  INPUT ROW PATTERN
4050: IDBF 88         DEY         ALL ROWS SCANNED?
4060: IDC0 D0 F5      BNE  AKA
4070: IDC2 A0 06      LDYIM $06  TURN DISPLAY OFF
4080: IDC4 8C 82 1A   STY  PBD
4090: IDC7 09 80      ORAIM $08  SET BIT7=1
4100: IDC9 49 FF      EORIM SFF  INVERT KEY PATTERN
4110: IDCB 60         RTS
4120:
4130:
4140:
4150:
4160:
4170:
4180:
4190: IDCC 48         SHOW PHA  SAVE DISPLAY
4200: IDCD 84 FC      STYZ  TEMP  SAVE Y REGISTER
4210: IDCF 4A         LSR   A
4220: IDD0 4A         LSR   A  GET HIGH ORDER NIBBLE
4230: IDD1 4A         LSR   A
4240: IDD2 4A         LSR   A
4250: IDD3 20 DF 1D   JSR  CONV D OUTPUT HIGH ORDER NIBBLE
4260: IDD6 68         PLA  GET DISPLAY AGAIN
4270: IDDF 29 0F      ANDIM $0F  MASK OFF HIGH ORDER NIBBLE
4280: IDD9 20 DF 1D   JSR  CONV D OUTPUT LOW ORDER NIBBLE
4290: IDDC A4 FC      LDYZ  TEMP  RESTORE Y REGISTER
4300: IDDE 60         RTS
4310:
4320:
4330:
4340:
4350:
4360:
4370:
4380:
4390: IDDF A8         CONV D TAY  USE NIBBLE AS INDEX
4400: IDE0 B9 0F 1F   LDAY  LOOK  FETCH SEGMENT PATTERN
4410: IDE3 8D 80 1A   STA  PAD  OUTPUT SEGMENT PATTERN
4420: IDE6 8E 82 1A   STX  PBD  OUTPUT DIGIT ENABLE
4430: IDE9 A0 7F      LDYIM $7F
4440: IDEB 88         DELAY DEY  DELAY 500 US APPROX.
4450: IDEC 10 FD      BPL  DELAY
4460: IDEE 8C 80 1A   STY  PAD  TURN SEGMENTS OFF
4470: IDF1 A0 06      LDYIM $06
4480: IDF3 8C 82 1A   STY  PBD  TURN DISPLAY OFF
4490: IDF6 E8         INX
4500: IDF7 E8         INX  ENABLE NEXT DIGIT
4510: IDF8 60         RTS
4520:
4530:
4540:
4550:
4560:
4570: IDP9 A2 21      GETKEY LDXIM $21  START AT ROW 0
4580: IDFB A0 01      GETKEA LDYIM $01  GET ONE ROW
4590: IDFD 20 B5 1D   JSR  ONEKEY A=0, NO KEY DEPRESSED
4600: IE00 D0 07      BNE  KEYIN
4610: IE02 E0 27      CPXIM $27
4620: IE04 D0 F5      BNE  GETKEA  EACH ROW SCANNED?
4630: IE06 A9 15      LDAIM $15  RETURN IF INVALID KEY
4640: IE08 60         RTS
4650: IE09 A0 FF      KEYIN LDYIM SFF
4660: IE0B 0A         KEYINA ASLA  SHIFT LEFT UNTIL Y=KEY NUMBER
4670: IE0C B0 03      BCS  KEYINB
4680: IE0E C8         INY
4690: IE10 10 FA      BPL  KEYINA
4700: IE11 8A         KEYINB TXA
4710: IE12 29 0F      ANDIM $0F  MASK MSD
4720: IE14 4A         LSR   A  DEVIDE BY 2
4730: IE15 AA         TRX
4740: IE16 98         TYA

```

```

4750: 1E17 18 03          BPL  KEYIND
4760: 1E19 18          KEYINC  CLC
4770: 1E1A 69 07        ADCIM $07  ADD ROW OFFSET
4780: 1E1C CA          KEYIND  DEX
4790: 1E1D D8 FA        BNE  KEYINC
4800: 1E1F 60          RTS

4810:
4820:
4830:
4840:
4850:
4860:
4870:
4880: 1E20 20 6F 1D  RDINST JSR  GETBYT  READ OP CODE
4890: 1E23 18 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  TEMPK
4940: 1E2E C6 F7        DECZ  COUNT
4950: 1E30 F0 12          BEQ  RDA  1 BYTE INSTRUCTION?
4960: 1E32 28 6F 1D      JSR  GETBYT IF NOT, READ FIRST OPERAND
4970: 1E35 18 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 SFF  N=1
5050: 1E46 60          RDB  RTS

5060:
5070:
5080:
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 D8 F6          BNE  WS  IF NOT, CONTINUE
5200: 1E5B 60          RTS

5210:
5220:
5230:
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
5420: 1E7D 8C 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:
5470:
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 DA          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:
5690:

```

DOWN MOVES A DATA FIELD BETWEEN CURAD AND CEND UPWARD BY THE AMOUNT IN BYTES

DOWN MOVES A DATA FIELD BETWEEN CURAD AND ENDAD DOWNWARD BY THE AMOUNT IN BYTES



```

5700:
5710: 1EA6 A5 E8      DOWN  LDAZ  CENDL
5720: 1EA8 85 EA       STA2 MOVADL MOVAD:=CEND
5730: 1EAA A5 E9       LDAZ  CENDH
5740: 1EAC 85 EB       STA2 MOVADH
5750: 1EAE A0 00      DNLOOP LDYIM $00
5760: 1EB0 B1 EA       LDAY1 MOVADL MOVE DOWNWARD BY THE AMOUNT IN BYTES
5770: 1EB2 A4 F6       LDY2  BYTES
5780: 1EB4 91 EA       STAY1 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          DNE
5860: 1EC3 A5 EA       DNA    LDAZ  MOVADL
5870: 1EC5 E9 01       SBCIM $01
5880: 1EC7 85 EA       STA2 MOVADL
5890: 1EC9 A5 EB       LDAZ  MOVADH MOVAD:=MOVAD-1
5900: 1ECB E9 00      SBCIM $00
5910: 1ECD 85 EB       STA2 MOVADH
5920: 1ECF 4C AE 1E   JMP  DNLOOP
5930: 1ED2 60          DNEND  RTS
5940:
5950:
5960:
5970:
5980: 1ED3 A5 E2      BEGIN  LDAZ  BEGADL
5990: 1ED5 85 E6       STA2  CURADL
6000: 1ED7 A5 E3       LDAZ  BEGADH CURAD:=BEGAD
6010: 1ED9 85 E7       STA2  CURADH
6020: 1EDB 60          RTS
6030:
6040:
6050:
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       STA2  CENDL
6110: 1EE3 A5 E9       LDAZ  CENDH
6120: 1EE5 69 00      ADCIM $00
6130: 1EE7 85 E9       STA2  CENDH
6140: 1EE9 60          RTS
6150:
6160:
6170:
6180:
6190: 1EEA 38          RECENT SEC
6200: 1EEB A5 E8       LDAZ  CENDL
6210: 1EED E5 F6      SBCZ  BYTES  CEND:=CEND-BYTES
6220: 1EEF 85 E8       STA2  CENDL
6230: 1EF1 A5 E9       LDAZ  CENDH
6240: 1EF3 E9 00      SBCIM $00
6250: 1EF5 85 E9       STA2  CENDH
6260: 1EF7 60          RTS
6270:
6280:
6290:
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       STA2  CURADL
6350: 1EFF A5 E7       LDAZ  CURADH
6360: 1F01 69 00      ADCIM $00
6370: 1F03 85 E7       STA2  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:
6460:
6470:
6480:
6490:
6500:
6510: 1F0F 40          THE LOOKUP TABLE "LOOK" IS USED, TO CONVERT
6520: 1F10 79          A HEX NUMBER INTO A 7 SEGMENT PATTERN.
6530: 1F11 24          THE LOOKUP TABLE "LEN" IS USED, TO CONVERT AN
6540: 1F12 30          INSTRUCTION INTO AN INSTRUCTION LENGTH.
6550: 1F13 19
6560: 1F14 12
6570: 1F15 02
6580: 1F16 78
6590: 1F17 00
6600: 1F18 10
6610: 1F19 08
6620: 1F1A 03
6630: 1F1B 46
6640: 1F1C 21
LOOK  =  $40  "0"
      =  $79  "1"
      =  $24  "2"
      =  $30  "3"
      =  $19  "4"
      =  $12  "5"
      =  $02  "6"
      =  $78  "7"
      =  $00  "8"
      =  $10  "9"
      =  $08  "A"
      =  $03  "B"
      =  $46  "C"
      =  $21  "D"

```

```

6650: 1F1D 06          =      S06      "E"
6660: 1F1E 0E          =      S0E      "F"
6670:
6680: 1F1F 02          LEN =      S02
6690: 1F20 02          =      S02
6700: 1F21 02          =      S02
6710: 1F22 01          =      S01
6720: 1F23 02          =      S02
6730: 1F24 02          =      S02
6740: 1F25 02          =      S02
6750: 1F26 01          =      S01
6760: 1F27 01          =      S01
6770: 1F28 02          =      S02
6780: 1F29 01          =      S01
6790: 1F2A 01          =      S01
6800: 1F2B 03          =      S03
6810: 1F2C 03          =      S03
6820: 1F2D 03          =      S03
6830: 1F2E 03          =      S03
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:
6890:
6900:
6910:
6920:
6930:
6940:
6950: 1F35 B1 E6          GETLBL LDAYI CURADD  FETCH CURRENT LABEL NUMBER FROM WS
6960: 1F37 A0 FF          LDYIM SFF    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 TABLE LABEL NR. IN WS = LABEL NR. ON PSEUDO STACK?
7000: 1F3F D0 0A          BNE  SYMNXT
7010: 1F41 88          DEY
7020: 1F42 B1 EC          LDAYI TABLE IF YES, GET HIGH ORDER ADD
7030: 1F44 AA          TAX
7040: 1F45 88          DEY
7050: 1F46 B1 EC          LDAYI TABLE GET LOW ORDER ADD
7060: 1F48 A0 01          LDYIM S01   PREPARE Y REGISTER
7070: 1F4A 00          SYMB  RTS
7080:
7090: 1F4B 88          SYMNXT DEY
7100: 1F4C 88          DEY
7110: 1F4D 88          DEY
7120: 1F4E D0 E9          BNE  SYMA
7130: 1F50 60          RTS
7140:
7150:
7160:
7170:
7180:
7190:
7200:
7210:
7220:
7230:
7240:
7250:
7260:
7270:
7280:
7290:
7300:
7310:
7320:
7330:
7340: 1F51 38          ASSEMB SEC
7350: 1F52 A5 E4          LDAZ  ENDADD
7360: 1F54 E9 FF          SRCIM SFF
7370: 1F56 85 EC          STA2  TABLE: =ENDAD-$FF
7380: 1F58 A5 E5          LDAZ  ENDADH
7390: 1F5A E9 0D          SRCIM S00
7400: 1F5C 85 ED          STA2  TABLEH
7410: 1F5E A9 FF          LDAIM SFF
7420: 1F60 85 EE          STA2  LABELS
7430: 1F62 20 D3 1E      JSR  BEGIN  CURAD: =BEGAD
7440:
7450: 1F65 20 5C 1E      PASSA JSR  OPEN  START PASS ONE, GET CURR. INSTR.
7460: 1F68 A0 00          LDYIM S00
7470: 1F6A B1 E6          LDAYI CURADD  FETCH CURRENT INSTRUCTION
7480: 1F6C C9 FF          CMPIM SFF   IS THE CURRENT INSTR. A LABEL?
7490: 1F6E D0 1D          BNE  NXTINS
7500: 1F70 C8          INY
7510: 1F71 B1 E6          LDAYI CURADD  IF YES, FETCH LABEL NR.
7520: 1F73 A4 EE          LDYZ LABELS
7530: 1F75 91 EC          STAYI TABLE DEPOSIT LABEL NR. ON SYMBOL STACK
7540: 1F77 88          DEY
7550: 1F78 A5 E7          LDAZ  CURADH  GET HIGH ORDER ADD
7560: 1F7A 91 EC          STAYI TABLE DEPOSIT ON SYMBOL STACK
7570: 1F7C 88          DEY
7580: 1F7D A5 E6          LDAZ  CURADD  GET LOW ORDER ADD
7590: 1F7F 91 EC          STAYI TABLE DEPOSIT ON SYMBOL STACK

```

GETLBL IS AN ASSEMBLER SUBROUTINE. IT SEARCHES FOR LABELS ON THE SYMBOL PSEUDO STACK. IF THIS STACK CONTAINS A VALID LABEL, IT RETURNS WITH THE HIGH ORDER LABEL ADDRESS IN X AND THE LOW ORDER LABEL ADDRESS IN A. IF NO VALID LABEL IS FOUND, IT RETURNS WITH Z=1.

ASSEMBLER MAIN ROUTINE

FOLLOWING INSTRUCTIONS ARE ASSEMBLED:

JSR INSTRUCTION  
JMP INSTRUCTION  
BRANCH INSTRUCTIONS

```

7600: 1F01 06          DEY
7610: 1F03 04 FE      JSR   LABELS ADJUST PSEUDO STACK POINTER
7620: 1F04 20 03 1E  JSR   UP      DELETE CURRENT LABEL IN WS
7630: 1F07 20 EA 1E   JSR   RECEND  ADJUST CURRENT END ADD
7640: 1F0A 4C 65 1F   JMF   PASSA  LOOK FOR MORE LABELS
7650:
7660: 1F0D 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      LDYAI 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 ENDA0 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      STAYI CURADL STORE LOW ORDER ADD
7900: 1FBE 8A          TXA
7910: 1FBF C8          INY
7920: 1FC0 91 E6      STAYI 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      STAYI CURADL INSERT BRANCH OFFSET IN WS
8030: 1FD2 4C AA 1F   JMP   PB
8040:
8050:
8060:
8070:
8080:
8090:
8100:
8110:
8120:
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:
8330:
8340:
8350:
8360:
8370:
8380:
8390:
8400:
8410:
8420:
8430:
8440:
8450:
8460:
8470:
8480:
8490:
8500:
8510:
8520:
8530:

```

THE SUBROUTINE BRANCH COMPUTES THE OFFSET OF BRANCH INSTRUCTIONS. THE 2 RIGHT HAND DISPLAYS SHOW THE COMPUTED OFFSET DEFINED BY THE 4 LEFT HAND DISPLAYS. THE PROGRAM MUST BE STOPPED WITH THE RESET KEY.

VECTORS AT THE END OF THE MEMORY:

```

1FFA $2F  NMI VECTOR
1FFB $1F
1FFC $1D  RESET VECTOR
1FFD $1C
1FFE $32  IRQ OR BRK VECTOR
1FFF $1F

```

END OF JUNIOR'S MONITOR

8540:	ACC	00F3	ADCEND	1EDC	ADDRESS	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	CMMD	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	GETREA	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:	NM1H	1A7B	NM1L	1A7A	NXTINS	1F8D	ONEKEY	1DB5
8750:	OPLN	1E5C	PADD	1AB1	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	SCAMB	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	STPA	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				

## Appendix 2

# Listings of the programs used in chapters 5 and 6

```

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 LDAM $00    DISPLAY 000000
0210: 0202 85 F9                                STAZ INH
0220: 0204 85 FA                                STAZ POINTL
0230: 0206 85 FB                                STAZ POINTH
0240: 0208 20 8F 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 06 02                        JMP    DA       WAIT FOR A NEW BYTE
0300:
0310:
0320:                                SUBROUTINES OF THE CONVERSION PROGRAM
0330:
0340: 0217 20 2E 02                        HEXDEC JSR    COMMN 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    COMMN  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 85 FA                                GRAZ 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                                COMMN 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                                LDAA HEXL    CORRECT SUBTRACTION ERROR
0520: 0238 69 0A                                ADXIM $0A
0530: 023A 60                                RTS
0540:
0550: 023B 38                                SURTRA SEC
0560: 023C A5 D7                                LDAA HEXL   16 BIT SUBTRACTION
0570: 023E E9 0A                                SBCIM $0A
0580: 0240 85 D7                                STAZ HEXL
0590: 0242 A5 D8                                LDAA 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
COMMN 022E    DA    0208    DISPL 0200    GETBYT 1D6F
HD     0223    HEXDEC 0217    HEXH   00D8    HEXL   02D7
INH    00F9    POINTH 00FB    POINTL 00FA    SUBTRA 023B
SUB    024C

SYMBOL TABLE
HEXH   00D8    INH    00F9    POINTL 00FA
DISPL  0200    DA     0208    HEXDEC 0217
POINTH 00FB    COMMN 022E    SUB    024C
GETBYT 1D6F

```

```

0610: DEMO ROUTINE
0620:
0630: 0000 ORG $0000
0640:
0650: I/O DEFINITION
0660:
0670: 0000 PAD * $1A00 DATA REGISTER
0680: 0000 PADD * $1A01 DATA DIRECTION REGISTER
0690: 0000 PBD * $1A02 DATA REGISTER
0700: 0000 PBDD * $1A03 DATA DIRECTION REGISTER
0710:
0720: 0000 A2 00 DEMO LDXIM $00
0730: 0002 8E 81 1A STX PADD PA0...PA7 IS INPUT
0740: 0005 E8 INX
0750: 0006 8E 83 1A STX PBDD PB0 IS OUTPUT
0760:
0770: 0009 AD 80 1A FREQ LDA PAD READ SWITCH PATTERN
0780: 000C 49 FF BORIM SFF INVERT PATTERN
0790: 000E A0 00 LDYIM $00 B0 IS ZERO
0800: 0010 8C 82 1A STY PBD TOGGLE SPEAKER ON
0810: 0013 20 20 00 JSR DELAY DELAY = SWITCHES * LOOP TIME
0820: 0016 C8 INY
0830: 0017 8C 82 1A STY PBD TOGGLE SPEAKER OFF
0840: 001A 20 20 00 JSR DELAY DELAY = SWITCHES * LOOP TIME
0850: 001D 4C 09 00 JMP FREQ RETURN
0860:
0870: SUBROUTINE DELAY
0880:
0890: 0020 AA DELAY TAX X-REGISTER IS THE DELAY COUNTER
0900: 0021 CA DEL DEX DELAY LOOP
0910: 0022 D0 FD BNE DEL
0920: 0024 60 RTS
0930:

```

SYMBOL TABLE

DELAY	0020	DEL	0021	DEMO	0000	FREQ	0009
PADD	1A01	PAD	1A00	PBDD	1A03	PBD	1A02

SYMBOL TABLE

DEMO	0000	FREQ	0009	DELAY	0020	DEL	0021
PAD	1A00	PADD	1A01	PBD	1A02	PBDD	1A03

```

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 LDAlM SF0  PA7...PA4 IS OUTPUT
0200: 0002 8D 81 1A STA  PADD  PA3...PA0 IS INPUT
0210: 0005 A9 01     LDAlM $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     LDAlM $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 LDAlM $00  B0 = 0
0360: 0026 8D 82 1A STA  PBD   TOGGLE SPEAKER ON
0370: 0029 BE 00 1A LXKY DEL  FETCH THE FREQUENCY
0380: 002C 20 AA 00 TA JSR  EQUAL  EQUALIZE 22 MICRO SEC.
0390: 002F CA        DEX   TA    HALF PERIODE PASSED?
0400: 0030 D0 FA      BNE  TA    WAIT, IF NOT
0410: 0032 A9 01     LDAlM $01  B0 = 1
0420: 0034 8D 82 1A STA  PBD   TOGGLE SPEAKER OFF
0430: 0037 BE 00 1A LXKY 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   TA    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 LDALM SF7 ALL ROWS ARE ONE
0040: 0046 85 D9 STAZ ROW
0050: 0048 A2 04 LDXM S04
0060: 004A CA DEY RETURN IF INVALID ROW
0070: 004B 38 F7 KEYA BMT KEYVAL IS FOUND
0080: 004D 06 D9 ASLZ ROW SPECIFIED ROW IS ZERO
0090: 004F A5 D9 LDZ 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 S0F SPECIFIED ROW, OUTPUT
0130: 0059 C9 0F CMPIM S0F NEXT ROW NUMBER
0140: 005B F8 ED BEY KEYA
0150: 005D 86 D8 STZ TEMPK SAVE ROW NUMBER
0160: 005F 85 DA STAZ KEY SAVE COLUMN NUMBER
0170: 0061 A2 00 LDXM S00
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 E8 04 CPXIM S04 ALL ROWS SCANNED?
0220: 006A D8 F7 BNE KEYB IF NOT CONTINUE
0230: 006C F8 D6 BEY KEYVAL RETURN IF INVALID ROW NUMBER
0240: 006E A5 D8 ROWA LDZ TEMPK GET ROW NUMBER AGAIN
0250: 0070 C9 03 CMPIM S03 ROW 0?
0260: 0072 D0 04 BNE ROWB
0270: 0074 8A TXA
0280: 0075 4C 94 00 JMP KEYC
0290:
0300: 0078 C9 02 ROWS CMPIM S02 ROW 1?
0310: 007A D8 06 BNE ROWC
0320: 007C 8A TXA
0330: 007D 18 CLC
0340: 007E 69 04 ADCIM S04
0350: 0080 D0 12 BNE KEYC
0360:
0370: 0082 C9 01 ROWC CMPIM S01 ROW 2?
0380: 0084 D8 06 BNE ROWD
0390: 0086 8A TXA
0400: 0087 18 CLC
0410: 0088 69 08 ADCIM S08
0420: 008A D0 08 BNE KEYC
0430:
0440: 008C C9 00 ROWD CMPIM S00 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 S0C
0490:
0500: 0094 85 DA KEYC STAZ KEY SAVE KEY VALUE
0510: 0096 A9 00 LDALM S00 RESET PORT A
0520: 0098 8D 80 1A STA PAD
0530: 009B 60 RTS
0540:
0550: 009C AD 80 1A KEYIN LDA PAD MASK OFF HIGH ORDER NIBBLE
0560: 009F 29 0F ANDIM S0F IF NO KEY: ACCU = $80
0570: 00A1 49 0F BORMI S0F
0580: 00A3 60 RTS
0590:
0600: 00A4 A0 FF DELAY LDYIM SFF 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

```



```

0010:          INPUT ROUTINE
0020:
0030: 0200          ORG    S0200
0040:
0050:          TEMPORARY DATA BUFFERS IN PAGE ZERO
0060:
0070: 0200          ROW    *    S00D9
0080: 0200          KEY    *    S00DA
0090: 0200          TEMPX  *    S00DB
0100: 0200          NOTEL  *    S00DC    NOTE POINTER
0110: 0200          NOTEH  *    S00DD
0120: 0200          LENGTH *    S00DE    TIME OF A DEPRESSED KEY
0130: 0200          ENDL  *    S00DF    END OF THE NOTE BUFFER
0140:
0150:          INTERVAL TIMER
0160:
0170: 0200          CNTA   *    $1AF4    DISABLE TIMER IRQ
0180: 0200          CNTG   *    $1AFE    ENABLE TIMER IRQ, CLK64T
0190:
0200:          GOTO MONITOR
0210:
0220: 0200          RESET  *    $1C1D    NEW I/O DEFINITION
0230:
0240:          I/O DEFINITION
0250:
0260: 0200          PAD    *    $1A80
0270: 0200          PADD   *    $1A81
0280: 0200          PBD    *    $1A82
0290: 0200          PBDD   *    $1A83
0300:
0310:          IRQ VECTOR
0320:
0330: 0200          IRQL   *    $1A7E
0340: 0200          IRQH   *    $1A7F
0350:
0360:
0370:          START OF THE INPUT PROGRAM
0380:
0390: 0200 78          INPUT SET    DISABLE IRQ LINE
0400: 0201 D8          CLD
0410: 0202 A9 20          LDAM IRQIN    SET UP IRQ VECTOR
0420: 0204 8D 7E 1A          STA  IRQL
0430: 0207 A9 1A          LDAM IRQIN    /256
0440: 0209 8D 7F 1A          STA  IRQH
0450: 020C A9 F0          LDAM SF0    PA7...PA4 IS OUTPUT, PA3...PA0 IS INPUT
0460: 020E 8D 81 1A          STA  PADD
0470: 0211 A9 81          LDAM S01
0480: 0213 8D 83 1A          STA  PBD    PB0 IS OUTPUT FOR SPEAKER
0490: 0216 8D 82 1A          STA  PBD    TOGGLE SPEAKER OFF
0500: 0219 85 DD          STAZ  NOTEH  HIGH ORDER BYTE OF NOTE POINTER
0510: 021B A0 00          LDYIM S00
0520: 021D 8C 00 1A          STY  PAD    SET ALL ROWS ZERO
0530: 0220 84 DC          STYZ  NOTEL  LOW ORDER BYTE OF NOTE POINTER
0540: 0222 A0 D8          LDYIM S08    DEFINE ENADDRESS OF INPUT BUFFER
0550: 0224 84 DF          STYZ  ENDL
0560: 0226 A9 77          LDAM S77    LOAD BOF CHARACTER
0570: 0228 91 DC          INA  STAIY  NOTEL  FILLUP WORKSPACE WITH EOFPS
0580: 022A 88          DEY
0590: 022B C0 FF          CPYIM SFF    WS FILLED UP?
0600: 022D D0 F9          BNE  INA.    IF NOT CONTINUE
0610: 022F 20 E8 02          KEYSCH JSR  KEYIN  ANY KEY DEPRESSED?
0620: 0232 F0 FB          BEQ  KEYSCH  WAIT IF NO KEY IS DEPRESSED
0630: 0234 20 E8 02          JSR  DELAY  DEBOUNCE KEYBOARD
0640: 0237 20 E8 02          JSR  KEYIN  STILL ANY KEY DEPRESSED
0650: 023A F0 FB          BEQ  KEYSCH  IF YES, CONTINUE
0660: 023C 20 88 02          JSR  KEVAL  COMPUTE KEY NUMBER
0670: 023F A9 00          LDALM S00
0680: 0241 85 DE          STAZ  LENGTH  RESET TIME COUNTER
0690: 0243 A9 FF          LDALM SFF    START TIMER, ENABLE TIMER IRQ,
0700: 0245 8D FE 1A          STA  CNTG    RESET IRQ LINE
0710: 0248 58          CLI
0720: 0249 A4 0A          LDY2  KEY    LOOKUP CONVERSION BY KEY VALUE
0730: 024B A9 00          LDALM S00    TOGGLE SPEAKER ON
0740: 024D 8D 82 1A          STA  PBD    PB0 IS LOG 0
0750: 0250 BE 00 1A          LDYX  DEL    FETCH DELAY
0760: 0253 20 E8 02          TA   JSR  EQUAL  EQUALIZE 20 MIKRO SEC
0770: 0256 CA          DEX
0780: 0257 D0 FA          BNE  TA     DELAY
0790: 0259 A9 01          LDALM S01    TOGGLE SPEAKER OFF
0800: 025B 8D 82 1A          STA  PBD
0810: 025E BE 00 1A          LDYX  DEL    FETCH DELAY AGAIN
0820: 0261 20 E8 02          TB   JSR  KEYIN  ANY KEY STILL DEPRESSED?
0830: 0264 F0 05          BEQ  STORE  BRANCH IF KEY IS RELEASED
0840: 0266 CA          DEX
0850: 0267 D0 F8          BNE  TB
0860: 0269 F0 E0          BEQ  TONE  CONTINUE AS LONG AS A KEY IS DEPRESSED
0870: 026B 8D F4 1A          STA  CNTA   RESET IRQ LINE,DISABLE TIMER IRQ
0880: 026E A5 DC          LDAAZ  NOTEL
0890: 0270 C5 DF          CMPEQ ENDL  IS WORKSPACE FULL?
0900: 0272 F0 11          BEQ  ST     IF YES, EXIT HERE
0910: 0274 98          TYA
0920: 0275 A0 00          LDYIM S00
0930: 0277 91 DC          STAIY  NOTEL  STORE KEY VALUE IN WS
0940: 0279 C8          INY
0950: 027A A5 DE          LDAAZ  LENGTH  GET TIME OF THE DEPRESSED KEY

```

```

0960: 027C 91 DC          STAYZ  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    KEYSN
1000: 0285 4C 1D 1C ST   JMP    RESET  BACK TO MONITOR
1010:
1020:
0010:                      SUBROUTINES OF THE INPUT PROGRAM
0020:
0030: 0288 A9 F7          KEYVAL LDAIM S07  ALL ROWS ARE ONE
0040: 028A 85 D9          STAZ  ROW
0050: 028C A2 04          LDZIM S04  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 88 LA       STA  PAD   OUTPUT ROW NUMBER
0110: 0298 AD 80 LA       LDA  PAD   IF NO KEY IS DEPRESSED IN THE
0120: 029B 29 0F          ANDIM S0F  SPECIFIED ROW, OUTPUT
0130: 029D C9 0F          CMPIM S0F  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          LDZIM S00
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          CPZIM S04  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 S03  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 S02  ROW 1?
0310: 02BE D0 06          BNE  ROWC
0320: 02C0 8A            TXA
0330: 02C1 18            CLC
0340: 02C2 69 04          ADCIM S04
0350: 02C4 D0 12          BNE  KEYC
0360:
0370: 02C6 C9 01          ROWC  CMPIM S01  ROW 2?
0380: 02CB D0 06          BNE  ROWD
0390: 02CA 8A            TXA
0400: 02CB 18            CLC
0410: 02CC 69 08          ADCIM S08
0420: 02CE D0 08          BNE  KEYC
0430:
0440: 02D0 C9 00          ROWD  CMPIM S00  ROW 3?
0450: 02D2 D0 04          BNE  KEYVAL RETURN, IF ROW IS INVALID
0460: 02D4 8A            TXA
0470: 02D5 18            CLC
0480: 02D6 69 0C          ADCIM S0C
0490:
0500: 02D8 85 DA          KEYC  STAZ  KEY   SAVE KEY VALUE
0510: 02DA A9 00          LDAIM S00  RESET PORT A
0520: 02DC 8D 80 LA       STA  PAD
0530: 02DF 60            RTS
0540:
0550: 02E0 AD 80 LA       KEYIN LDA  PAD
0560: 02E3 29 0F          ANDIM S0F  MASK OFF HIGH ORDER NIBBLE
0570: 02E5 49 0F          BORLH S0F  IF NO KEY: ACCU = $00
0580: 02E7 60            RTS
0590:
0600: 02E8 A0 FF          DELAY LDYIM SFF  SET DELAY COUNTER
0610: 02EA 88            DEY
0620: 02EB D0 FD          BNE  DELA   DELA  TIME OUT ?
0630: 02ED 60            RTS
0640:
0650: 02EE EA            EQUAL  NOP      EQUALIZE 20 MICRO SBC
0660: 02EF EA            NOP
0670: 02F0 EA            NOP
0680: 02F1 EA            NOP
0690: 02F2 EA            NOP
0700: 02F3 60            RTS
0710:

```

```

0720: 1A00          ORG 11A00
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 11A20
0950:
0960:          TIMER INTERRUPT PROGRAM
0970:
0980: 1A20 48      IRQIN  PMA      SAVE ACCU
0990: 1A21 E6 DE    INCZ  LENGTH INCREMENT TIME
1000: 1A23 A9 FF    LDALM SFF  TIMER OFFSET IS SFF
1010: 1A25 8D FE 1A STA  CNTG  START TIMER AGAIN
1020: 1A26 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 0200     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
PBD 1A83      PBD 1A82      RESET 1C1D    ROMA 02B2
ROWB 02EC     ROWC 02C6     ROWD 02D0     ROW 00D9
ST 0205       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 0200
INA 0228      KEYSCN 022F   TONE 024B     TA 0253
TB 0261       STORE 026B    ST 0285      KEYVAL 0288
KEYA 028E     KEYB 02A7     ROMA 02B2     ROWB 029C
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     PBD 1A83     CNTA 1AF4
CNTG 1AFE     RESET 1C1D

```

```

0010: REPEAT ROUTINE
0020:
0030: 0000 ORG 50000
0040:
0050: TEMPORARY DATABUFFERS IN PAGE ZERO
0060:
0070: 0000 KEY * 500DA
0080: 0000 NOTE1 * 500DC
0090: 0000 NOTEH * 500DD
0100: 0000 LENGTH * 500DE
0110:
0120: INTERVAL TIMER
0130:
0140: 0000 CNTA * 51AF4 DISABLE TIMER IRQ
0150: 0000 CNTD * 51AF7 DISABLE TIMER IRQ, CLK1KT
0160: 0000 CNTG * 51AFE ENABLE TIMER IRQ, CLK64T
0170: 0000 RDFLAG * 51AD5 B7 IS TIMER FLAG
0180:
0190: GOTO MONITOR
0200:
0210: 0000 RESET * 51C1D NEW I/O DEFINITION
0220:
0230: I/O DEFINITION
0240:
0250: 0000 PBD * 51A62
0260: 0000 PBD0 * 51A83
0270:
0280: IRQ VECTOR
0290:
0300: 0000 IRQL * 51A7E
0310: 0000 IRQH * 51A7F
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 IRQORE SET UP IRQ VECTOR
0390: 0004 8D 7E 1A STA IRQL
0400: 0007 A9 1A LDAIM IRQORE /256
0410: 0009 8D 7F 1A STA IRQH
0420: 000C A9 01 LDAIM S01 PB0 IS OUTPUT
0430: 000E 8D 83 1A STA PBD0
0440: 0011 8D 82 1A STA PBD TOGGLE SPEAKER OFF
0450: 0014 85 DD STAZ NOTEH SET NOTE POINTER
0460: 0016 A9 00 LDAIM S00
0470: 0018 85 DC STAZ NOTE1 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 SFF SET TIMER ENABLE TIMER IRQ
0520: 0020 8D FE 1A STA CNTG
0530: 0023 A0 00 LDYIM S00 FETCH NOTE
0540: 0025 B1 DC LDYIM NOTE1
0550: 0027 85 DA STAZ KEY
0560: 0029 C8 INY FETCH LENGTH
0570: 002A B1 DC LDYIM NOTE1
0580: 002C 85 DE STAZ LENGTH
0590: 002E A4 DA LDYZ KEY LOOKUP CONVERSION
0600:
0610: 0030 A9 00 TONE LDAIM S00 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 MICRO SEC*X
0670: 003E A9 01 LDAIM S01 TOGGLE SPEAKER OFF
0680: 0040 8D 82 1A STA PBD
0690: 0043 BE 00 1A LDXY DEL GET FREQUENCY AGAIN
0700: 0046 A5 DE LDZ LENGTH GET LENGTH
0710: 0048 30 00 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 S04 LOOP TIME = 4*CNTD*PRESET
0770: 0054 A9 30 TONED LDAIM S30 PRESET = S30
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 NOTE1 ADJUST NOTE POINTER
0850: 0063 E6 DC INCZ NOTE1
0860: 0065 A0 00 LDYIM S00
0870: 0067 B1 DC LDYIM NOTE1 END OF NOTE BUFFER?
0880: 0069 C9 77 CMTIM S77 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    EQUALB JMP      EQUALB
0070: 0074 EA          EQUALB NOP
0080: 0075 4C 78 00    EQUALB JMP      END
0090: 0076 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          DECC LENGTH DECREMENT TIME
0390: 1A33 A9 FF          LDWIM SFF    TIMER OFFSET IS SFF
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
CNTG 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     PBD0 1A83
PBD 1A62      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
PBD0 1A83   RDFLAG 1AD5   CNTG 1AF4     CNTD 1AF7
CNTG 1AFE   RESET 1C1D

```