

the practical introduction  
to a powerful system

# JUNIOR COMPUTER



Book 1

Elektor

# The Elektor Junior Computer

the practical introduction  
to a powerful system

A.Nachtmann  
G.H.Nachbar

Elektor Publishers Ltd.

**Copyright © 1980 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 05 x

## Foreword

This book sets out to prove that anybody can build and use a computer. The Junior Computer was designed to be simple, inexpensive and yet have full programming potential. It is a complete microcomputer on a single board and incorporates the modern 6502 microprocessor. Once the basics of programming have been mastered, the Junior Computer can be expanded into a more sophisticated system.

The Junior Computer Book 1 consists of four chapters:

Chapter 1 provides detailed building instructions together with a full description of the Junior Computer's internal and external structure.

Chapter 2 deals with the basics in programming: how to 'compute' in binary.

Chapter 3 shows how to bring the completed Junior Computer to life and start communicating with it.

Chapter 4 ends the book with a few practical programming examples. Initially simple, they prepare the way for more complex operations to be considered in book 2.

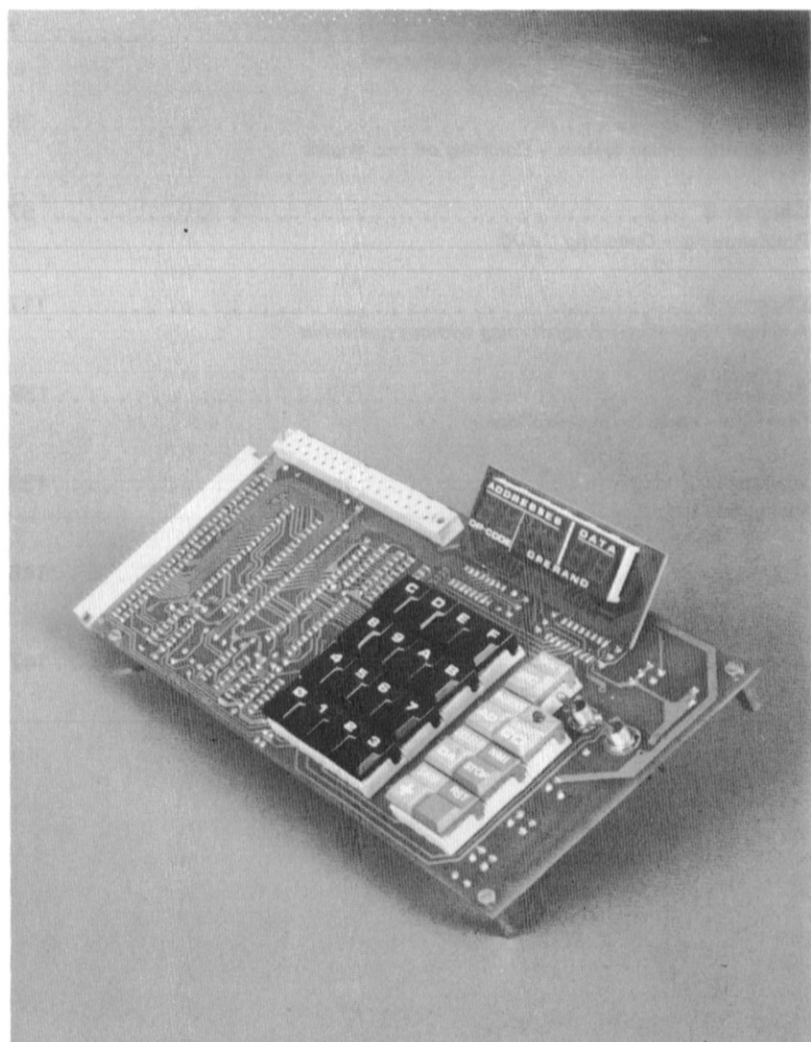
Now that your curiosity has been aroused, read on and satisfy it!

*The authors.*

**The printed circuit boards mentioned in this book are available from the Elektor Printed Circuit Board Service. For further information you are referred to the most recent Elektor issue. This will also contain details about the Elektor Software Service.**

# Contents

<b>Chapter 1</b> . . . . .	<b>7</b>
<b>Getting acquainted with the "Junior Computer"</b>	
<b>Chapter 2</b> . . . . .	<b>35</b>
<b>The binary number system – <i>Counting on two fingers</i></b>	
<b>Chapter 3</b> . . . . .	<b>57</b>
<b>Programming – <i>Operating the JC</i></b>	
<b>Chapter 4</b> . . . . .	<b>117</b>
<b>A simple beginning – <i>Programming without headaches</i></b>	
<b>Appendix 1</b> . . . . .	<b>138</b>
<b>Instruction codes in numerical order</b>	
<b>Appendix 2</b> . . . . .	<b>139</b>
<b>Instruction listing</b>	
<b>Appendix 3</b> . . . . .	<b>145</b>
<b>Hex dump of the monitor program</b>	
<b>Appendix 4</b> . . . . .	<b>147</b>
<b>Pin assignment of the connectors</b>	



# Getting acquainted with the 'Junior Computer'

The term 'Junior' may imply that this computer is only suitable for children or amateurs. This is certainly NOT the case. We set out to design a compact computer that would be inexpensive and simple to build, yet have the capabilities of much larger systems. Although small in size, the JC has plenty of programming power, which makes it ideal for use by amateurs and professionals alike. Also, of course, the system is fully expandable thereby allowing the user to add more 'bits and pieces' as required.

Many people regard computers as being highly complex devices and believe that their construction and operation should be left to the 'experts'. We, however, have a different opinion and set out to justify it by designing the Junior Computer.

In principle, a microcomputer is really quite simple. For its construction, little more than basic electrical know-how is required. It is simply a matter of putting little black boxes into the correct holes – anybody can do a jigsaw! The important aspect of any electronics system is what it can do, rather than how it does it. In the case of the microcomputer it is the 'instruction set' that tells us this. The instruction set is the list of various commands and directions (given by the programmer) that the microprocessor 'understands'. So the challenge here lies not so much in the electronics involved, as in learning how to use the instruction set to tell the microcomputer what you want it to do. It's like driving a car, you don't have to know what's going on under the bonnet to be able to operate it. The question here is how to tell the computer what you want (in language it understands) and then to interpret the 'answer'.

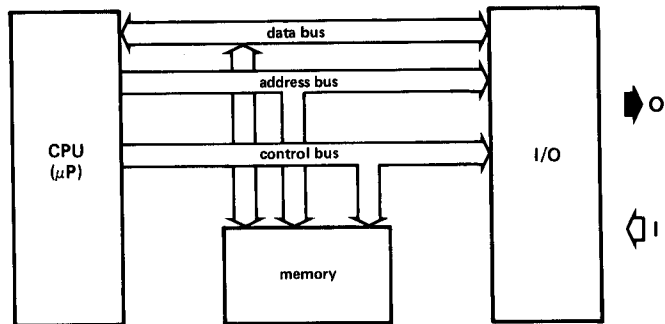


## How it works

Although having just stated that we don't need to know what is going on inside the computer to be able to use it, a brief description of its operation will help to understand what follows.

As can be seen from the block diagram in figure 1, a (micro)computer consists of three basic sections. These are the central processing unit (CPU), the input/output (I/O) section and the memory section. Information is transferred between the three 'blocks' by way of three groups of lines called *buses*, the *address bus*, *data bus* and *control bus*. A bus is quite simply a common line or collection of lines that are connected to more than one device.

A computer works with information, or data, which is in a form it can understand (specifically digital pulses). As its name suggests, the data bus carries this information to or from the various sections of the computer. The data bus consists of eight conductors and is therefore capable of transferring eight *bits* of data at a time. A bit is one piece of digital information or Binary digit. A group of eight bits is commonly called a *byte*. The largest computer in the universe is totally useless unless it is able to communicate with the outside world. This is where the *input/output*



80915 - 1-1

**Figure 1. The basic block diagram of a computer consists of three blocks and three buses. The latter provide the connections between the blocks.**

(I/O) section comes into the picture. For humans to be able to understand what the computer has to say, and vice-versa, some form of translation medium has to be incorporated. This is usually carried out by means of a keyboard and video terminal, or some other form of display. However, this is not to say that communication is limited to these.

On to the *memory*. The memory is simply a store where the computer holds all the relevant information (instructions, data etc.) required for it to perform a particular task. There is no such thing as an 'intelligent' computer (not yet anyway – as far as we know!). A computer has to be told explicitly what to do and in what order. Data is stored in individual compartments in the memory. These compartments are usually referred to as *locations* and each has its own (unique) *address*. Via the address bus,

the computer can pinpoint the exact memory location, and therefore the data, that is required. The address bus is also used to select the various input or output devices needed by a particular program.

Last, but by no means least, is the control bus. The control bus regulates various internal functions as well as telling the data bus which way to allow data to flow, whether to transfer data into or away from the CPU.

### A little more technical

After looking at the block diagram of a 'basic' microcomputer, we move on to that of the Junior Computer (figure 2). First, the three buses. The address bus is formed by sixteen lines and is independent of the other two buses (the data bus and the control bus). With 16 lines, the CPU is capable of addressing up to  $2^{16}$  or 65,536 different memory locations. Thus, more than sixty-five thousand (or '65 k') different pieces of data are at the computer's 'fingertips' (provided, of course, this amount of memory is available).

The data bus consists of eight lines, but is bi-directional. This means that information can be moved in two directions, to or from the microprocessor. Of course, data can only be transferred in one direction at a time. The direction of data transfer is determined by the control bus. If the computer is told to *read* information then the control bus allows data to be transferred from the memory (or any other source) to the CPU. Conversely, if it is told to *write* information, the control bus allows data transfer from the CPU to the memory or any other device. The control bus does this via bi-directional data bus buffers which, according to control bus signals, allow data to pass in the proper direction only.

### Memory

Memory comes in two main types, which is why there are two memory blocks in figure 2. One is RAM and the other is ROM. There are also two types of data: permanent (like a system *monitor* program) and temporary (most of the programs entered by the user). Permanent data therefore will only ever be 'read', whereas temporary data has to be both 'read' and 'written'. When talking about memory, the terms 'reading' and 'writing' refer to the act of seeing what is in memory and entering data into memory respectively. Memory that can only be read is called ROM or Read Only Memory. Where a system monitor program is used it is invariably stored in ROM. Memory that can be written into as well as read from is called RAM, for Random Access Memory. RAM is used to store such things as intermediate results and programs which are under development. Random access memory is therefore often called work memory. The signal that controls the bi-directional data bus buffer mentioned earlier, also tells the memory whether it is going to be read from or written into, hence the term, READ/WRITE.

It seems reasonable to point out at this time that when information is read from memory, that information is not lost. Similarly, the act of reading this page does not remove the text. The information is read and transferred to the brain (CPU) and further processed. RAMs do however forget when

their power supply is cut off. When the information contained in RAM is to be saved for extended periods of time it is usually transferred to a more permanent form of data storage, cassette tape or floppy disc for example. This is more convenient and safer than leaving it in RAM. If the mains should drop out, all information in mains powered RAM would be lost.

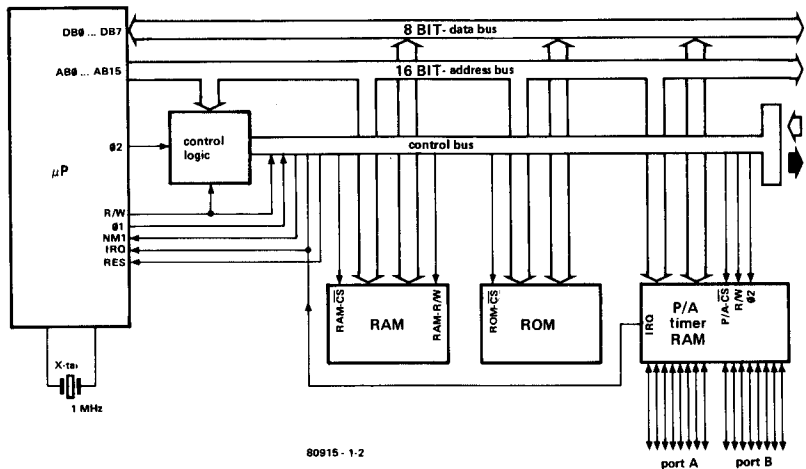


Figure 2. A detailed version of the block diagram in figure 1, this time with a specific computer in mind: the Junior Computer.

## I/O (the translator)

The block marked I/O maintains contact between the computer and the outside world. In figure 2 it is called PIA or Peripheral Interface Adapter. Just as with RAMs, bi-directional data transfer is necessary. The 8 bit data bus is passed out as two bundles of eight conductors each, through port A and port B. The active port is determined by the address bus. Each individual line can operate independently of the other 15 at any given moment, as an input or as an output.

Data can also be held at the PIA for short periods of time (notice that the RAM is in the same block as the PIA in figure 2). This facility can be used when the CPU has to do something else at the same time as data is being transferred through one of the ports. Information from either direction may be stored in this memory, but only from one direction at a time. The address bus informs the PIA which port, which direction, and whether or not to hold out-going or in-coming information.

There are also three buses that go to the outside (the three arrows pointing right in figure 2) but these are for future expansion of the system rather than communication with the outside world. As far as the JC is concerned, the outside world is everything beyond the keyboard and display.

## The CPU: the centre of activity

The Central Processing Unit is the 'heart' of the microcomputer system. Briefly, the function of the CPU is to control the operation of all the other units and to process data. The CPU contains a number of registers which are used to temporarily store address, data and instruction information for decoding and manipulation purposes, etc.

The CPU also contains the *program counter* which simply counts the steps in the program. Its output can be fed onto the address bus in order to have access to the memory for the retrieval of program instructions. To determine the next successive address in a particular program, the program counter and the instructions already executed are analysed. Exactly how all this is carried out, however, is outside the scope of this chapter.

The microprocessor has a sort of built-in 'pacemaker' or clock, which is used to generate the timing pulses around which all operation is based. The two signals produced by this clock generator,  $\emptyset 1$  and  $\emptyset 2$ , are  $180^\circ$  out of phase with each other. Without them the complete system would be useless, for they provide the 'heart-beat'.

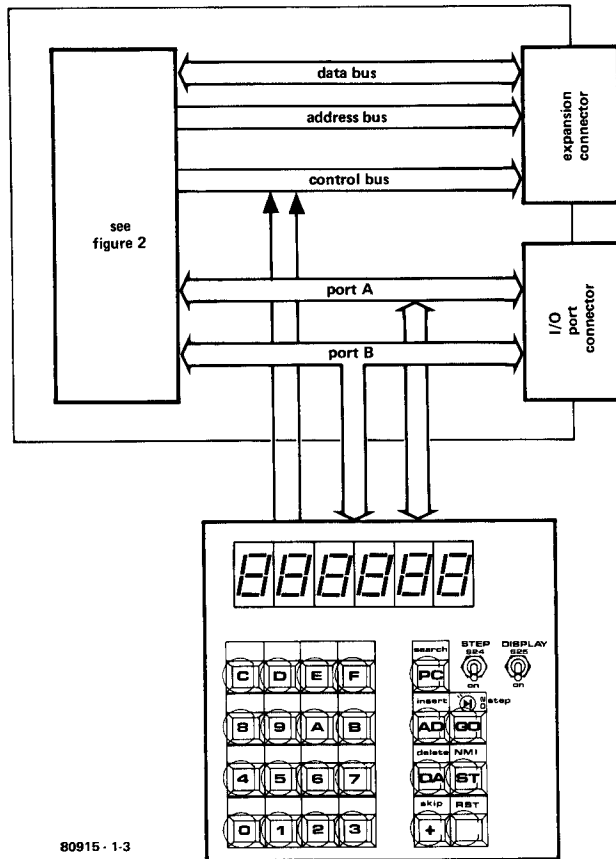
There are three other signals shown in figure 2, namely RES, IRQ and NMI. RES is the reset signal and is virtually self-explanatory. This signal tells the Junior Computer to go to the 'start' condition. The other two, IRQ (Interrupt ReQuest) and NMI (Non-Maskable Interrupt), are used to modify or step through the program while it is being run. Information then comes from the outside to tell the computer what to do next. This feature can be useful when the computer is used with a relatively slow device (a human for instance). The computer can manage over half a million operations per second whereas the human may only be able to manage 3 or 4 during the same period. Once the interrupt is finished the computer will continue with the main program from where it left off. In the event that both of the interrupt functions are used at the same time, they are given a priority which is determined by the program. It should be noted that the interrupt request can be controlled by the program whereas, as its name implies, the non-maskable interrupt cannot.

A programmable timer can also be seen in the PIA block in figure 2. More attention will be given to this in chapter 5 (Book 2).

## Peripherals

The keyboard and display complete the package. They are shown in figure 3. The keyboard consists of 23 key-switches and 2 toggle switches. Sixteen of these switches are used for entering information (in hexadecimal form) into the computer. The remaining keys are assigned various control functions. The display consists of six seven-segment LEDs and shows address and data information, again in hexadecimal form.

The keyboard and display are connected to the computer via ports A and B. Port A is designed for bi-directional data transfer whereas port B is uni-directional. There are two signals from the keyboard which are placed on the control bus, namely, RES and NMI. These belong to the key-switches RST and ST respectively. More will be said about these when discussing operation. The sixteen lines from ports A and B are also connected to a 31 pin connector for future expansion.



80915 · 1-3

**Figure 3.** A further development of the diagram in figure 2: communication with the outside world is now possible with the aid of a keyboard to input (I = Input in I/O) data and a display showing six hexadecimal figures to output (O = Output in I/O) data.

The address, data and control buses are accessible via a 64 pin expansion connector. The reason for these sockets is quite simple. Looking at it realistically, the hexadecimal keyboard as the standard input source, and the hexadecimal display as the output indication, are the simplest and least expensive methods of interacting with the computer. There are also many other I/O possibilities, all fine and dandy, but they all require more money. Most beginners do not want to part with (or have) the kind of outlay required for more sophisticated input/output devices. Beginners, however, have a habit of becoming experts so the JC is designed to grow into an expert's computer.

The 64 pin expansion connector may be used to expand the memory

capacity of the system so that longer and more complex programs can be run. This is practically a necessity for more 'grown-up' computers (you never seem to have enough memory). Bearing this in mind, the expansion bus was designed to be compatible with the Elektor SC/MP system bus. Now for a more detailed look at the Junior Computer's electronics, the so-called hardware. It can be said that the hardware is the 'flesh and blood' of the computer and that the programs form its 'personality'.

### Circuit diagram

The circuit diagram of the Junior Computer is shown in figure 4. The CPU (IC1) is a 6502 microprocessor. Readers unfamiliar with the various types can be assured that the 6502 is a fast, high quality device. It has a 'powerful' instruction set with a great variety of (useful) programming possibilities.

The microprocessor needs something to 'keep the blood flowing': a clock generator. This is constructed using N1, R1, C1, D1 and a one megahertz (1 MHz) crystal. Two clock signals are generated,  $\emptyset 1$  and  $\emptyset 2$ , for the address bus and data bus respectively. The address bus consists of lines  $A\emptyset \dots A15$  while the data bus consists of lines  $D\emptyset \dots D7$ .

The electrical signals on the address and data buses are coded digital information. What is this code? Imagine a numbering system with only *two* numbers  $\emptyset$  and 1 as opposed to the normal ten:  $\emptyset \dots 9$ . The numbers nought and one are represented by ' $\emptyset$ ' and '1' respectively (no change), but the number two is shown as '1 $\emptyset$ '. Three is represented by '11', four by '1 $\emptyset\emptyset$ ' etc. This is shown below for the numbers  $\emptyset \dots 15$ . Note:  $\emptyset$  is used for 'zero' to avoid confusion with the letter O.

decimal number	binary equivalent
$\emptyset$	$\emptyset\emptyset\emptyset\emptyset$
1	$\emptyset\emptyset\emptyset 1$
2	$\emptyset\emptyset 1\emptyset$
3	$\emptyset\emptyset 11$
4	$\emptyset 1\emptyset\emptyset$
5	$\emptyset 1\emptyset 1$
6	$\emptyset 11\emptyset$
7	$\emptyset 111$
8	$1\emptyset\emptyset\emptyset$
9	$1\emptyset\emptyset 1$
1 $\emptyset$	$1\emptyset 1\emptyset$
11	$1\emptyset 11$
12	$11\emptyset\emptyset$
13	$11\emptyset 1$
14	$111\emptyset$
15	$1111$

As can be seen, the limit for 4 lines of binary data is reached with only sixteen combinations, whereas with a normal base ten count ten thousand combinations are possible ( $\emptyset \dots 9999$ ). As was said earlier, the address bus has sixteen lines or  $2^{16}$  (65,536) different combinations. Therefore the

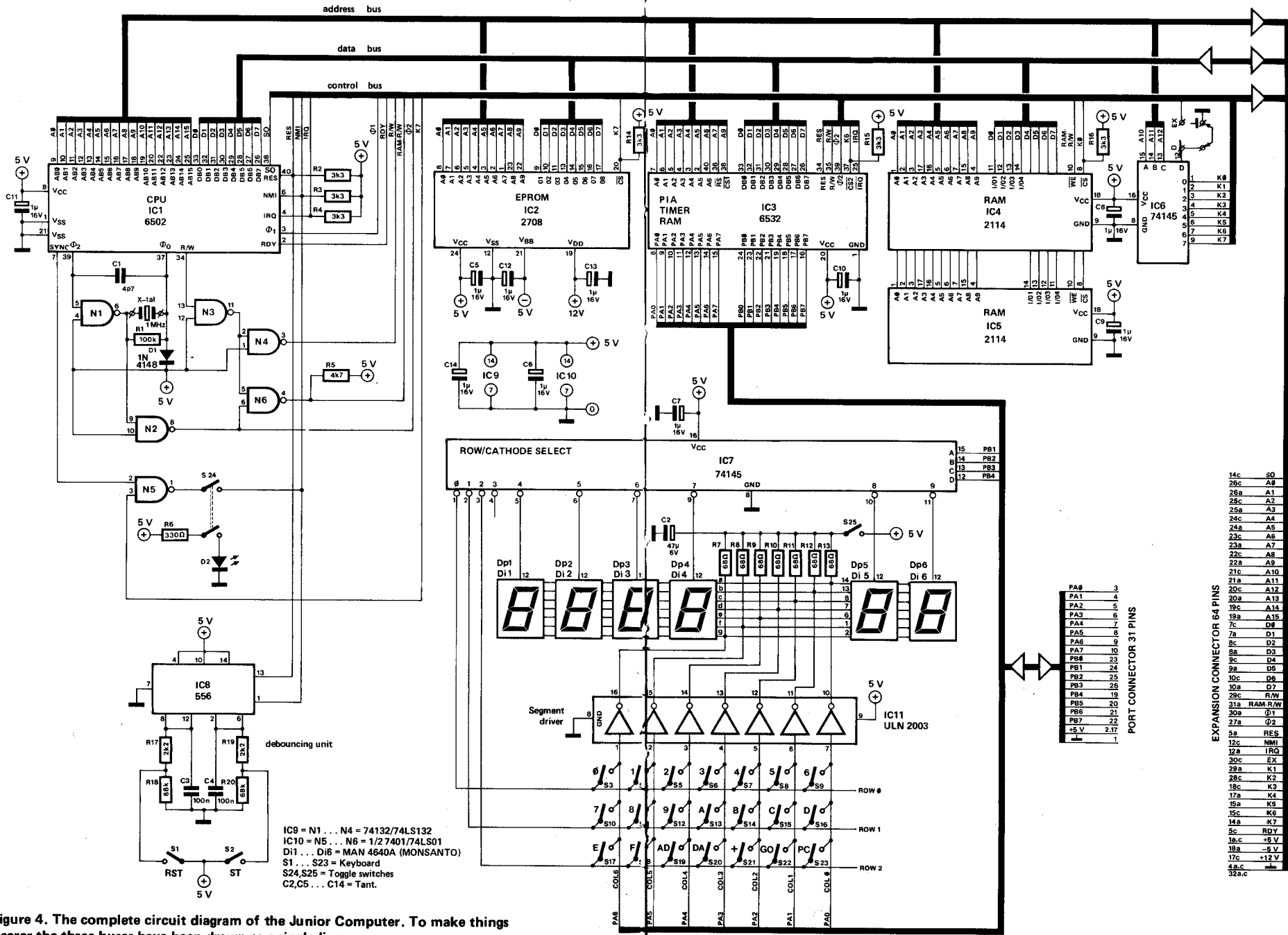


Figure 4. The complete circuit diagram of the Junior Computer. To make things clearer the three buses have been drawn as a single line.

limitation imposed by working with a number system with two as its base does not really pose a problem.

### Memory organisation

You will soon become familiar with the memory organisation of the Junior Computer once you gain further insight into the way in which memory operates in a computer system. In figure 4 the memory consists of two blocks of RAM and one block of EPROM (Erasable Programmable Read Only Memory). The EPROM (IC2) is for storage of permanent data (the system monitor program) and the RAMs (IC4 and IC5) are for operational use. Data transfer is accomplished in 8-bit blocks, or one full byte at a time. The EPROM has the capacity to store 1024 of these 8-bit groups. In computer terminology this is a 1 k byte EPROM (k meaning kilo or thousand). The various bytes have to be available on request, thus there are ten address lines connected to it giving  $2^{10}$  or 1024 different combinations. Strange coincidence that this should be the same number as the capacity of the memory!

The RAMs have a capacity of 1024 half-bytes, but, as there are two of these, their total capacity is 1024 full bytes (the same as the EPROM). The data bus lines are arranged so that the first half (four bits) of the data is in IC4 and the second half is in IC5.

The memory also receives directions from the control bus, selection signals for instance. The EPROM and the RAMs are connected to the same address lines, so how does the right one operate at the correct time? Both types of devices have a Chip Select ( $\overline{CS}$ ) input. When this input is taken high (+5 volts or logic '1') the memory is disabled. Conversely, if the input is taken low (zero volts, ground or logic '0') the memory is able to operate. These  $\overline{CS}$  signals are derived from the address decoder, IC6, and passed to the EPROM as K7 and to the RAMs as K0.

Ten address lines are utilised by the memory, leaving six. As  $2^6 = 64$ , this means that if these are used as chip select lines, up to 64 blocks of 1 k each are addressable. This amounts to 64 k – not the 65 k mentioned earlier, due to the fact that all the numbers have been rounded off. The total still comes to  $2^{16} = 65,536$ .

With 64 'block address lines' available, all memory addressing is done with the first ten address lines. In the basic JC only the first eight memory blocks are decoded, namely K0 . . . K7 from the address decoder. Three address lines (A10 . . . A12) are fed to IC6 which promptly decodes them into the eight lines required. The table below shows this to advantage.

#### DATA CASE

A15 . . . A13	A12	A11	A10	A9 . . . A0	active	memory block
X	0	0	0	X	K0	1 k RAM (IC4, IC5)
X	0	0	1	X	K1	1 k external RAM, ROM
X	0	1	1	X	K3	1 k external RAM, ROM
X	1	0	0	X	K4	1 k external RAM, ROM
X	1	0	1	X	K5	1 k external RAM, ROM
X	1	1	0	X	K6	RAM in PIA (IC3)
X	1	1	1	X	K7	1 k EPROM (IC2)



'X' means 'don't care'. In other words, its state may be either a '1' or a '0' to get the listed result.

Of the 64 k then, 8 k are directly accessible. To address more memory, the address decoder must be expanded.

In addition to the chip select signal, the RAMs require another signal that tells the memory whether information is going to be read from or written into RAM. This is where the  $R/\overline{W}$  line comes into the picture. If this line is taken high the memory is going to be read from, and if it is taken low information can be written into it. This signal comes from the NAND gate N6 and is a mixture of the  $\emptyset 2$  clock pulse and the  $R/\overline{W}$  signal of the microprocessor. This guarantees that no data can be transferred while the data bus is not stabilised.

### Control bus

A number of control signals have already been covered, the rest now follow. For correct operation both the microprocessor (IC1) and the peripheral interface adapter (IC3) have to be initialised. This is achieved via the reset signal RES. The reset line is normally held high by a pull-up resistor, R2. A reset is generated when the keyboard switch RST is pressed. Operation of the RST key triggers the timer in one half of IC8 which is used to suppress any contact bounce this key might produce. The output of the timer is connected directly to the reset line.

There are two ways in which a program being run can be interrupted by means of the non-maskable interrupt (NMI). The first one is provided by the STOP key S2. You will notice that this key uses the other half of IC8 for contact bounce suppression. The second is provided by the STEP switch S24. When this is in the 'on' position and the output of N5 goes from high to low, the NMI line (normally held high via pull-up resistor R3) is also taken low. This feature is important when wanting to step through the program 'byte by byte'. It should be noted that as K7 is connected to one of the inputs of N5, the output of N5 will always be high when the EPROM is selected. As N5 is a NAND gate, if one of its inputs is low the output will be high. All this means, effectively, that you cannot step through the monitor program held in IC2.

The program can also be interrupted if the IRQ (interrupt request) connection between IC1 and IC3 is taken low. This line is again held high normally, this time by pull-up resistor R4. Not only can the IRQ facility be used manually, but it can also be used via the timer in the PIA. When utilised by the program in this way it is called a 'software interrupt'. The NMI and IRQ lines are also accessible on the 64 pin expansion connector. Also present on the control bus are the two clock signals  $\emptyset 1$  and  $\emptyset 2$  which control the PIA and RAM  $R/\overline{W}$  signals. As mentioned previously, these determine the direction of data transfer. Finally, the signals RDY and SO, neither of which is used in the basic JC, are for future expansion with dynamic RAMs, and the line EX (see IC6) is important when expansion of the address decoding becomes necessary.

## PIA

The PIA is capable of transferring data in two directions via ports A and B. Each port has its own I/O register and eight-bit data direction register. The information contained in the data direction register determines which of the individual port lines are to be used as an input or output. If a particular bit in this register is a '1' the PIA is instructed to activate the associated output, whereas if the bit is a '0' an input is activated. The contents of the data direction registers is determined by the program or software. Actual data transfer is carried out via the R/W signal. As with the RAMs, when this line is high a read operation is specified and data will be transferred from the PIA to the CPU. When the R/W line is low, a write operation is specified and data will be transferred from the CPU to the PIA.

There is also a limited amount of RAM in the PIA, 128 bytes to be exact. Together with the 1024 bytes of IC4 and IC5 this gives a total of 1152 bytes (more than enough for the budding amateur!). Address lines A0 . . . A6 are used to gain access to the 128 bytes of RAM in the PIA. Address line A7 is connected to the RAM select input ( $\overline{RS}$ ) of the PIA. In other words, when A7 is high the RAM is enabled and the CPU has direct access to it. When A7 is low the PIA RAM is disabled. Reading and writing to this RAM is, of course, controlled by the R/W signal. The other address line (A9) connected to the PIA controls the selection of the I/O ports and the timer. When this line is high, address lines A0 . . . A6 determine which of the various functions of the PIA are to be used. For instance, port A or B enabled, direction of data transfer, access to the timer etc. Note that when A9 is high the PIA RAM cannot be accessed. The remaining signal connected to the PIA is K6 from the address decoder. The PIA is disabled (regardless of the condition of the other inputs) when K6 is high. The PIA is such a versatile and multi-functional device that we could quite easily write a whole book about it, but as far as this chapter is concerned, enough is enough.

### Links with the outside world

The keyboard and display comprise the Junior Computer's communication links (or peripherals) with the outside world. These are connected to port A by seven lines, to port B by four lines and to the control bus by two lines. The last two are the previously mentioned signals, reset and stop (RST and ST). Switch S24 has been mentioned before with relation to the NMI signal. It is also used to select either normal operation or step-by-step program development.

The other keys shown in figure 4 (S3 . . . S23) are arranged in a matrix of three rows and seven columns. Sixteen of these keys are used for entering data into the computer in hexadecimal code (to be dealt with in chapter 2). The word data has a broad meaning here as it also includes address information. The remaining five keys have been assigned various control functions. These will be discussed in depth in chapter 3.

Information going to the display, and data from the keyboard is transferred via seven lines of port A. In this instance it can be seen why bi-directional data transfer is very useful. The information on the displays is controlled by the software in the monitor program, which also ensures

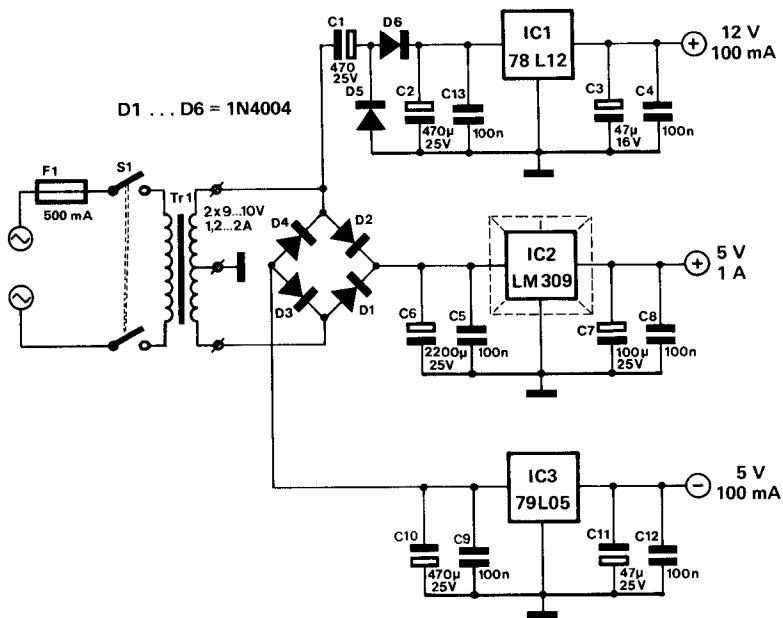
that key function signals are recognised. A BCD-to-decimal decoder, IC7, decodes the information presented on the four lines PB1 . . . PB4 from port B. This decoded information is used to multiplex the displays and check the state of the rows of keys to see which one, if any, is being depressed. The first three outputs of IC7 are used to scan the keyboard. If a key is depressed while one of these outputs is high the code for that particular key is passed to the CPU via port A. The third output of IC7 is not used — it isn't needed. When one of the remaining six outputs of IC7 is high, the appropriate display is turned on, and the coded segment information from the CPU is transferred via port A to the segment drivers contained in IC11 and from there to the particular display concerned. This process of successive selection of one device at a time is an elegant method of reducing the number of parts required for a particular application. Saving a few parts however costs a little software. The cost of the extra memory required, on the other hand, is far less than the cost of all the parts it would take to duplicate the functions it performs. The displays can be used in two different modes. Usually, the four left hand displays (Dp1 . . . Dp4) will indicate an address and the other two (Dp5 and Dp6) will show the data in the address location concerned. As a second possibility, Dp1 and Dp2 can show the hexadecimal code of an instruction (op-code) while the remainder show the address of the data corresponding to this instruction. The latter mode makes program entry much simpler. The only other switch to be mentioned is the display turn-off switch S25. This comes in very handy for saving power if the JC is to be used with a terminal or other external device.

## Power supply

Even the most sophisticated of computers obviously needs a power supply. Three supply voltages are required by the Junior Computer and a suitable circuit is shown in figure 5. The three voltage levels are +5 volts (for all the ICs and displays), -5 volts and +12 volts (for the EPROM) and these are supplied by the voltage regulators IC1 . . . IC3. Each regulator IC has its own set of capacitors, C2 . . . C13, to ensure the necessary decoupling. Now that we have discussed the basic theory and operation of the Junior Computer we can start putting it all together.

## Construction

Where do we start? To make assembly as simple and as trouble-free as possible, this section should be read with extra care and attention. Construction is accomplished in three basic steps. First, the components are mounted on the printed circuit boards. As the main printed circuit board is double sided, we do not advise readers to make it themselves. All boards are directly available from Elektor (EPS Nos. 80089-1, 80089-2, 80089-3). Once the boards are complete and correctly interconnected, it is time to test them. This is the second and (hopefully, with good quality parts) fastest step. The third and final step is the mechanical assembly: putting the Junior Computer in its case.



80915 · 1-5

Figure 5. The power supply of the Junior Computer provides three stable voltages.

## Main board

The Junior Computer is very simple to construct as it is a *single board* computer. That is to say, all the electronic parts are mounted on the same board. Now sceptics may say: 'there are *three* boards, how can you possibly call it a single board computer?'. The answer to that is quite simple. One of the boards is a power supply and so we don't count it! Two remain, the main board and the display board. The latter is a small board carrying the displays and is attached to the main board. It could easily have been designed as part of the main board but that would have made it bigger and this way the displays can be tilted at an angle of  $45^\circ$ , making for better readability.

The main board is double sided, that is to say, there are copper tracks on both sides of the board and, in this case anyway, components on both sides too. Certain copper tracks on one side of the board are connected to copper tracks on the other side. This is possible because the board has 'plated through' holes. Before any assembly is started it is a good idea to check all the plated through holes (there are over six hundred of them!). This can be done with the aid of an ohmmeter, or (if you don't possess an ohmmeter) by means of the inexpensive method shown in figure 14a. The low voltage secondary of a bell transformer is used in series with a doorbell to indicate continuity. Using two pieces of wire, one on the top side of the

hole and the other on the opposite side, if the bell sounds the plating is good. This method has one advantage over the ohmmeter — you don't have to look at the scale every time to confirm a good connection. If a hole should happen to be found to have defective plating, it can be remedied by soldering a wire link in its place or by simply using the lead of the component to be soldered in that hole. All this may take a little time but it could save a lot of frustration later.

The two sides of the main board are, of course, very different. The component overlay for the top side is shown in figure 6 and that for the bottom side in figure 7. Figures 8 and 9 show the track layout for the top and bottom sides respectively. The keyboard and display board are mounted on the top side of the board and all the rest of the components, excluding the power supply, are mounted on the bottom side.

### Mounting the components

Now assembly can begin in earnest. The soldering iron used should have a 'pencil' tip and be in the 20-30 Watt range. The components should be mounted in the following order (see figure 7 and the full parts list for the main board):

1. The resistors R1 . . . R20 are the first components to be installed. After soldering, the excess lead length should be cut off as close to the surface of the printed circuit board as possible. This is a logical safety measure, to prevent shorts between adjacent connections. For those readers who are unfamiliar with the resistor colour code the corresponding colours for the values used are listed below.

100 k: brown-black-yellow (gold)

3k3: orange-orange-red (gold)

4k7: yellow-violet-red (gold)

330  $\Omega$ : orange-orange-brown (gold)

68  $\Omega$ : blue-grey-black (gold)

2k2: red-red-red (gold)

68 k: blue-grey-orange (gold)

The first ring is the ring closest to one of the ends. The fourth ring indicates the tolerance (how much above or below the actual value it may be) of the resistor. Most resistors will have a gold ring which indicates a tolerance of  $\pm 5\%$ . It is possible (though unlikely) to get a 1% (brown) or a 2% (red) resistor. If one of your resistors has a fourth ring of silver (10%) it should not be used. Another resistor will have to be found with a tolerance of  $\pm 5\%$  or less to take its place.

2. The next component to be installed is the diode D1. Care must be taken to connect the diode the right way round. Its polarity is usually given by a ring around one end of the diode case. This ring denotes the cathode (the cathode is the thick line at one end of the triangle in the circuit diagram). In the unlikely event that the diode does not have a ring at all, or it has a ring exactly in the centre, the only positive way to determine its polarity is with an ohmmeter. When testing with an ohmmeter, it will be found that the diode exhibits a rather high resistance in one direction and a much lower resistance in the other. This only tells you that the diode is a good one, it does not tell you its

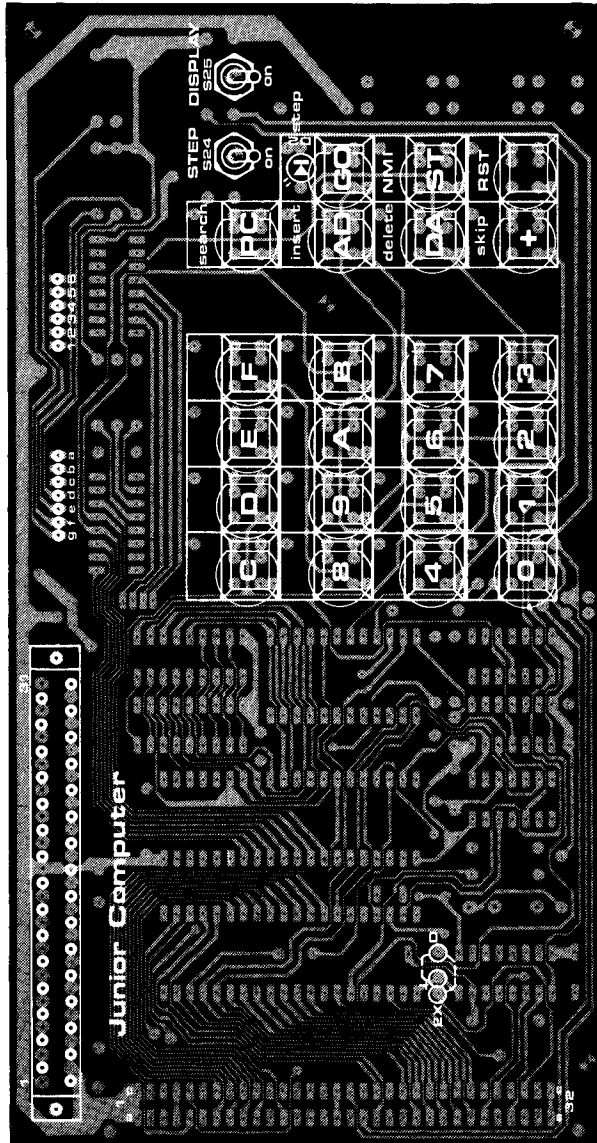


Figure 6. The component overlay of the keyboard section of the Junior Computer (EPS 80089-1).

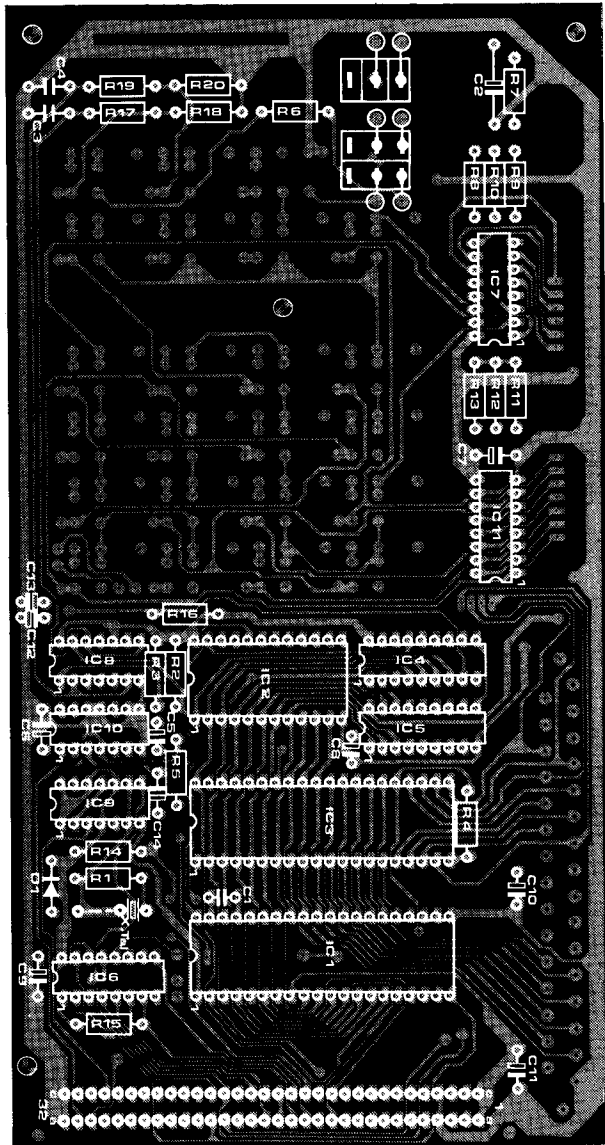
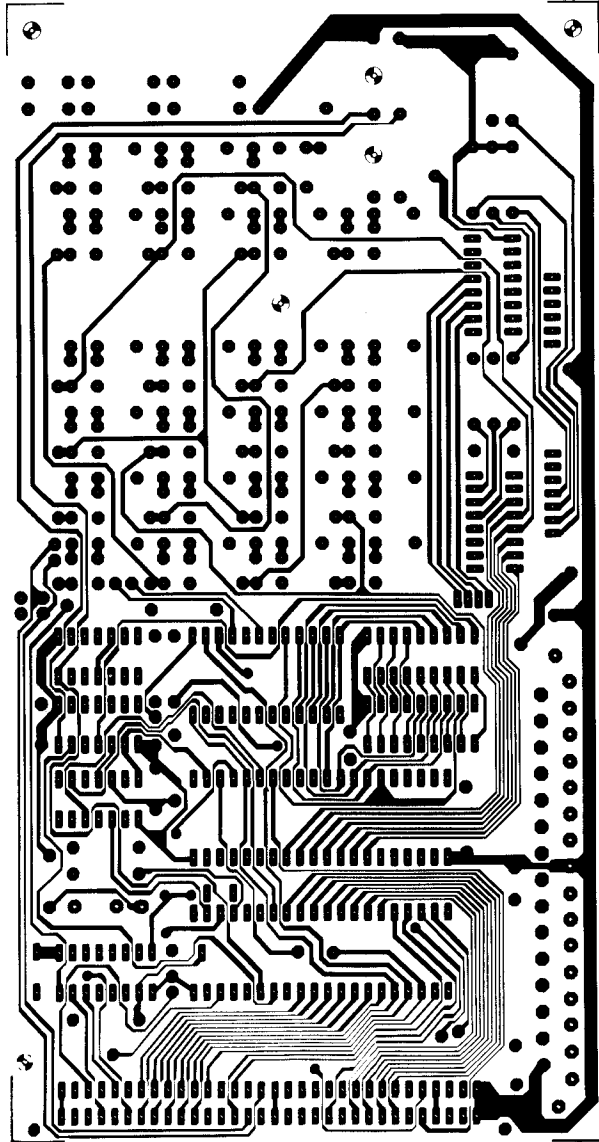


Figure 7. The overlay of the component side of the main printed circuit board.



**Figure 8. The track pattern of the upper side of the main board.**



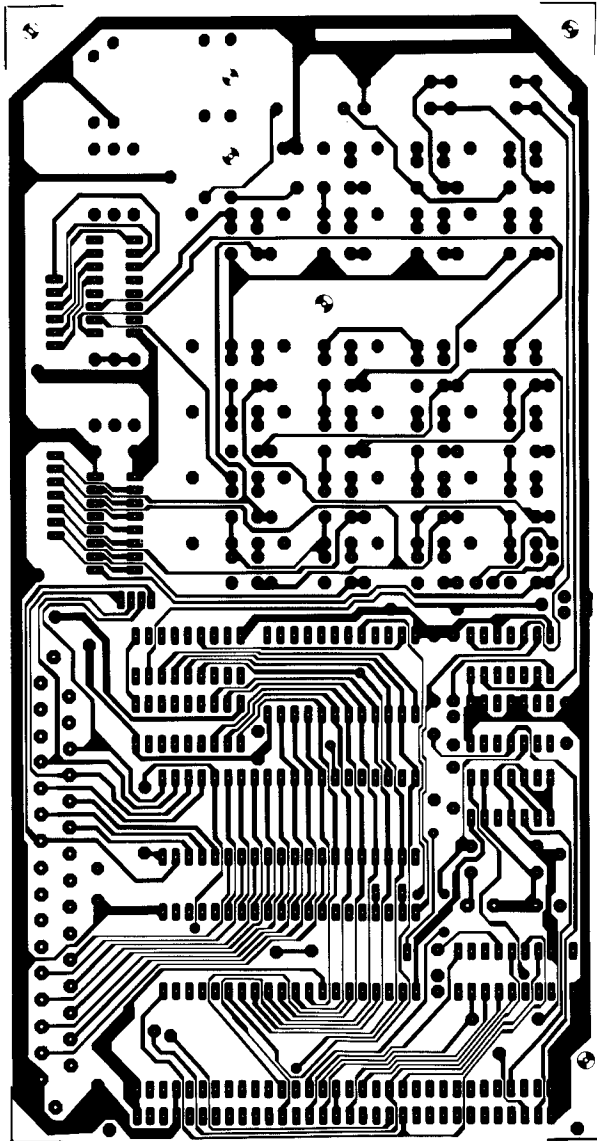


Figure 9. The track pattern of the lower side of the main board.

## Parts list for the main circuit board of the Junior Computer.

The complete circuit diagram is shown in figure 4, the two component overlays are given in figures 6 and 7 and the two track patterns in figures 8 and 9.

### Resistors:

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

### Capacitors:

C1 = 10 p ceramic  
C2 = 47  $\mu$ /6 V tantalum  
C3,C4 = 100 n MKH  
C5 . . . C14 = 1  $\mu$ /35 V tantalum

### Semiconductors:

IC1 = 6502 (Rockwell)  
IC2 = 2708  
IC3 = 6532 (Rockwell)  
IC4,IC5 = 2114

IC6,IC7 = 74145  
IC8 = 556  
IC9 = 74LS00, 7400, 74LS132  
IC10 = 74LS01, 7401  
IC11 = ULN2003 (Sprague)  
D1 = 1N4148

### Miscellaneous:

S1 . . . S21,S23 = digitast (Shadow)  
S22 = digitast + LED  
S24 = double pole switch  
S25 = single pole switch  
connector 64-pole male perpendicular solder to DIN 41612  
connector 31-pole female perpendicular solder to DIN 41617  
1 MHz-crystal  
1 24-pin IC sockets  
2 40-pin IC sockets  
1 printed circuit board EPS 80089-1

- polarity. The next step is to measure a diode with a known polarity and note which probe is connected to the cathode when the diode conducts. If no known diode is available then *usually* the red probe of the ohmmeter is the cathode when the diode conducts.
3. The capacitors C1, C3 and C4 are now installed and their excess leads cut off.
  4. The electrolytic capacitors C2 and C5 . . . C14 are then mounted. Normally with capacitors and resistors you can install them regardless of their polarity. This is not the case with electrolytics. These have a positive and a negative side. In the circuit diagram the negative side is shown as a shaded (■) rectangle and the positive side as an empty (□) rectangle. On electrolytics that do not have a plus sign, the positive end is identified in one of two fashions. Either one end has an indented ring or the positive lead has a red mark.
  5. The next items to be soldered in are the sockets for the ICs. We recommend that sockets be used for *all* the ICs but, as these are an added expense, it is possible to mount the smaller ones directly onto the board. Sockets are essential for IC1 . . . IC3. There are of course two

ways to mount an IC (if you guessed a right way and a wrong way you are not far out!). On the parts layout (figure 7) there is a 'notch' at one end of every IC. If you look at the drawing of an IC with the notch at the top, then pin one is at the top left hand side. This is also indicated on the layout. The numbering of the pins goes down the left hand side and continues up the right hand side. So for a 14 pin IC the lower left is pin 7, the lower right is pin 8, and the top right is pin 14. As far as the IC itself is concerned, the 'notch' can take various forms. It could be (and usually is) a notch just like that in the parts overlay, but it could also be a 'dot' impressed in the body of the IC. This 'dot' denotes pin 1, and the IC should be installed with the dot on the same side as in the parts overlay. Not quite so important but very very useful is the fact that IC sockets also have pin 1 marked in some way or other — the socket can be the wrong way round as long as the IC inserted into it is the correct way round.

6. The 1 MHz crystal can be mounted directly onto the board, or a crystal holder can be used.
7. This step is optional. As it is not likely to be needed just yet, the 64 pin expansion connector can be installed at a later date. When mounting the connector be sure to 'screw it down tight' before soldering any of the pins.

If, however, you delay installation of the connector, provision must be made for the supply lines. These would otherwise use the expansion connector to come onto the main board. The power supply connections are:

+5 volts: pins 1a or 1c

ground (0 volts): pins 4a, 4c, 32a or 32c

−5 volts: pin 18a

+12 volts: pin 17c

The obvious solution is to use terminal pins as a temporary connection medium but care must be taken when de-soldering the pins, as the copper track is very thin in places and may lift off if excessive heat is applied.

That completes the mounting of components on the bottom side of the board, and now it is the turn of the keyboard side.

8. The only wire link on the board should be mounted next. It should be soldered between the points marked (L) and (D).
9. The two toggle switches (S24 and S25) can be installed next. These should be mounted so that the switch housings are on the underside of the board. The switches are connected to the main board by six short flexible insulated wire links. The positions for these are clearly marked on the underside of the board.
10. If required, the 31 pin connector can now be soldered into place.
11. The keyboard is next, along with D2. Care should be taken when installing the key switches, they should lie flush with the printed circuit board. D2 is mounted inside the 'GO' key. Remember, even though it is an LED, it is still a diode, and its polarity will have to be determined. The method described in part 2 can also be employed here.

That completes construction of the main board, or does it? It certainly

won't hurt to check all the ICs, diodes, and electrolytic capacitors for correct polarity, and every component for proper placement. Above all check your soldering for bridges between tracks, dry joints, etc. Soldering mistakes are still the biggest headaches on home-built electronic equipment.

### The display board

The component overlay and the track layout for the display printed circuit board (EPS 80089 - 2) are shown in figures 10 and 11 respectively. There are really only two steps involved in the construction of this board: mounting the six seven-segment displays and connecting the display board to the main board. The displays should not present any problems as there is only one way to mount them – they have an asymmetrical pin-out. The connection to the main board consists of thirteen conductors: seven for

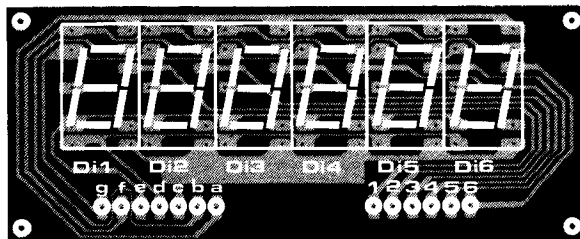


Figure 10. The component overlay of the display board (EPS 80089-2).

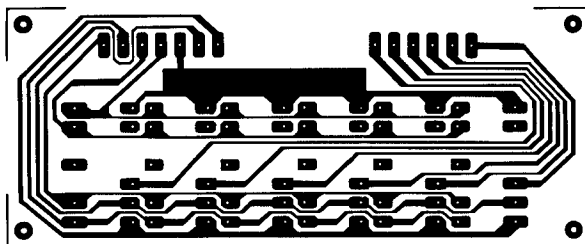


Figure 11. The track pattern of the display board.

#### Parts list for the display board of the Junior Computer.

The component overlay is shown in figure 10 and the track pattern in figure 11.

#### Semiconductors:

Di1 . . . Di6 = MAN 4640A  
common cathode (Monsanto)

#### Miscellaneous:

1 printed circuit board  
EPS 80089-2

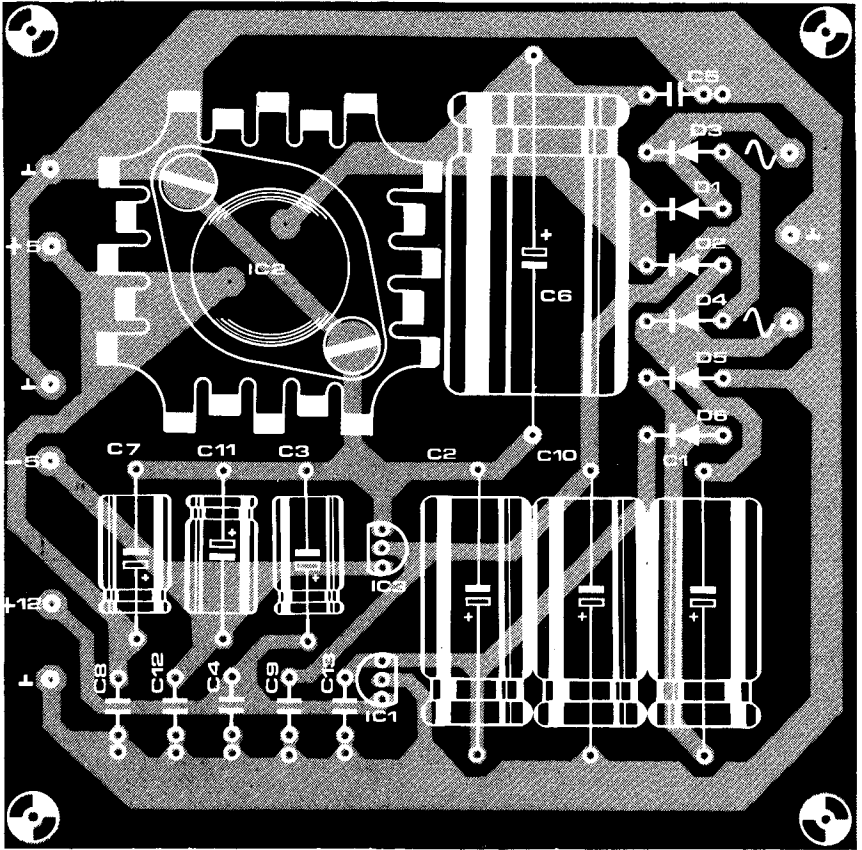


Figure 12. The component overlay of the power supply board (EPS 80089-3).

**Parts list for the supply board of the Junior Computer.**

The circuit diagram is shown in figure 5 and the board is given in figures 12 and 13.

**Capacitors:**

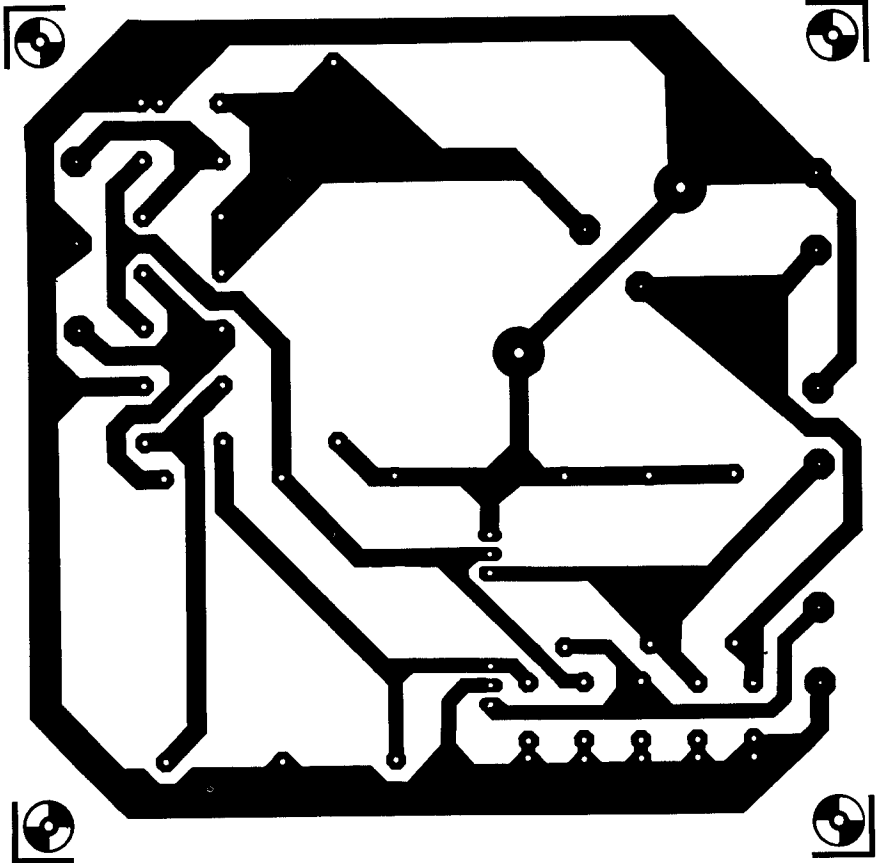
- C1, C2, C10 = 470  $\mu$ /25 V
- C3, C11 = 47  $\mu$ /25 V
- C4, C5, C8, C9, C12, C13 = 100 n MKH
- C6 = 2200  $\mu$ /25 V
- C7 = 100  $\mu$ /25 V

**Semiconductors:**

- IC1 = 78L12ACP (5%)
- IC2 = LM 309K
- IC3 = 79L05ACP (5%)
- D1 . . . D6 = 1N4004

**Miscellaneous:**

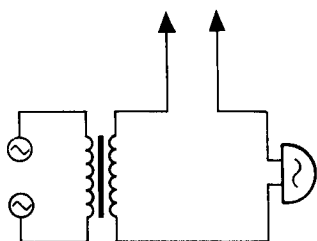
- Tr1 = transformer prim. 220 V  
sec. 2 x 9 . . . 10 V/1.2 . . . 2 A
- S1 = double pole switch
- F1 = fuse 500 mA, with fuse holder
- 1 printed circuit board EPS 80089-3
- 1 finned heat-sink for IC2



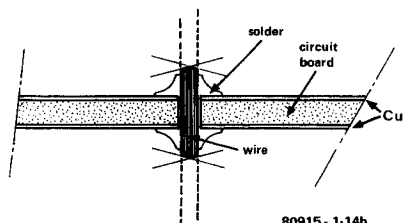
**Figure 13. The track pattern of the power supply board.**

the segment drives and six for the display drives. These connections are labelled on the main board above the keyboard assembly. The distance between the two boards should be about 5 mm. The thirteen wires should be gauge and  $1\frac{1}{2}$  to 2 cm long. They should be soldered onto the display board so that they protrude from the copper-clad side. It is advisable to check the component overlay as well as the printing on the main board when connecting the two boards to each other. Mount (but do not solder) the display board onto the main board. The display board should be tilted at an angle of approximately  $45^\circ$  relative to the main board. The wires can now be soldered and any excess lead length cut off.

Note: If it is more convenient, the display can be mounted at some remote location. So-called 'ribbon cable' is ideal for this. This is a flat set of conductors which are colour-coded for easy connection. It goes without saying that extreme care must be taken to ensure that the leads are not mixed up.



80915 - 1-14a



80915 - 1-14b

Figure 14. The electrical connections need not necessarily be tested with a multimeter (ohmmeter); it can also be done acoustically with the aid of an inexpensive bell and bell transformer. Note: this method may only be used if the components have not yet been mounted on the board. Figure b shows how to connect the two sides of a home made board (which has not got plated through holes) with a vertical wire link.

## Power supply board

The component overlay and track pattern for the power supply board are shown in figures 12 and 13 respectively. Like the display board, the construction of this board should not present any problems. Care should, of course, be taken to ensure that the diodes and electrolytic capacitors are mounted with the correct polarity. A heat sink is required for IC2 (the LM 309K).

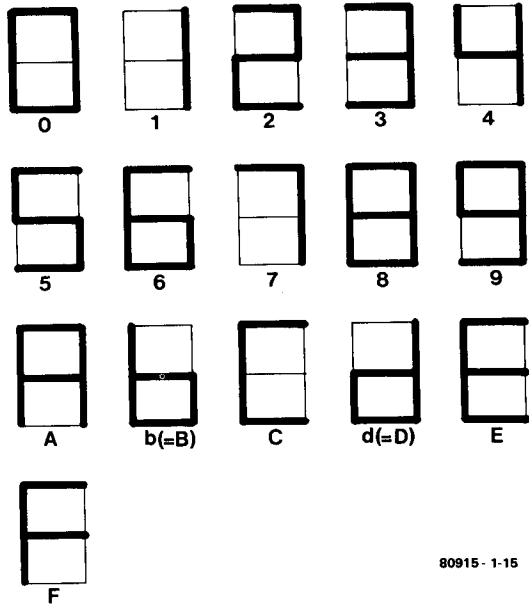
## Will it work?

The three boards are now complete. The next stage is to connect the transformer Tr1 to the supply board, and via switch S1 and fuse F1 to the mains. Since in the up-coming test phase operation is going to be temporary, it can be jury-rigged at this stage. This does not mean that it should stay that way.

Before starting the test procedure it is advisable to check all the components once again. It is better to be safe than sorry. Perhaps you are able to find someone else who is able to give the circuit a look-over (another set of eyes often finds things previously overlooked).

The first thing to check is, of course, the power supply board (do *not* connect it up to the main computer board yet). Plug it in. No puff of smoke? A good sign! Oh, you forgot to switch S1 on. Still no smoke? If the answer to this question is no, the power supply has just passed its first test. Measure (with a multimeter set for DC volts) the supply output voltages. The readings should be within 5% of the rated output. If the supply voltages are outside their allowable limits something is obviously amiss. This is very unlikely, however, owing to the quality of components and simple design of the circuit.

If the power supply checks out all right, the wires between it and the main board can be soldered into place. Make *absolutely* sure that the wires are



80915 - 1-15

**Figure 15.** When the RST key is depressed the display should show a (apparently) random combination of the above hexadecimal figures. This shows the Junior Computer is working properly.

connected correctly. Double check, triple check, you'll only have yourself to blame if it is wrong. Once you are sure everything is connected correctly set switches S24 (STEP) 'off' and S25 (DISPLAY) 'on'. You now apply power once again and . . . nothing happens. Don't panic – this is what is supposed to happen. Now press the reset switch (RST). If all is in order the displays should give a clear hexadecimal number. To understand what is going on you will, of course, have to learn about the hexadecimal code given in chapter 2. For the time being you only have to compare the readouts with figure 15. The displays should show a random combination of these expressions. Readers who have already digested chapters 2 and 3 will recognise this as a sort of conditional jump instruction. If the display has the above, the next section can be skipped and you can move on to the mechanical assembly.

### **If the unthinkable happens . . .**

We sincerely hope this section will not have to be read, but just in case something is wrong the most common faults and how to deal with them are listed below. The first thing to check is the power supply voltages. Although these have already been checked, there may be something on the main board which is causing problems.



- Solder shorts. Solder forming a bridge between adjacent tracks. They don't always have to be obvious either. Hair-line solder bridges can be very troublesome.
  - Bad solder joints. 'Dry joints'. They can happen to anyone and are typified by a 'shattered' surface and poor contact with the copper track. If one is found, touch the joint with the soldering iron and apply a little additional solder.
  - Bad IC socket connection. It can happen. Poor contact between the IC and the socket. A careful inspection may even reveal a pin bent underneath the IC, not going into the socket! It is rare, but possible to have dirty contacts in the socket itself. If pushing down on the IC causes correct operation, then this is almost certainly the case. A little alcohol, on a cotton-bud, brushed over the top of the socket (allowing some to run into the socket) will usually remedy this.
  - Bad tracks on home-made printed circuit boards. This could be caused by insufficient or too much time in the etchant or by hairs on the artwork. Problems with the through-board connections should have been eliminated if the board was tested as outlined earlier.
  - Incorrectly installed diodes, electrolytic capacitors or ICs. If ICs are in the wrong position or mounted the wrong way round this will, obviously, cause the JC to malfunction. Are the connections between the main board and the power supply board correct and good? Between the main board and the display board? Up to now nothing but 'normal' items have been discussed. Here are a few special hints:
    - Measure the voltage between pins 13 and 7 of IC8 (pin 7 is negative and pin 13 is positive). It should be +5 volts. Press the reset button (RST) on the keyboard, the voltage between those two pins should now be approximately 0.5 volts. If this is not the case the problem could involve one of the following parts: IC8 (the double timer), the pull-up resistor R2 or the reset switch itself.
    - If all checks out okay so far then measure the *resistance* between pin 12 of IC6 and ground (of course, whenever checking resistance the power should be turned off). It should be zero ohms. If not, then the wire link is in the wrong place.
    - The clock generator constitutes the heartbeat of the computer. With a dual-trace oscilloscope the signals (Ø1 and Ø2) on pins 30a and 27a of the expansion connector can be monitored. The earth of the scope should be connected to computer ground (pins 4a or 4c) and the A (or Y1) input to pin 30a and the B (or Y2) input to pin 27a. There should be two 'out of phase' signals on the screen. When one goes high, the other should go low and vice-versa. The peak-to-peak voltage should be somewhere between three to five volts. The scope should have no trouble keeping the signals steady. They should be stable. If any of the above is not true the culprits could be C1, IC9 or D1.
- Chances are that one of the above 'trouble-shooting' techniques will solve your problems; however, if you are still unable to get the Junior Computer up and running, you can call the Elektor technical staff on Monday afternoons for additional help (see latest magazine for details).

## The case

The case has three basic purposes: to protect the circuitry from the elements, to allow for convenient operation, and to make the computer look really smart.

There are two popular methods of building cases for projects: the 'cigar box' method and the buy-ready-made method. Ready-made small computer cases usually have a display panel. The display board of the Junior Computer can be mounted behind this once all the necessary holes have been made. A clearance hole for the keyboard and toggle switches will also have to be made in the upper surface of the case. The fuse holder and mains socket can be mounted at the rear.

The 'cigar box' builder will have to design the case to meet the above criteria. Keep in mind that in the future, various expansion boards will come along and the case will also have to accommodate these.

The boards are mounted in the case using 'stand-offs'. Be careful when installing the main board that the keyboard switches operate freely to avoid mis-entered information.

Now the fun can begin.

# The binary number system

## Counting on two fingers

The fact that homo sapiens has ten fingers is probably the reason why we tend to count things in tens. Our set of numbers has ten figures and we have a form of coding system that allows us to manipulate them. While the Junior Computer also works with and manipulates numbers it only has the equivalent of two fingers on which to count. This may seem to be a 'bit' of a disadvantage at first sight but this chapter aims to prove otherwise by delving deeper into the binary (base two) numbering system.

Take a 1, a 9, an 8, and another 1. Now write them down one after the other from left to right.

1981

We immediately associate this with a number. A year number for instance, or the price of an expensive article. Using the mathematical code learned at school, it can mean a lot more. The result could be 'next year' or it could represent a telephone number. The latter is at the same time a code which, along with an area code, determines the position of various relays and other switching equipment.

If the (telephone) number 1981 were fed into a computer it would look quite different:

11110111101

As mentioned in chapter 1, the circles with diagonal lines through them represent 'zeros'. The 'slash' is used to differentiate between a zero and the capital letter 'O'. Since there are only two figures in the computer's numbering system, they will appear far more often than in normal mathematics.

It should be pointed out at this stage that 11110111101 does not mean 11, 110, 111, 101. The number looks rather strange because there are only ones and zeros in it. With the normal numbering system of ten figures

there is only a 10% chance that any one number will come up in a particular position. The chance that the number 11, 110, 111, 101 would come up in normal arithmetic is very small – we worked it out to be about two millionths of one percent! Using scientific notation, the number of different possibilities that can be represented with the same amount of figures as those shown above is:

for a base ten (decimal) numbering system  
 $10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 = 10^{11}$   
 $= 100,000,000,000$

for a base two (binary) numbering system  
 $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^{11} = 2048$

Quite a difference.

### The structure of a number

Every number can be viewed as a sum of lesser amounts. There are many possibilities and the number of possibilities increase as the base goes up. A decimal number can be divided into units of ones, tens, hundreds, thousands and so on. It can then be further defined by making it unit multiplied by a number. For example,  $300 = 3 \times 100$ . For the number 1981 it would look like this:

$1000 =$	one thousands unit	$= 1 \times 10^3 =$	$1000 \times 1$	
$900 =$	nine hundreds units	$= 9 \times 10^2 =$	$100 \times 9$	
$80 =$	eight tens units	$= 8 \times 10^1 =$	$10 \times 8$	
$+ 1 =$	one ones unit	$= 1 \times 10^0 =$	$1 \times 1$	
$1981$				$1981$

It can also be looked at in a different way:

$1024 =$	one 1024 unit	$= 1 \times 2^{10} =$	$1024 \times 1$	
$512 =$	one 512 unit	$= 1 \times 2^9 =$	$512 \times 1$	
$256 =$	one 256 unit	$= 1 \times 2^8 =$	$256 \times 1$	
$128 =$	one 128 unit	$= 1 \times 2^7 =$	$128 \times 1$	
$0 =$	no 64 unit	$= 0 \times 2^6 =$	$64 \times 0$	
$32 =$	one 32 unit	$= 1 \times 2^5 =$	$32 \times 1$	
$16 =$	one 16 unit	$= 1 \times 2^4 =$	$16 \times 1$	
$8 =$	one 8 unit	$= 1 \times 2^3 =$	$8 \times 1$	
$4 =$	one 4 unit	$= 1 \times 2^2 =$	$4 \times 1$	
$0 =$	no 2 unit	$= 0 \times 2^1 =$	$2 \times 0$	
$+ 1 =$	one 1 unit	$= 1 \times 2^0 =$	$1 \times 1$	
$1981$				$11110111101$

A one shows the presence of a power of two and a zero shows its absence

$11110111101$

As you can see, here the numbers are no longer divided by powers of ten, but rather by powers of two. It should be remembered that we are still talking about the same number, 1981. Even in the binary table, numbers to the base ten have been used (all the numbers not made up of 'ones' and

'noughts'). By now it should be clear that, when working in binary, only two symbols are possible; 1 and 0.

## Why?

Readers may well be wondering why we should leave our tried and trusted base ten system for a crazy one with only two numbers. The answer is quite simple. If we take another look at the binary equivalent of 1981 we can see that there is always a 'yes' or 'no' answer to the question: 'Is there, or is there not a power of two present at any given position in the number 11110111101?'. A 'yes' answer would be indicated by a '1' and a 'no' answer by a '0', thus there are only two possibilities.

Numbers have to be interpreted, moved and manipulated in one way or another by electronic circuitry inside the computer. This means that some form of electronic device will have to operate for each number. If the computer is working with a base two numbering system then only an 'on'/'off' relationship is required. The two possibilities are usually 'logic states' and are defined as follows:

1 = logic one = a voltage is present or 'high'

0 = logic zero = no voltage (0 volts) or 'low'.

(Note: This is termed 'positive logic'. The vast majority of logic circuitry works in this manner, but a certain amount of 'negative logic' is used which is exactly the opposite of that given above).

The obvious advantage here is that it is much simpler to design an electronic circuit for binary operation (only two possible output states) than one for decimal operation (ten possible output states).

There is also another advantage to using a binary system and that is in decision making. We shall be covering 'flow charts' later in this chapter and when developing these there are often times when a decision has to be made. Something along the lines of: does ABC equal XYZ? If the answer had to be given in a base ten system there could be ten different choices, whereas in a base two system there are only two, 'yes' or 'no'. If ten choices are desirable then four yes/no decisions can follow one another.

Looking at it in this light it is also possible to say that the number 11110111101 is *simpler* than the number 1981. The former gives only one choice per position, a power of two or no power of two. The latter however gives ten possible (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) factors of the power of ten per position. By opting for greater length rather than greater breadth, the computer is allowed to operate much more efficiently.

## Bits and bytes

The word 'bit' has been coined so that it is possible to refer to an individual digit in a given binary number. Bit stands for Binary digit. A bit can therefore take the form of either a '0' or a '1'. Bits are almost always in groups or 'words'. Just as the information on this page consists of letters to make up words, the same can be done with bits. If the word consists of eight bits then it is generally called a 'byte' (there are other more drawn-out systems where there are sixteen bits to a word). Words with only four bits are sometimes called 'nibbles'.

You should remember from chapter 1 that the Junior Computer's data bus is eight bits wide. This means that the data bus transfers information one byte at a time. For the sixteen bit address bus however, two bytes are required.

Bits and bytes are not only used to describe data transfer length, but also for:

- computer instructions that are entered by the user.
- the four bit code used for defining the normal base ten numbers in binary. This is called binary-coded-decimal (BCD) and is dealt with later on.
- the sixteen bit (double-byte) address code which defines a memory location or peripheral address.
- ASCII (American Standards Code for Information Interchange). This is a code that represents all the letters of the alphabet, numerals, punctuation marks and many other symbols.

All of these (and many other) digital codes consist of a number of bits each of which can be either a '1' or a '0'.

## Hexadecimal

We accept the fact that a numbering system with only two digits is ideal for the computer, but what about human/computer intercommunication? If information has to be entered into the computer in the form of ones and noughts the only possible result is total chaos. Think about double-byte addressing for example; sixteen numbers for one address location! In practice this just will not work as mistakes are inevitable.

To overcome this problem a simpler numbering system is required, one which can easily be interpreted by both computer and (more importantly) operator. The most obvious choice for this is the *hexadecimal* system. Hexadecimal simply means sixteen numbers. Earlier, when we discussed conversions from decimal to binary numbers, we saw that the amount of figures required to express a number in binary increased dramatically. It is therefore only logical to assume that if a base higher than ten be chosen (in this case sixteen) the amount of figures required to represent a number should decrease. With a base sixteen system, binary numbers are reduced in length by a factor of four. In other words, an 8-bit data byte will only require two symbols. For a decimal system ten symbols are required, namely 1, 2, 3, 4, 5, 6, 7, 8, 9 and 0. For a binary system only two are necessary, one and zero. It follows then that for a numbering system with a base of sixteen, sixteen symbols are required.

Since there are only ten symbols in common usage, six new ones will have to be created. The numbers 10 to 15 can not be used here because once the number 9 is exceeded the situation gets extremely confused. The hexadecimal system therefore uses the 'normal' numbers 0 . . . 9 and the letters A, B, C, D, E, and F. What really happens is that one of the sixteen symbols is assigned to a four bit word as follows:

0 = 0 = 0000	4 = 4 = 0100	8 = 8 = 1000	C = 12 = 1100
1 = 1 = 0001	5 = 5 = 0101	9 = 9 = 1001	D = 13 = 1101
2 = 2 = 0010	6 = 6 = 0110	A = 10 = 1010	E = 14 = 1110
3 = 3 = 0011	7 = 7 = 0111	B = 11 = 1011	F = 15 = 1111

That seems easy enough, but how is it put into practice?

Quite simply. A binary number is divided into nibbles (groups of four bits each) from right to left. The division must come out evenly. There are times when one, two or three numbers are missing and it is impossible to divide the number without a remainder. In this case, as many noughts as required are added to the left hand side to make the division work out correctly.

For example:

$$\underbrace{(0)110101010111100}_{6} = 6ABC_{16}$$

6    A    B    C

The index '16', behind the 6ABC, indicates that we are talking about the hexadecimal code. It is rarely used in practice, but is shown here to familiarise readers with it in case they should ever come across it.

To avoid confusion, nearly all computer information and specifications are given in hexadecimal code, even if it is not intended as a binary number. It should also be pointed out that the hexadecimal system is not just 'binary shorthand'. It is a fully-fledged base sixteen numbering system. The number  $6ABC_{16}$  (if expanded) would look like this:

$$6 \times 16^3 + A \times 16^2 + B \times 16^1 + C \times 16^0 =$$
$$6 \times 4096 + 10 \times 256 + 11 \times 16 + 12 \times 1 = 27,324_{10} \text{ (decimal)}$$

This can be checked against the binary number from which 6ABC was derived.

The reason why everything works out so nicely is because 16 is a power of 2. Two is the base of the binary system and  $2^4 = 16$ . Notice the power of four here. Remember that the binary number was divided up into groups of four? This is the explanation.

An older system does exist, called octal, which divides the binary number into groups of three. The base here is not sixteen but, as its name suggests, eight and the code consists of the numbers 0 . . . 7. If we take a look at the powers of two once more, we see that  $2^3 = 8$ . In the future a system with a base of 32 (the next step on the base two ladder) may come into use, but that would involve a considerable amount of (human) memory work. The operator would have to learn an extra 22 numbers (those besides the normal 0 . . . 9).

## BCD

As previously mentioned, BCD is an abbreviation of Binary Coded Decimal. Every number in the decimal number set is assigned a four bit binary code, thus:

0	=	0000
1	=	0001
2	=	0010
3	=	0011
4	=	0100
5	=	0101
6	=	0110
7	=	0111
8	=	1000
9	=	1001

As can be seen, the codes are exactly the same as those for the numbers 0 . . . 9 in the hexadecimal system. The BCD expression for a larger decimal number simply consists of the relevant amount of these four bit codes. For example, the number 1981 expressed in BCD form is:

0001100110000001  
1 9 8 1

It looks nothing like the binary equivalent of the same number. As we already know, that is:

11110111101

By splitting the BCD number into groups of four bits each, it is very easy to deduce the decimal number it represents. One disadvantage of the BCD system is the fact that any mathematical function is difficult to perform. A table of all the number systems of use to the Junior Computer operator are shown in figure 1.

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

Figure 1. This table shows the various number sets that will be used by Junior Computer operators. The leading zeros are dropped from the binary numbers as they are not strictly necessary. This is also true of normal decimal numbers: we would write '7' rather than '07'.

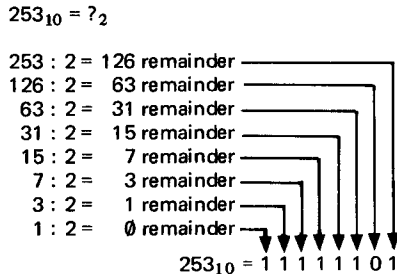


## Binary conversion

It is often necessary to convert a decimal number into its binary equivalent and vice versa. To convert from binary to decimal is extremely simple. Where a '1' occurs in a binary number then the number contains the relevant power of two. The powers of two are simply written down in their decimal form and added together to produce the decimal equivalent as shown in the following example:

$$\begin{aligned} 11010111 &= 2^0 + 2^1 + 2^2 + 2^4 + 2^6 + 2^7 \\ &= 1 + 2 + 4 + 16 + 64 + 128 = 215 \end{aligned}$$

Conversion from decimal to binary is a little more complicated as illustrated in figure 2. Basically, it involves dividing two into the number and writing down the remainder after each division. The sequence of remainders obtained by dividing the number right down to zero represents the binary equivalent of the number. It is, of course, possible to convert decimal numbers directly into their hexadecimal equivalents and vice versa, but it is much easier to convert the number into binary first.



**Figure 2.** The conversion of a decimal number into its binary equivalent is carried out by repeatedly dividing the decimal number by two. The remainder from the division, whether it be one or zero, is written down to form the binary number as shown.

## Binary arithmetic

Binary numbers can be added, subtracted, multiplied and divided, just like any other kind of number. The basic principle is the same as that for the decimal system. However, since the base of the binary system is only two, this does lead to certain simplifications and short cuts.

### Addition

When adding two decimal numbers together, it is quite usual to have to carry a one if the amount in one column exceeds 9. For example:

1	carry the one from the units column
129	
+ 243	
372	sum

There are also carry-overs in binary addition, but the carry occurs when the amount in any column exceeds one (this is a base two system, not base ten). Another example will clarify this:

$$\begin{array}{r}
 1111\ 1 \qquad \text{carry one from preceding column} \\
 101101 \\
 + \quad 10101 \\
 \hline
 1000010 \qquad \text{sum}
 \end{array}$$

All the normal rules of mathematics still apply;

$$\begin{array}{l}
 0 + 0 = 0 \\
 0 + 1 = 1 \\
 1 + 0 = 1 \\
 1 + 1 = 0 \text{ and carry the } 1 \text{ thus} = 10
 \end{array}$$

These basic rules can be seen quite clearly in the binary (as well as the decimal) columns in figure 1. Each number is equal to the preceding number plus one.

## Subtraction

When subtracting in decimal a system of borrowing numbers is used to ensure that the difference in any one column is not negative, as the following example shows:

$$\begin{array}{r}
 87 \qquad \text{borrowing a one from the 8 results in 14 units} \\
 1984 \qquad \text{borrowing a one from the 9 results in 17 tens} \\
 - \quad 199 \\
 \hline
 1785 \qquad \text{difference}
 \end{array}$$

For binary numbers the method is the same:

$$\begin{array}{r}
 111111 \qquad \text{borrowed} \\
 11000001 \\
 - \quad 1111110 \\
 \hline
 01000011 \qquad \text{difference}
 \end{array}$$

The rules of subtraction learned at school apply yet again:

$$\begin{array}{l}
 0 - 0 = 0 \\
 1 - 0 = 1 \\
 1 - 1 = 0 \\
 0 - 1 = 1 \text{ after borrowing, thus } 10 - 01 = 01
 \end{array}$$

Again, this can be clearly seen in the binary column of figure 1. Starting from the bottom, if you follow the column up you will see that as you move up a line, one has been subtracted from the line beneath.

This method of subtraction is all very well, but as far as the computer is concerned it is rather long-winded. A much more elegant method exists whereby subtraction can be performed by addition! Sounds Irish? Read on.

The method the computer uses is called 'complement and add'. The complement of a binary number is obtained by inverting each bit in that

number. Thus the complement of 1001 is 0110. The computer performs the subtraction of two numbers by complementing the subtrahend (the number to be subtracted) and adding it to the minuend (the number to be subtracted from). The result of the carry from the sum of the most significant (left hand) bits is used to determine the next operation. If the carry bit is a 1, it indicates that the result is a positive number and that a procedure called 'end around carry' must be performed to obtain the final result of the subtraction. This simply means that the carry bit is added to the least significant (right hand) bit of the intermediate result. If, on the other hand, the carry bit is 0, the result is negative and the final answer is the complement of the sum (intermediate result). A couple of examples should serve to clarify the situation:

$$\begin{array}{r}
 1001 \text{ (9)} - 0011 \text{ (3)} \\
 1001 \text{ (9)} \\
 \underline{1100} \text{ (complement of 3)} \\
 1\ 0101 \text{ (sum)} \\
 \quad \underline{1} \text{ (end around carry)} \\
 + 0110 \text{ (result)}
 \end{array}$$

Answer = +6

$$\begin{array}{r}
 1000 \text{ (8)} - 1100 \text{ (12)} \\
 1000 \text{ (8)} \\
 \underline{0011} \text{ (complement of 12)} \\
 0\ 1011 \text{ (sum)} \\
 - 0100 \text{ (result = complement of sum)}
 \end{array}$$

Answer = -4

This may seem rather drawn out for the human but as far as the computer is concerned it is a much easier and therefore faster and more efficient method of subtraction.

## Multiplication

Here again, the same rules apply as for decimal multiplication. To start with, multiply the decimal numbers 147 and 231 together:

$$\begin{array}{r}
 147 \\
 \times 231 \\
 \hline
 147 \\
 441 \\
 + 294 \\
 \hline
 = 33957
 \end{array}$$

First the result from the multiplication of 147 by 1 is written down. Following that comes the product of 147 x 3 and as the multiplication is actually between 147 and 30 the answer is shifted one place to the left. Finally, the result of 147 x 2 is written down. As before, this is shifted because it is actually an operation between 147 and 200. This time, of

course, it is shifted two places to the left. These three totals are then added together to produce the required answer (33957). If a digit in the multiplier is 1, then all that is required for that line of the calculation is to shift the multiplicand one place to the left. If a digit of the multiplier is 0, then that answer is 0, but the next answer is shifted two places to the left. In binary multiplication, the only existing digits are 1 and 0, so that the multiplication procedure consists simply of shift and add steps. Here is an example where the numbers 1011 and 1010 are multiplied:

$$\begin{array}{r}
 1011 \\
 \times 1010 \\
 \hline
 0000 \\
 1011 \\
 0000 \\
 1011 \\
 + 1 \qquad \text{carry one, due to the addition} \\
 \hline
 1101110 \quad \text{product}
 \end{array}$$

The computer would perform the operation step by step as follows:

1011 (11) x 1010 (10)  
 multiplier (1010) bits

	multiplicand:	1011	
LSB = 0	write zero		0000
	shift left	10110	
bit 2 = 1	add		10110
	shift left	101100	
bit 3 = 0	add zero		10110
	shift left	1011000	
MSB = 1	add		1101110
	answer		1101110

## Division

First a trusted decimal example. The division of 2091 by 17:

$$\begin{array}{r}
 123 \\
 \hline
 17 \overline{) 2091} \\
 \underline{17} \phantom{00} \\
 39 \phantom{0} \\
 \underline{34} \phantom{0} \\
 51 \\
 \underline{51} \\
 00
 \end{array}$$

Binary division is just the same, but even simpler. If the remainder is larger than the divisor then a 1 is written in the corresponding position of the quotient. If the remainder is smaller than the divisor then a 0 is written in the quotient. As an example, the number 100010010101 (2197) is divided by 1101 (13) to give a result of 10101001 (169):

$$\begin{array}{r}
 \underline{10101001} \\
 1101) 100010010101 \\
 \underline{1101} \\
 001000 \\
 \underline{0000} \\
 10000 \\
 \underline{1101} \\
 000111 \\
 \underline{0000} \\
 01110 \\
 \underline{1101} \\
 000011 \\
 \underline{0000} \\
 00110 \\
 \underline{0000} \\
 01101 \\
 \underline{1101} \\
 0000
 \end{array}$$

All operations have been included in the above example for the sake of clarity. In practice however, the steps resulting in a zero product would be omitted and the digits of the number to be divided brought down until the sum is greater than the divisor. This would result in a much shorter calculation as shown below:

$$\begin{array}{r}
 \underline{10101001} \\
 1101) 100010010101 \\
 \underline{1101} \\
 10000 \\
 \underline{1101} \\
 1110 \\
 \underline{1101} \\
 1101 \\
 \underline{1101} \\
 0000
 \end{array}$$

If calculations have to be carried out with hexadecimal numbers, it would be advisable to first convert them into binary form, perform the operation, and then convert the result back into hexadecimal.

### Negative numbers

Up to now, all calculations have been carried out with positive numbers.

Negative numbers can (and do) exist in the binary system as well as in the decimal system, albeit in a different form.

There is absolutely no point in placing a minus (-) sign in front of a binary number, for the computer will not recognise it. As would be expected, there are different methods of representing negative binary numbers, but we shall concentrate on the most common method. In any case, this is the method used in the Junior Computer.

The 6502 microprocessor used in the JC has a data word length of eight bits (1 byte). This means that it is possible to produce 256 different combinations: everything between (and including) 00000000 and 11111111 (00 . . . FF in hexadecimal, and 0 . . . 255 in decimal). For more combinations, more bits would have to be added. The numbers 00000000 . . . 11111111 are shown on a vertical line rather like a tape measure in figure 3. The distance between any two is the same, while the furthest possible distance between any two is 255 units. The total cannot exceed 11111111 because there is no ninth bit. (In chapter three, we shall see that a ninth bit is made available by using a so-called 'carry flag').

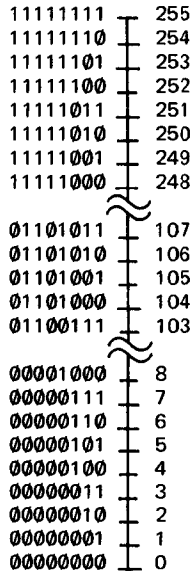


Figure 3. Number line for all possible positive eight bit numbers.

How does all this affect negative numbers? Take seven of the available eight bits in a given byte. This would mean that the possible combinations would range from 00000000 up to and including 01111111. A total number of 128 combinations, half the number that are possible with an 8 bit word. These would represent the positive numbers in binary system. The negative numbers are represented as follows.

Starting from zero subtract one, thus:

$$\begin{array}{r}
 00000000 \\
 - 00000001 \\
 \hline
 11111111 = \text{zero minus one} = -1 \\
 - 00000001 \quad (\text{now subtract one again}) \\
 \hline
 11111110 = -2 \\
 - 00000001 \quad (\text{and again}) \\
 \hline
 11111101 = -3
 \end{array}$$

Continuing like this will produce a number line with its centre at zero and its two extremes being +127 and -128 (see figure 4). The negative numbers are given in 'two's complement' notation. It is not the intention to confuse the reader with all sorts of weird and wonderful formulae, but rather to show how all the various mathematical functions are performed. A two's complement number is obtained by first complementing the binary number and then adding one to the result. As mentioned before, complementing a binary number consists simply of inverting each bit of the number. As an example, positive 3 (decimal) is represented by 00000011. Substituting all the ones for zeros and vice versa, results in 11111100. Adding one (00000001) to this gives us 11111101, which is the equivalent of -3 in binary (see above and figure 4).

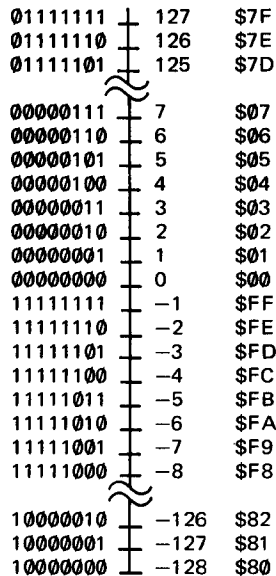


Figure 4. Number line for all positive and negative eight bit numbers using the two's complement method. A dollar sign precedes the hexadecimal equivalents.

It is therefore possible to produce negative binary numbers at the cost of one bit (the eighth). This is due to the fact that the total number of variations is always limited by the number of bits available. Negative binary numbers always start with a one (the left-most bit), while positive binary numbers of the same word length always start with a zero. This zero can be dropped if desired. In this instance, the MSB (Most Significant Bit) can be likened to a positive/negative sign. Various negative binary numbers, along with their hexadecimal code, are shown in figure 5.

### **Analysing problems by means of flow charts**

Regardless of the actual job function the Junior Computer is required to perform, whether it be to play games, handle accounts, control domestic appliances etc., it first has to be programmed by the user. Before the computer can be programmed, however, a detailed analysis of the particular problem is required. For very small and simple programs it may be possible to convert the problem directly just by using the instruction set (see chapter 3) of the microprocessor. This is certainly not true of longer and more complicated programs.

A flow chart gives an overall view of the problem solving process from start to finish. It will indicate all the various points in the program, at which tests and decisions have to be made, and what operation to perform upon obtaining a particular result etc. At some points there may be a number of possible paths that can be taken, depending on the result of a particular operation. All these paths can be clearly mapped out in the flow chart. The most common flow chart symbols, together with their meaning, are shown in figure 6.

The first step in any flow chart is 'start'; this is contained in a so-called 'terminal' symbol. The same symbol, with the word 'end', is used to indicate the completion of the program.

Program 'operations' are contained in a rectangle and the text inside the rectangle defines the actual operation to be performed. This text should be kept as short as possible and only the steps required for complete understanding of the operation are necessary. In the example given in figure 6, A is assigned the value of the sum of B plus C. This is quite sufficient and steps like fetch A, fetch B, etc. are omitted.

Once an operation has been performed it may be necessary to make a 'decision' about the result. Decisions are indicated in the flow chart by a diamond shaped symbol. In the example, the value of A is compared to the value of D and a decision on what to do next is made, depending on whether the two values are equal or not. The decision may well lead to the next symbol shown in figure 6. that of the 'input/output' statement. This is indicated in the flow chart by a parallelogram. At this stage in the flow chart information can be entered into or output from the computer. For example the operator may require the computer to provide an intermediate result, or the computer may require a new value for a particular variable, etc.

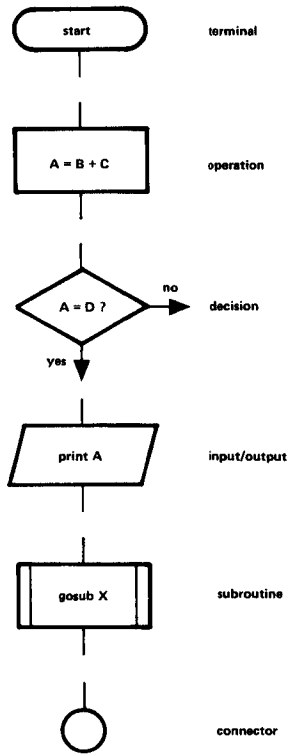
There are occasions when a certain operation, or series of operations, has to be performed a number of times. When this occurs it is usual (and good



binary	decimal	hexa- decimal
11111111	- 1	FF
11111110	- 2	FE
11111101	- 3	FD
11111100	- 4	FC
11111011	- 5	FB
11111010	- 6	FA
11111001	- 7	F9
11111000	- 8	F8
11110111	- 9	F7
11110110	-10	F6
11110101	-11	F5
11110100	-12	F4
11110011	-13	F3
11110010	-14	F2
11110001	-15	F1
11110000	-16	F0
11101111	-17	EF
11101110	-18	EE
11101101	-19	ED
11101100	-20	EC
11101011	-21	EB
11101010	-22	EA
11101001	-23	E9
11101000	-24	E8
11100111	-25	E7
11100110	-26	E6
11100101	-27	E5
11100100	-28	E4
11100011	-29	E3
11100010	-30	E2
11100001	-31	E1
11100000	-32	E0
11011111	-33	DF
11011110	-34	DE
11011101	-35	DD
11011100	-36	DC
11011011	-37	DB
11011010	-38	DA
11011001	-39	D9
11011000	-40	D8
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.

**Figure 5.** This table shows some of the negative binary numbers along with their corresponding decimal and hexadecimal values.

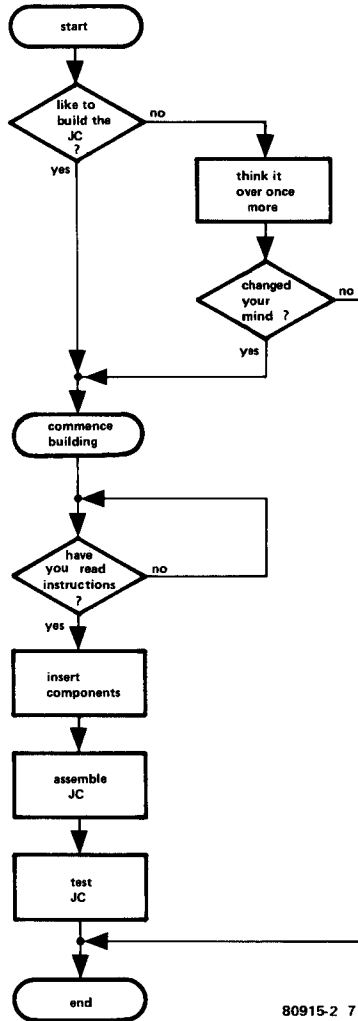
practice) to enclose the sequence as a small program or 'subroutine' inside the main program. The symbol for a subroutine is shown as a rectangle with two straight lines on each side. Each subroutine would, of course, have its own flow chart. As expected, long and complicated programs lead to long and complicated flow charts which may well spread over quite a few pages. The various parts of the flow chart can be joined together by means of the 'connector' symbol also shown in figure 6.



80915-2 6

**Figure 6. The most commonly used flow chart symbols. There are many more, but the majority of programs can be charted using just these six.**

Just by using these symbols, it is possible to 'rough out' virtually any program. Once a rough flow chart is drawn the various subroutines are filled in. As the process continues the diagram becomes more and more defined. A time will come when the flow chart cannot be expanded any further and the various symbols can be converted into instructions for the Junior Computer.

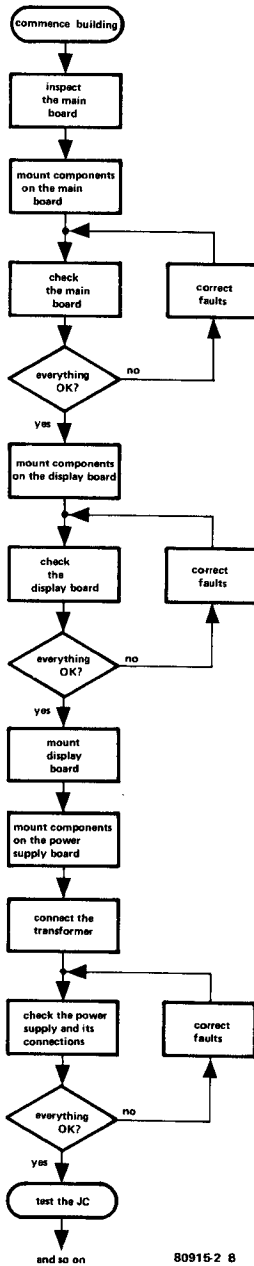


80915-2 7

Figure 7. The rough flow chart for the building of the Junior Computer.

By way of illustration, an actual example of a flow chart is shown in figure 7. The problem concerns the construction of the Junior Computer as outlined in chapter 1 and will of course never result in a computer program.

As can be seen, the first decision to be made is whether or not to build the JC. Once this decision has been made and the response is positive, the



80916-2 8

Figure 8. The flow chart of figure 7 has been further expanded to show all the basic constructional steps mentioned in chapter 1.

constructional instructions are to be read followed by the installation of the various components, the assembly, and the test procedure. If the answer was no then the flow chart prompts the reader to have a re-think. If the answer is still no the reader is directed to the end of the flow chart. On the other hand, if the reader has changed his/her mind then he/she is directed back into the main stream of the program.

his/her mind then he/she is directed back into the main stream of the program.

The flow chart in figure 7 is only a very rough diagram. It is further expanded in figure 8. It can be seen that all the basic sections mentioned in chapter 1 are laid out here.

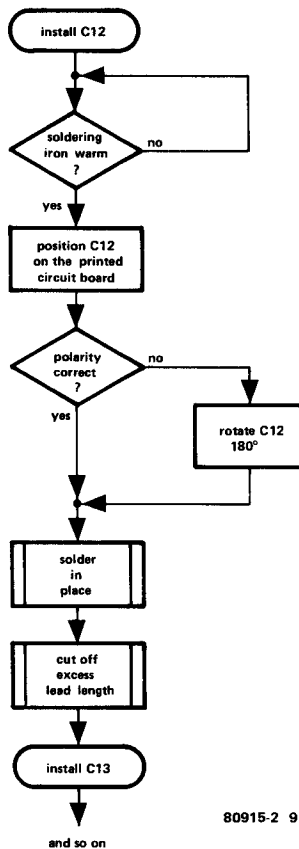


Figure 9. Part of the block marked "mount components on the main board" in figure 8 consists of the installation of C12. There are three blocks shown as subroutines which means that this flow chart can be expanded still further.

Taking it a step further, we can produce a flow chart for the installation of individual parts on the main board. This is illustrated in figure 9. Three boxes are shown as subroutines in figure 9, which means that the operation which takes place has yet to be fully defined. More will be said about this in the next chapter.

### Exercise time

From the following exercises we can deduce how much we have learnt about hexadecimal and binary number systems.

- 1) *Convert the following decimal numbers to binary:*
  - a) 16
  - b) 24
  - c) 125
  - d) 513
  - e) 756
- 2) *Convert the following binary numbers to decimal:*
  - a) 0111
  - b) 1001
  - c) 1100101
  - d) 1011011
  - e) 1110010101
- 3) *Convert the following decimal numbers to BCD:*
  - a) 12
  - b) 37
  - c) 128
  - d) 412
  - e) 3762
- 4) *Convert the following BCD numbers to decimal:*
  - a) 1001
  - b) 0101
  - c) 10000110
  - d) 00111000
  - e) 100101110010
- 5) *Convert the following binary numbers to hexadecimal:*
  - a) 00101111
  - b) 11111
  - c) 101000111
  - d) 110101010
  - e) 001011
- 6) *Convert the following hexadecimal number to binary:*
  - a) 132
  - b) A014
  - c) 0356
  - d) C5E1
  - e) ABBA (not to be confused with the Swedish import)
- 7) *Perform the following binary calculations:*
  - a) 01001111 + 11000111

- b)  $1110011 + 1111111$   
 c)  $1111111 + 1$   
 d)  $11110000 + 1111$   
 e)  $10101010 + 1010101$   
 f)  $01110100 - 1101$   
 g)  $11110000 - 1111$   
 h)  $10111000 - 10000001$   
 i)  $10101111 - 10101111$   
 j)  $100 - 11111011$   
 k)  $11110001 \times 01111$   
 l)  $101 \times 1111111$   
 m)  $1010 \times 1010$   
 n)  $11 \times 1111111$   
 o)  $10000000101 \div 111$   
 p)  $110100000000 \div 1101$   
 q)  $10011011110110010 \div 1001110$
- 8) Perform the following hexadecimal calculations:
- a)  $A + B$   
 b)  $D3 - 3E$   
 c)  $ABBA \times 4$   
 d)  $B9A0 \div 0B$

## Answers

- 1: a)  $10000$   
 b)  $11000$   
 c)  $1111101$   
 d)  $1000000001$   
 e)  $1011110100$
- 2: a) 7  
 b) 9  
 c) 101  
 d) 91  
 e) 917
- 3: a)  $00010010$   
 b)  $00110111$   
 c)  $000100101000$   
 d)  $010000010010$   
 e)  $0011011101100010$
- 4: a) 9  
 b) 5  
 c) 86  
 d) 38  
 e) 972
- 5: a) 2F  
 b) 1F  
 c) 147  
 d) 1AA  
 e) 0B

- 6: a) 000100110010  
 b) 1010000000010100  
 c) 0000001101010110  
 d) 1100010111100001  
 e) 1010101110111010
- 7: a) 100010110  
 b) 101110010  
 c) 100000000  
 d) 11111111  
 e) 11111111  
 f) 1100111  
 g) 11100001  
 h) 110111  
 i) 0  
 j) 100001001 (−247 in 9 bit two's complement notation)  
 k) 111000011111  
 l) 10011111011  
 m) 1100100  
 n) 1011111101  
 o) 10010011  
 p) 100000000  
 q) 1111111111
- 8: a) 15  
 b) 95  
 c) 2AEE8  
 d) 10E0



# Programming

## Operating the JC

Now that the Junior Computer is complete and operational, and the background information from chapter 2 has been digested, we come to the 'user's manual'. What can we do with the JC and how do we go about it? These questions will be answered in this chapter.

Before we are able to use the JC we must have some knowledge of the more serious aspect of programming. Do not worry however, as programming can be quite fun. By the end of this chapter it should be possible to develop your own (short) programs and to make the JC perform more or less as you want it to. Chapter 4, on the other hand, deals with more complex programming and gives some programming 'tricks' to further simplify matters. But first . . .

### Familiarisation with the terrain

As we know, the hardware 'dashboard' consists of two toggle switches, 23 keyboard switches and 6 seven-segment displays — as shown in figure 1a. Once the system has been switched on, the RST key is depressed to initialise the microprocessor. The monitor program will then scan the keyboard to check whether or not any of the switches are being depressed. If one of the hexadecimal switches is pressed (0 . . . F), this will be indicated on the display.

The four displays on the left-hand side will show the address location in hexadecimal code and the two right-hand ones will give the 'contents' of that location. In principle, all the address locations from 0000 . . . FFFF are possible.

Take for example that we want to enter certain data into a given area of memory. From address location 0200 on we want to enter 18, A9, 03 etc. The sequence of operations is as follows:

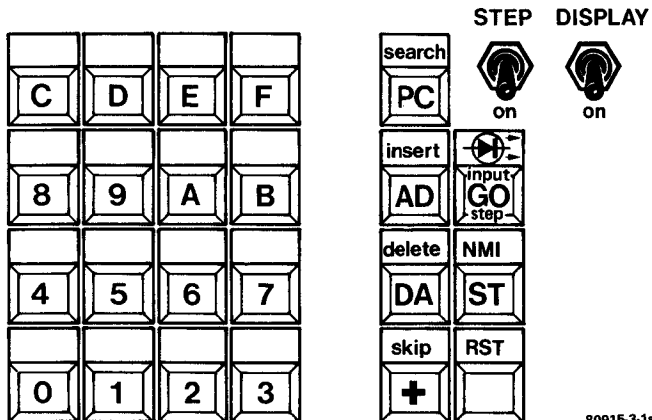
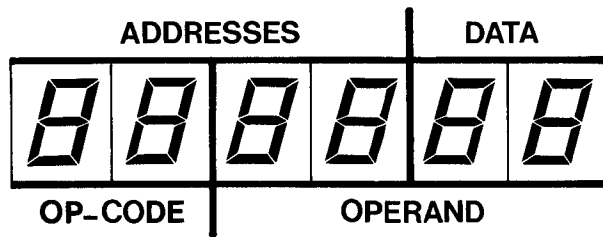
				adres	data				
RST				xxxx	xx				
AD				xxxx	xx				
0	2	0	0	0200	xx				
DA				0200	xx				
		1	8	0200	18	0200	1 8		CLC
+		A	9	0201	A9	0201	A 9		LDA
+		0	3	0202	03	0202	0 3		
+		6	9	0203	69	0203	6 9		ADC
+		0	7	0204	07	0204	0 7		
+		8	D	0205	8D	0205	8 D		STA
+		0	0	0206	00	0206	0 0		
+		0	3	0207	03	0207	0 3		
+		4	C	0208	4C	0208	4 C		JMP
+		0	0	0209	00	0209	0 0		
+		0	3	020A	03	020A	0 3		
AD						020B	X X		
1	A	7	A	1A7A	xx	020C	X X		
DA				1A7A	xx	1A79	X X		
		0	0	1A7A	00	1A7A	0 0		
+		1	C	1A7B	1C	1A7B	1 C		
						1A7C	X X		

That was certainly a bit long-winded. What have we done exactly? First, the RST key initiated the monitor program. By depressing AD we informed the computer that we wish to place data at a certain address location. The 'crosses' (x) at various points in the chart indicate that the corresponding data is irrelevant. In other words, the state of the data lines is unimportant, they could be high (1) or low (0).

The address location we want to start at is 0200. By pressing the keys 0, 2, 0 and 0 (again) the correct address will be ready to receive data. After pressing DA followed by 18, memory location 0200 is loaded with 18 (hexadecimal). Strictly speaking 'loaded' is not the correct term as location 0200 was not empty. The instruction 18 (for that is what it is) 'replaced' the previous contents of the memory location.

The address is incremented (advanced) by pressing the '+' key. The following data will then be placed in the next memory location. It is not necessary to depress DA again unless the data is to be placed elsewhere in memory. If, for instance, the following data is to be placed in address location 1A7A, the AD key will have to be pressed first.

On the right-hand side of the chart is a 'memory map' which shows the contents of each of the locations that are of importance to this particular program. The two columns of boxes are: on the left, the address location and on the right, the contents of that address location.



80915-3-1a

Figure 1a. The 'hardware dashboard' of the Junior Computer. Besides the sixteen hexadecimal keys and the seven control keys, there are two function switches. The text in small letters indicates the editing functions that the JC can perform. Under normal conditions the four left-hand displays show a particular address location and the two right-hand displays the data contained at that address.

We should now be familiar with the following keyboard switches:

- 0 . . . F, for entering data and address information;
- RST, which initialises the microprocessor and activates the monitor program and the display;
- AD, which instructs the computer to go into the address mode for loading one or more address locations;
- DA, which instructs the computer to go into the data mode;
- +, which advances the address location by one. The mode (address or data) is not affected.

Note: As with most pocket calculators, the address and data information is keyed in from left to right, but the display will appear to move from right to left.

As mentioned earlier, all the information we have just entered is not a random set of hexadecimal figures, but a real program to add two numbers together and store the result. This is what happens:

The first operation code, 18 (at address location 0200), is the code for the instruction CLC-Clear Carry. The computer then moves on to address

0201 where it finds the code (A9) for Load Accumulator immediate — LDA. The data to be loaded (03) is contained in the next address 0202. Address 0203 contains the 'op-code' (operation-code) 69 which instructs the microprocessor to Add the contents of the following memory location to the accumulator with Carry. The contents of the following address (0204) is 07, the accumulator contains 03, the result after the addition will, therefore, be 0A in the accumulator.

The op-code 8D at address 0205 instructs the microprocessor to Store Accumulator in memory — STA. As absolute addressing (more about this later) is used for this instruction, the actual location where the result is to be stored is found from the contents of the following *two* addresses, 0206 and 0207. The last two bytes of the destination are found in address 0206, and the first two bytes in 0207. Thus the destination address for the contents of the accumulator is 0300 — *not* 0030. The next address in the program (0208) contains the op-code 4C, the code for Jump, or Jump instruction. This tells the computer to jump to the location contained in the next two addresses. 0209 and 020A, which in this case is 0300. This is also where the result of the operation (addition) is to be found. The computer will then perform the operation corresponding to the op-code 0A which is ASL-A, Arithmetic Shift Left Accumulator, or move the contents of the accumulator one place to the left. As there is no instruction at address 0301, the computer will not know what to do next and will 'crash' the program.

It is important to realise therefore, that the operation of a program is hardly ever self-explanatory and care must be taken when developing programs to ensure that the processor always has 'something to do'. Programming is all about selecting various instructions and arranging them in the correct order so that the computer performs a certain task to produce the desired result. As always, the old adage applies — ask a silly question and you get a silly answer!

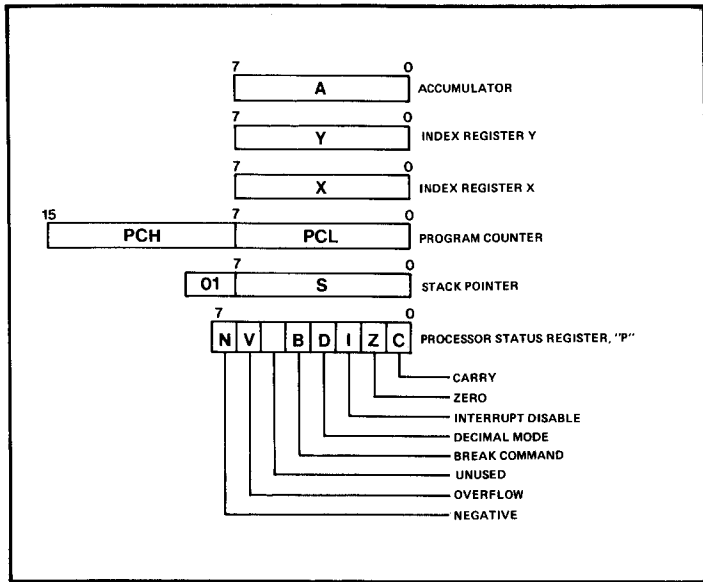
### The 'software-dashboard'

Before we go further into the software possibilities of the Junior Computer, such as the instruction set and the different addressing modes, it is advisable to have a clear idea of the internal register structure of the 6502 microprocessor. The processor contains six programmable registers as shown in the 'software-dashboard' of figure 1b.

The rectangle marked 'A' in the figure is the eight-bit accumulator. This register is used to manipulate data and to transfer information to and from memory. It is often used as a 'waiting station' for data travelling between memory locations.

The X and Y index registers (also 8-bit registers) contain data whereby any memory location can be addressed indirectly. This will be expanded on later.

The program counter, PC, is a sixteen-bit register which contains the address of the memory location where the next instruction is to be found. The program counter is divided into two registers, PCL (L stands for Low) and PCH (H = High), which contain the lower order address bytes and the higher order address bytes respectively.



**Figure 1b.** The 'software dashboard' of the Junior Computer — or to be more precise, the 6502 microprocessor. This figure shows all the internal registers that are accessible to the user. The status register could also be called the 'flag' register.

The stack pointer, S, is used for temporary storage of address locations. The stack pointer is 8-bits wide which means that the maximum stack length is 256 bytes. The processor always appends 01 as the high order byte of any stack address, which means that memory locations 0100 . . . 01FF are permanently assigned to the stack.

Lastly, the (processor) status register, P. This register contains information which reflects the results of various operations in the form of 'flags'. Each flag is effectively connected to a flip-flop which can be set or reset. One such flag is the 'carry'. This is set (logic '1') as soon as the result of an operation (addition) produces a carry from the highest (most significant) bit. This flag can be used as a 'ninth' bit. Flags N and Z indicate whether the result was negative or zero respectively. The remaining flags will be dealt with in greater detail later on.

### Address repertoire

There are a full thirteen different addressing modes available to the user of the JC. This is one of the strong points of the 6502 microprocessor. The instruction set consists of 56 'powerful' instructions which, when combined with so many addressing possibilities provide the operator with an optimum amount of freedom.

Although the 6502 has fewer instructions than most other microprocessors,

this does mean that there are fewer instructions to remember! All instructions are decoded by the computer with the aid of its internal 'micro-program'. To help the operator (programmer) all instructions also have an abbreviated form where mnemonics are used – ADC is a lot shorter than writing 'add memory to accumulator with carry'.

### Immediate addressing

Instructions for immediate addressing refer to data in the work memory which are dealt with as soon as the instruction is known. The instructions consist of two bytes; the first for the actual instruction (op-code) and the second for the operand data. The symbol # is used to indicate that the following number is data, as some examples will show:

LDA #7A means: load the accumulator with 7A,

LDX #3B means: load the X Register with 3B. In this instance the Y register could have been used in which case the instruction would be LDY #3B.

ADC # (byte) means: add the contents of the following memory location to the existing contents of the accumulator with carry. This can also be expressed as:  $A + M + C \rightarrow A$ . Whether or not the carry is added to the result depends on the state of the carry flag. This flag must always be reset prior to addition, with the CLC instruction (clear carry) for instance. The program so far looks like this:

CLC clear carry (C = 01)

LDA 13 load accumulator with 13

ADC 08 add 08 to it

BRK stop as soon as the addition is complete.

The last step is needed to make the program complete and to inform the computer that the desired task has in fact been carried out.

To execute this program on the JC we first need to determine the instruction codes (see table at the back of this book). We find that CLC = 18; LDA = A9; ADC = 69; BRK = 00. A suitable start address would be 0100 and off we go (power on, display on, STEP switch off):

				adres:	display:	
RST	AD			xxxx	xx	
0	1	0	0	0100	xx	
DA		1	8	0100	18	CLC
+		A	9	0101	A9	LDA #
+		1	3	0102	13	
+		6	9	0103	69	ADC #
+		0	8	0104	08	
+		0	0	0105	00	BRK
AD				0105	00	
1	A	7	E	1A7E	xx	
DA		0	0	1A7E	00	
+		1	C	1A7F	1C	
AD						

0	1	0	0	0100	18	
GO				0107	xx	programstart
AD				0107	xx	
0	0	F	3	00F3	1B	result

There are a few things that need to be clarified. Why, for instance, the sudden jump from 0105 to 1A7E? And why is the result held in address 00F3? When the microprocessor encounters a break instruction it will jump back to the monitor program at the point indicated by the contents of 1A7E and 1A7F, in this case 1C00. This section of the monitor program contains a 'save' routine which ensures that the current contents of each register are loaded into specific (RAM) memory locations. The result is as follows:

Address 00EF contains the contents of PCL,  
 address 00F0 contains the contents of PCH,  
 address 00F1 contains the contents of P,  
 address 00F2 contains the contents of S,  
 address 00F3 contains the contents of A,  
 address 00F4 contains the contents of Y, and  
 address 00F5 contains the contents of X.

The accumulator contents are held in 00F3. As the accumulator held the result of the addition. in this case 1B, 00F3 is the place to find the answer. Subtraction of numbers is also possible (not so with some smaller micro-processors). In this instance the useful instruction is: SBC # (byte) which means: subtract memory from accumulator with borrow (= A - M - C → A). It should be remembered that the JC uses the two's-complement method of subtraction (see chapter 2). In order to obtain the correct result, the carry flag has to be set, thus C = 1, C̄ = 0. This can be accomplished by the instruction SEC (set carry flag). The program to subtract two numbers looks like this:

```
SEC      (op-code 38)
LDA # 13 (op-code A9)
SBC # 08 (op-code E9)
BRK     (op-code 00)
```

As before we can use 0100 as the start address. The program can be entered via the keyboard as follows:

key				address	data	
RST	AD			xxxx	xx	
0	1	0	0	0100	xx	
DA		3	8	0100	38	SEC
+		A	9	0101	A9	LDA #
+		1	3	0102	13	
+		E	9	0103	E9	SBC #
+		0	8	0104	08	
+		0	0	0105	00	BRK

AD				0105	00	} see note
1	A	7	E	1A7E	xx	
DA		0	0	1A7E	00	
+		1	C	1A7F	1C	
AD						
0	1	0	0	0100	38	program start
GO				0107	xx	
AD				0107	xx	
0	0	F	3	00F3	0B	result

Note: Addresses 1A7E and 1A7F do not need to be loaded if they still contain the data from the previous program example, that is, provided the power switch has not been turned off in the meantime.

As before, address 00F3 contains the result of the subtraction: 13 (decimal 19) - 08 = 0B (decimal 11).

### Logic Instructions

The processor is not only capable of performing arithmetic operations, but it is also capable of implementing logic functions. The OR-function is one well known logic operation and can be implemented by using the instruction: ORA # (byte) which means: OR memory with accumulator - AVM → A. The op-code for this instruction is 09. A short example will clarify what it does:

LDA # AA load the accumulator with AA

ORA # 0F perform the OR-function bit by bit with 0F

BRK stop

AA in hexadecimal = 10101010

0F in hexadecimal = 00001111

result after OR = 10101111 (AF)

Whenever a particular bit in the accumulator OR the memory location (second half of the instruction) is a '1' the result will be '1'. When *both* bits are '0' the result will be '0'. Again, the result will be held in the accumulator. Figure 2a illustrates the principle via hardware (OR gates).

Another common logic operation is the AND function. This is implemented with the instruction AND (A ∧ M → A). A short program:

LDA # AA load the accumulator with AA

AND # 0F AND each bit with 0F

BRK stop

AA in hexadecimal = 10101010

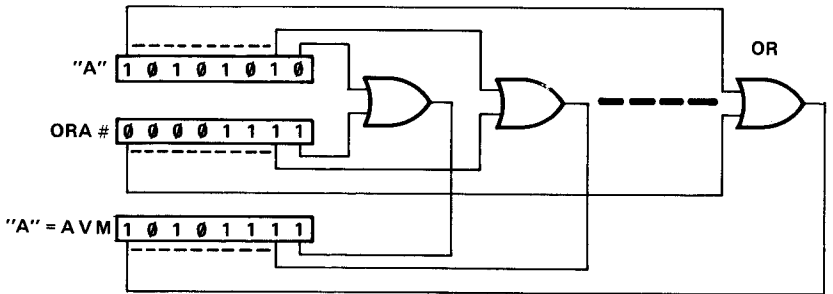
0F in hexadecimal = 00001111

result after AND = 00001010 (0A)

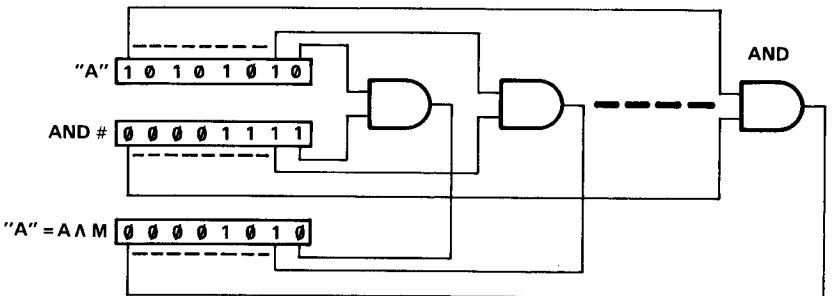
Where the bits in both the accumulator and memory are '1' the result will be '1', when *either* of the bits are '0' the result will be '0'. Figure 2b illustrates what happens with an AND gate.

The third logic function to be considered is the EXclusive OR (EXOR) function. The instruction EOR can be represented by: A ∨ M → A. The

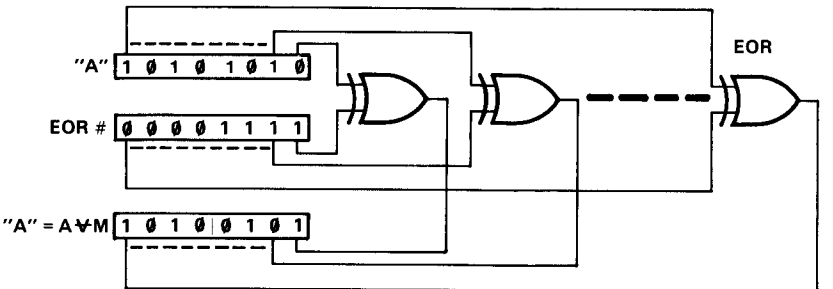




80915-3-2a



80915-3-2b



80915-3-2c

**Figure 2. The operation of the instructions ORA #, AND #, and EOR # is shown here symbolically by the use of logic gates.**

op-code is 49 . . . .

LDA #AA load accumulator with AA

EOR #0F EXOR each bit with 0F

BRK stop

AA in hexadecimal = 10101010

0F in hexadecimal = 00001111

result after EXOR = 10100101 (A5)

When both bits are the same the result will be '0', otherwise the result is a '1'. It can be seen that this instruction can be used to invert certain bits in the accumulator. Figure 2c illustrates the EXOR function. The three given examples can be combined to form a single program. This is keyed in as follows:

key				address	data		
RST	AD			xxxx	xx		
0	1	0	0	0100	xx		
DA		A	9	0100	A9	LDA #	start 1
+		A	A	0101	AA		
+		0	9	0102	09	ORA #	
+		0	F	0103	0F		
+		0	0	0104	00	BRK	end 1
+		A	9	0105	A9	LDA #	start 2
+		A	A	0106	AA		
+		2	9	0107	29	AND #	
+		0	F	0108	0F		
+		0	0	0109	00	BRK	end 2
+		A	9	010A	A9	LDA #	start 3
+		A	A	010B	AA		
+		4	9	010C	49	EOR #	
+		0	F	010D	0F		
+		0	0	010E	00	BRK	end 3
AD							
0	1	0	0	0100	A9	start address 1	
GO				0106	AA	stop 1	
AD							
0	0	F	3	00F3	AF	result 1	
AD							
0	1	0	5	0105	A9	start address 2	
GO				010B	AA	stop 2	
AD							
0	0	F	3	00F3	0A	result 2	
AD							
0	1	0	A	010A	A9	start address 3	
GO				0110	xx	stop 3	
AD							
0	0	F	3	00F3	A5	result 3	

As there are three 'breaks' in the program all three can be run one after the other. Be sure to enter the correct start address before pressing 'GO'. The three logic instructions have very important functions. They can be used to 'mask out' certain bits in a byte and leave the remainder unaltered.

*So far we have dealt with immediate instructions. We have covered eight of them up to now: LDA #, LDX #, LDY #, ADC #, SBC #, ORA #, AND #, and EOR #. We have also learned about the instructions CLC, SEC, and BRK. These will be covered in greater detail in the section describing implied addressing.*

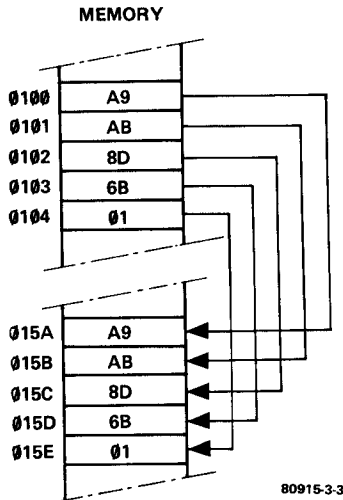
### Absolute addressing

An instruction that uses absolute addressing directly specifies the address of the memory location to be referred to. The two bytes following the opcode will contain the actual address of the memory location concerned. Let us simply look at another program example:

```
LDA-0100 load accumulator with the contents of 0100
STA-015A store accumulator contents in 015A
LDA-0101 load accumulator with the contents of 0101
STA-015B store accumulator contents in 015B
LDA-0102 load accumulator with the contents of 0102
STA-015C store accumulator contents in 015C
LDA-0103 load accumulator with the contents of 0103
STA-015D store accumulator contents in 015D
LDA-0104 load accumulator with contents of 0104
STA-015E store accumulator contents in 015E
```

Note: The hyphen between the instruction and the address location shows that absolute addressing is being used.

The end result of this program is that the contents of address locations 0100 . . . 0104 have been copied into locations 015A . . . 015E. The



**Figure 3.** A 'memory map' of the result of copying data from one memory area to another.

accumulator was used as a 'go-between'. The first four memory locations still contain their original information – the data was copied not transferred. A memory map of the result is shown in figure 3.

The instruction LDA should now be familiar, but it is shown here without the # symbol. On the other hand, the instruction STA – is new. It is used to store the contents of the accumulator in a specific memory location (A → M). All absolute address instructions require three bytes; the first for the instruction code, the second for the low order address byte (ADL; L = Low) and the third for the high order address byte (ADH; H = High).

Let us take a look at a program where two sixteen bit numbers are added together using absolute addressing. Both numbers consist of two bytes (HOB = High Order Byte, LOB = Low Order Byte). The result will again be a sixteen bit number, but there may have to be a carry added to the extreme left hand side (MSB).

Before 'jumping' into the actual program it may be advisable to examine the flow chart shown in figure 4. As can be seen, six memory locations (0100 . . . 0105) are reserved for the two sixteen bit numbers and the

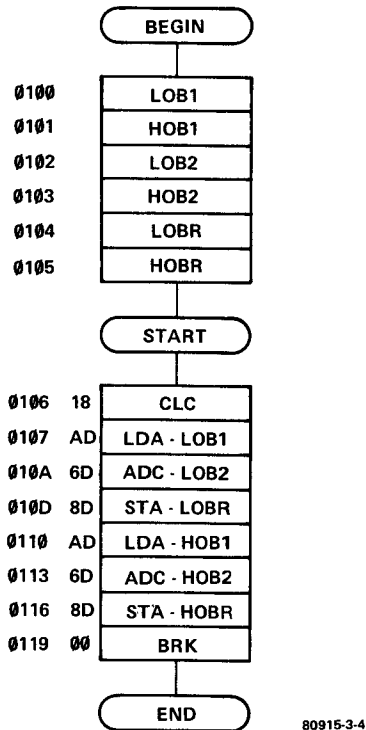


Figure 4. Flow chart of the program to add two 16-bit numbers together and store the result.

result. The actual program does not start until address 0106 and continues up to and including 0119.

The two numbers we are going to add together are 04EF and 23AB:

(STEP: OFF; DISPLAY: ON)				address	data		
RST	AD			xxxx	xx		
1	A	7	A	1A7A	xx		
DA		0	0	1A7A	00	} enable STEP routine	
+		1	C	1A7B	1C		
++				1A7D	xx		
+		0	0	1A7E	00	} enable BRK routine	
+		1	C	1A7F	1C		
AD				1A7F	1C		
0	1	0	0	0100	xx		
DA		E	F	0100	EF	LOB1	
+		0	4	0101	04	HOB1	
+		A	B	0102	AB	LOB2	
+		2	3	0103	23	HOB2	
+				0104	xx	reserved for LOBR	
+				0105	xx	reserved for HOBR	
+		1	8	CLC	0106	18	clear carry-flag
+		A	D	LDA-	0107	AD	} LOB1 in A
+		0	0		0108	00	
+		0	1		0109	01	
+		6	D	ADC-	010A	6D	} A+LOB2→A (including carry-flag)
+		0	2		010B	02	
+		0	1		010C	01	
+		8	D	STA-	010D	8D	} result (= LOBR) to address 0104
+		0	4		010E	04	
+		0	1		010F	01	
+		A	D	LDA-	0110	AD	} HOB1 in A
+		0	1		0111	01	
+		0	1		0112	01	
+		6	D	ADC-	0113	6D	} A+HOB2→A (including carry-flag)
+		0	3		0114	03	
+		0	1		0115	01	
+		8	D	STA-	0116	8D	} result (= HOBR) to address 0105
+		0	5		0117	05	
+		0	1		0118	01	
+		0	0	BRK	0119	00	end of program
AD							
+	1	0	6		0106	18	start address

GO				011B	xx	program running
AD						
0	1	0	4	0104	9A	result: LOBR
+				0105	28	result: HOBR

Now let us examine the program in greater detail. The two numbers to be added were:

04EF: hexadecimal 00000100 11101111

23AB: hexadecimal 00100011 10101011

← HOB → ← LOB →

Firstly, the carry flag is reset at address 0106. After loading the accumulator with LOB1 and adding LOB2 to it the carry flag is set, thus:

	11101111	LOB1
	10101011	LOB2
	+ 111 1111	carry
	<u>1</u> 10011010	LOBR
carry	←9→	←A→

The carry flag will remain set even after storing LOBR (in address 0104) and loading HOB1 into the accumulator. When HOB2 is added, however, the carry flag will be reset once more:

	00000100	HOB1
	00100011	HOB2
	1	carry from LOBR
	+ 111	carry
	<u>0</u> 00101000	HOBR
carry	←2→	←8→

The result of the addition can be seen in locations 0104 and 0105.

Note: The comments at addresses 1A7A . . . 1A7F refer to interrupt routines contained in the monitor program. These will be dealt with in greater detail later on.

Most of the instructions in the last program used absolute addressing. The three bytes of these instructions are easily recognisable. There are, of course, many other instructions that use absolute addressing. The following is a list of the more commonly used ones:

Memory reference instructions:

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

Arithmetic instructions:

ADC- op-code 6D (A+M+C → A) add memory to accumulator  
 with carry  
 SBC- op-code ED (A-M- $\bar{C}$  → A) subtract memory from accumulator  
 with borrow

INC- op-code EE (M + 1 → M) increment memory by one  
 DEC- op-code CE (M - 1 → M) decrement memory by one  
 The last two instructions simply cause the memory contents to be increased (INC-) or decreased (DEC-) by one (00000001).

Logic instructions:

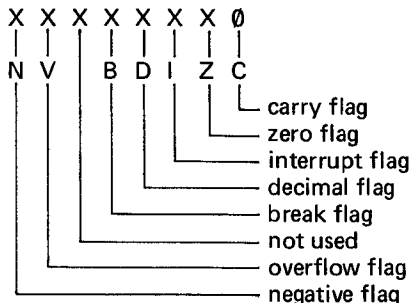
ORA- op-code 0D (A ∨ M → A) OR memory with accumulator  
 AND- op-code 2D (A ∧ M → A) AND memory with accumulator  
 EOR- op-code 4D (A ⊕ M → A) Exclusive OR memory with accumulator

## Step by step programming

There are two ways of running a program on the Junior Computer. The first, and most obvious, is to enter the start address of the program and press the 'GO' key (with the STEP switch in the off position). The program will then run until the processor encounters a BRK instruction. The other method is to place the STEP switch in the on position. This will light the LED in the STEP/GO key and enable the program to be run 'instruction by instruction', provided that address locations 1A7A and 1A7B contain 00 and 1C respectively. Each time the STEP/GO key is operated the next instruction in the program will be executed. This is illustrated in the memory map of figure 5.

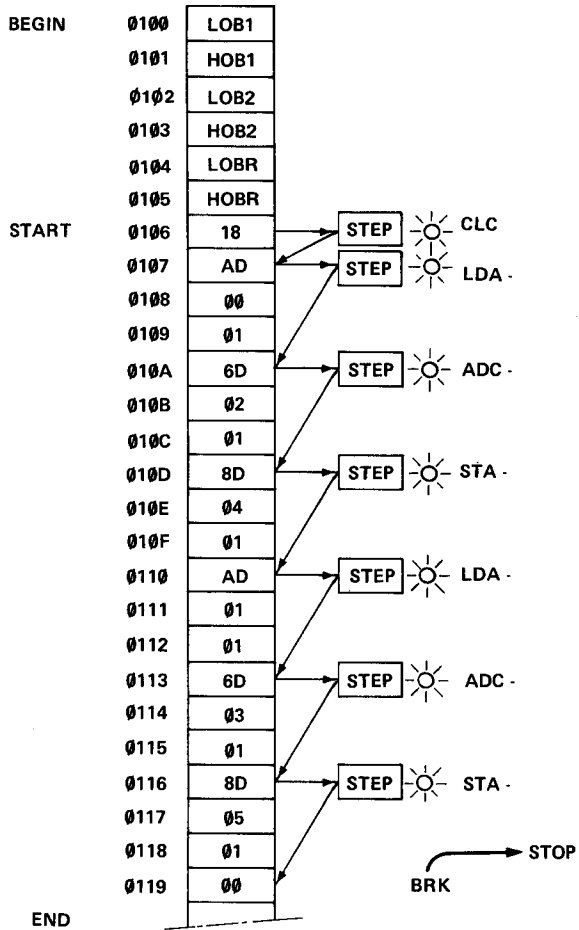
Once the start address (0106) has been set up the STEP/GO key is depressed and the first instruction is carried out. In this case the instruction is to clear the carry bit. To make sure that the carry flag has in fact been reset we can examine the P register. As mentioned previously, the contents of the status register are also held in address 00F1. By pressing the AD key followed by this address the display will reflect the contents of the status register.

It may not be quite clear as to what the information on the display actually means, therefore a look at the construction of the status register would not be amiss.



The flags that are of no importance to us at the moment are indicated by an 'X'. The carry flag, however, is the least significant bit in the register (extreme right). It is therefore very simple to tell whether the carry flag is set or reset as the number on the right-hand display will be even if the flag is zero and odd if the flag is set.

Going back to stepping through the program, if the PC key is operated the display will show 0107 AD. As we know, AD is the op-code for LDA-, the



80915-3-5

Figure 5. The program of figure 4 shown as a memory map and operated by using the STEP function.

next instruction in the sequence. Pressing the STEP/GO key will now produce 010A 6D on the display. This shows that the LDA-instruction has been executed and the value of LOB1 (EF) should be held in the accumulator. Again, a quick check will verify that this is indeed the case. The complete program can be stepped through in this manner and it can be seen that this mode of operation is a useful aid to 'debugging' and at the same time a practical 'tool' for educational purposes.



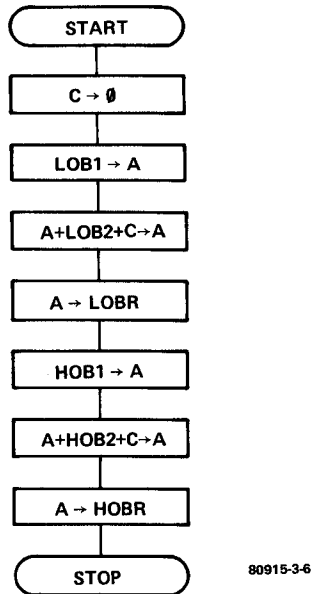


Figure 6. The 'rough' (but complete) flow chart of the program shown in figure 4.

### Zero page addressing

Here we discuss a special form of absolute addressing. In this case, the instruction has only two bytes of object code. The high order address byte (ADH) of zero page instructions is always zero. The addressable range, therefore, runs from 0000 to 00FF. These 256 different locations belong to 'page zero'. The following 256 bytes belong to page one etc. This leads us to figure 7 which shows the page address structure of the Junior Computer.

Pages 0 . . . 3 contain the RAM, or work memory. Page 1A belongs to the PIA and is divided into RAM, I/O addressing and the timer. Pages 1C . . . 1F are reserved for the monitor program. In the standard JC model only pages 00 . . . 1F can be addressed because of the incomplete address decoding. This means that the highest addressable memory location is 1FFF.

As the computer knows that the high order byte of every zero page instruction is 00, we can effectively save one memory location out of three when using absolute addressing.

The mnemonics used for zero page instructions are the same as those used for absolute addressing, but to indicate the difference they are appended with a Z. By now, they should look quite familiar:

Memory reference instructions:

LDAZ op-code A5 (M → A) load accumulator with memory

LDXZ op-code A6 (M → X) load index X with memory

LDYZ op-code A4 (M → Y) load index Y with memory

STAZ op-code 85 (A → M) store accumulator in memory

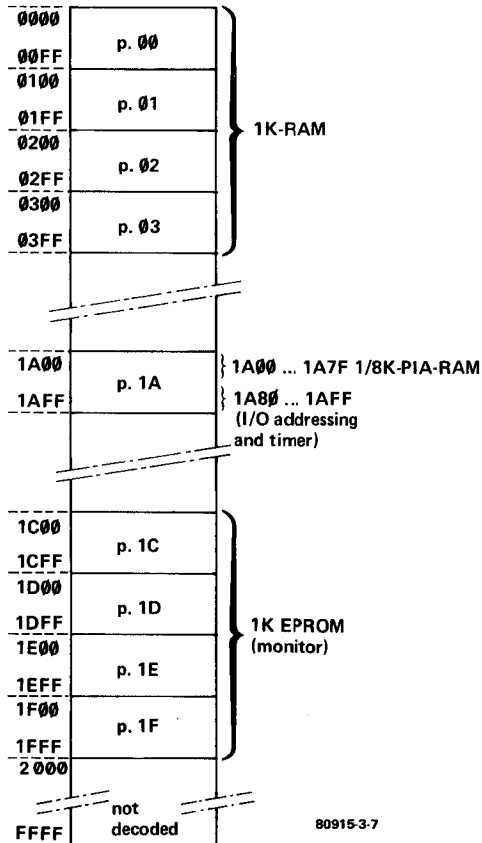


Figure 7. The page address structure of the Junior Computer. The left-hand (high order) bytes of addresses in a page are always the same.

STXZ op-code 86 ( $X \rightarrow M$ ) store index X in memory

STYZ op-code 84 ( $Y \rightarrow M$ ) store index Y in memory

Arithmetic instructions:

ADCZ op-code 65 ( $A+M+C \rightarrow A$ ) add memory to accumulator with carry

SBCZ op-code E5 ( $A-M-C \rightarrow A$ ) subtract memory from accumulator with borrow

INCZ op-code E6 ( $M+1 \rightarrow M$ ) increment memory by one

DECZ op-code C6 ( $M-1 \rightarrow M$ ) decrement memory by one

Logic instructions:

ORAZ op-code 05 ( $A \vee M \rightarrow M$ ) OR memory with accumulator

ANDZ op-code 25 ( $A \wedge M \rightarrow M$ ) AND memory with accumulator

EORZ op-code 45 ( $A \nabla M \rightarrow M$ ) Exclusive OR memory with accumulator

## Relative addressing

This address mode is only used for 'branch' instructions, of which there are two different types: conditional and unconditional. Unconditional branch instructions always cause a jump whereas with conditional branch instructions there are always certain things to take into account first. These decisions arise from the flow chart that is made when developing a program. The majority of flow chart symbols are converted into actual instructions along the lines of: if flag X is set jump to subroutine A, if flag Y is reset increment counter etc.

Conditional branch instructions have two bytes of object code. The second byte is treated as an 8-bit, signed binary number, which is added to the program counter after the PC contents have been incremented to address the next program instruction. The following is a list of the conditional branch instructions:

1. BCC op-code 90 Branch if Carry Clear (C = 0)  
BCS op-code B0 Branch if Carry Set (C = 1)
2. BNE op-code D0 Branch if Not Equal to zero (Z = 0)  
BEQ op-code F0 Branch if Equal to zero (Z = 1)
3. BPL op-code 10 Branch if Plus (N = 0)  
BMI op-code 30 Branch if Minus (N = 1)
4. BVC op-code 50 Branch if overflow Clear (V = 0)  
BVS op-code 70 Branch if overflow Set (V = 1)

These instructions can now be used to develop programs. The unconditional branch instructions will be discussed later in this chapter.

Let us take a look at the flow chart in figure 8. At the start of the program (0200) the Y register is loaded with the value 0A. The next instruction is DEY, which reduces the value in the Y register by one – thus 0A becomes 09. Moving down the flow chart we come to a conditional branch instruction, BNE. From the above we discover that the microprocessor will only effect a branch if the contents of the Y register are not equal to zero. At present time the Y register contains 09 therefore the processor *will* branch back to the address containing the DEY instruction. As soon as the value in the Y register becomes zero the processor will stop branching and the program will stop as the last instruction is BRK. This program does nothing more than continually decrement the contents of the Y register until the value contained there becomes zero. Routines such as this are often used as 'delay loops'.

On the left-hand side of the diamond symbol containing the branch instruction there are two hexadecimal numbers, D0 and FD. The first is of course the instruction while the second (FD) is the displacement value which is used to calculate the effective address. If the program is to branch backwards (as in this case) the displacement value will be negative. A forward branch requires a positive displacement value.

The hexadecimal equivalent of FD is 11111101 which is, in two's complement notation, -3. As the displacement is negative the branch will be three memory locations backwards (calculated from the address immediately following the one containing the displacement value (0205 - 3 = 0202). This may become a bit clearer from the following:

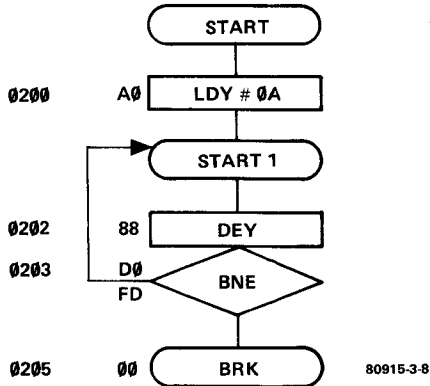


Figure 8. Example of a 'delay loop' using a conditional branch instruction.

```

0200 A0 LDY
0201 0A
0202 88 DEY
0203 D0 BNE
0204 FD displacement value
0205 00 BRK

```

(You may not have realised it, but we have just discovered a new instruction — DEY — reduce the contents of the Y register by one).

The effective address is calculated from the location *following* the displacement value as this is where the program counter will be pointing once the program has reached address 0204. Remember that the program counter is incremented *before* the next instruction is carried out.

The effective address can be anything up to a maximum of 127 steps forward (+127: hexadecimal 00 . . . 7F) or 128 steps backwards (–128: hexadecimal FF . . . 80). This provides all 256 possibilities of a single byte.

### Calculating displacements with the monitor program

There are two important aspects to take into account when calculating displacements for branch instructions: where the branch originates and where it must end. This is quite easily seen on a flow chart such as the one in figure 8, but when there are numerous branch instructions in a program it is less work to let the JC perform all the calculations.

Using figure 8 as an example, this can be carried out as follows:

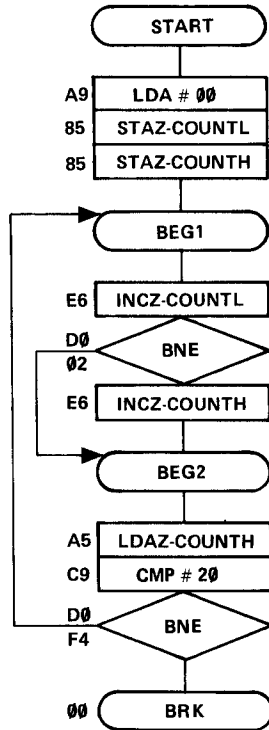
AD				xxxx	xx
1	F	D	5	1FD5	D8
GO				0000	00
0	3	0	2	0302	FD — displacement
RST					

First, the address 1FD5 is keyed in. This is the start address of the monitor-

displacement-routine BRANCH. Once the start address has been entered, the GO key is pressed. The low order byte of the address where the branch instruction is stored can then be keyed in (03), immediately followed by the low order byte of the destination address (02). The two address bytes will appear on the left-hand side of the display and the data displays will show the calculated displacement (FD). The reset key can now be pressed and the program from figure 8 can be entered:

AD						
0	2	0	0	0200	xx	
DA		A	0	0200	A0	LDY #
+		0	A	0201	0A	
+		8	8	0202	88	DEY
+		D	0	0203	D0	BNE
+		F	D	0204	FD	displacement
+		0	0	0205	00	BRK

The program can be run as normal, but it may be interesting to see what happens in the STEP mode!



COUNTL = 0000  
 COUNTH = 0001

80915-3-9

Figure 9. Flow chart of the 'software counter' program.

## A software counter

With all this new knowledge of branch instructions, we are now able to construct a counter without picking up a soldering iron! In the process we will also become familiar with a new instruction, CMP.

This particular counter will start from zero and stop as soon as it reaches its preset limit of 2000 (hexadecimal). The flow chart for the program is shown in figure 9. Two bytes are required for the contents of the software counter and these are stored (using zero page addressing) in locations 0000 (COUNTL) and 0001 (COUNTH).

Initially, the accumulator is loaded with 00 and the counter is 'reset' (00 stored in both COUNTL and COUNTH). Following this the contents of COUNTL are increased by one and we reach the first branch instruction (BNE). This instruction tests the state of the zero flag (Z) and, as the Z-flag is high, the program will branch to BEG2.

The accumulator is then loaded with the contents of COUNTH which is then compared, via the CMP instruction, with the value 20 (the preset final value for the high order byte). This new instruction (op-code C9) sets the zero flag if the contents of the accumulator (COUNTH) and the contents of the following byte (20) are the same. Initially, of course, the value in COUNTH is still zero. Therefore the program will continue to the second branch instruction and from there jump back to BEG1. When the contents of COUNTH reach 20 the zero flag will be set, no branch will take place, and the program will stop.

In other words, COUNTL becomes zero after every 256 cycles. This increases the value of COUNTH by one until its value reaches 20. The following is the actual program:

AD											
1	F	D	5	1FD5	D8	start address of					
GO				0000	00	displacement routine					
1	8	1	C	181C	02	displacement for first BNE					
2	0	1	6	2016	F4	displacement for second					
						BNE					
RST	AD										
0	2	1	0	0210	xx	start address of program					
DA		A	9	0210	A9	LDA #					
+		0	0	0211	00						
+		8	5	0212	85	STAZ-					
+		0	0	0213	00						
+		8	5	0214	85	STAZ-					
+		0	1	0215	01						
+		E	6	0216	E6	INCZ-					
+		0	0	0217	00						
+		D	0	0218	D0	BNE					
+		0	2	0219	02	displacement					
+		E	6	021A	E6	INCZ-					

+	Ø	1	8	Ø21B	<sup>1</sup> Ø1	
+	A	5	7	Ø21C	<sup>2</sup> A5	LDAZ-
+	Ø	1	6	Ø21D	Ø1	
+	C	9	5	Ø21E	C9	CMP #
+	2	Ø	4	Ø21F	2Ø	operand CMP #
+	D	Ø	3	Ø22Ø	DØ	BNE
+	F	4	2	Ø221	F4	displacement
+	Ø	Ø	1	Ø222	ØØ	BRK
AD				Ø222	xx	
Ø	2	1	Ø	Ø21Ø	A9	start address
GO				Ø212	xx	run

Initially, the displacement values for both branch instructions were calculated by using the monitor program. This produced the two values F4 and Ø2. It is not strictly necessary to carry out this procedure each time (as we supply the answer anyway), but it does mean that you become more familiar with the keyboard and the monitor program.

### Unconditional branch instructions

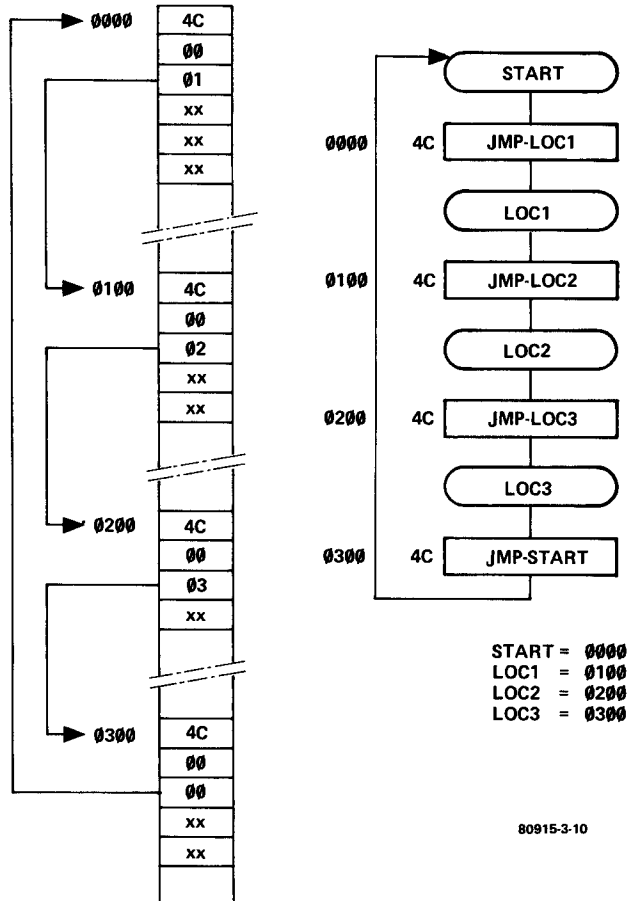
As their name implies, no conditions have to be met for these instructions to be carried out. The first unconditional branch instruction to be described is the JMP-instruction. This uses absolute (look out for that hyphen) or (still to be covered) indirect addressing. When absolute addressing is specified the op-code is 4C. It is, therefore, a three byte instruction the last two bytes of which contain the address location to which the program must jump.

Figure 10 shows a memory map and a flow chart of a simple program containing nothing but unconditional jump instructions. The program will jump from START to locations 1, 2 and 3 in turn before jumping back to START. It may seem a little pointless, but it is all serious work.

### JSR and RTS – jumping to and from subroutines

Two more unconditional branch instructions are: JSR, Jump to SubRoutine (op-code 2Ø) and RTS, ReTurn from Subroutine (op-code 6Ø). Subroutines form a very important part of any (complex) program. If an operation, or series of operations, has to be performed a number of times during a program it only has to be entered once – as a subroutine.

The JSR-instruction uses absolute addressing to jump to the start of the subroutine and once the operation has been performed the computer will jump back to the main program exactly where it left off (provided, of course, the subroutine ends with the RTS instruction). This is illustrated in figure 11. The subroutine is contained in address locations 1A21 . . . 1A3B, and it is being used twice in the main program. When the program reaches address Ø223 it discovers the first JSR instruction. The following two locations, Ø224 and Ø225, contain the low order byte (ADL) and the



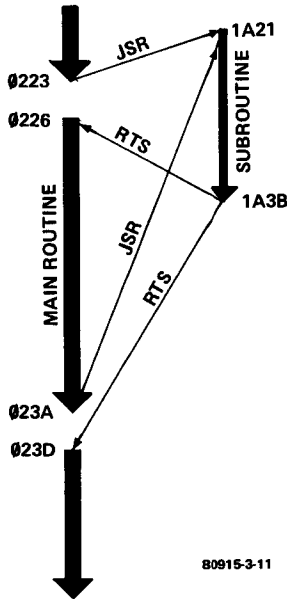
**Figure 10. A simple program to demonstrate the use of unconditional jump instructions.**

high order byte (ADH) of the start address of the subroutine. Before actually jumping to the subroutine the contents of the program counter (0225) are stored in a 'stack'. At the end of the subroutine the contents of the stack are replaced in the program counter, the program counter is incremented and the program will continue from this point (0226). When the main program reaches the second JSR instruction at address 023A it will again jump to the subroutine and return to address 023D.

### Stack and stack pointer

It is possible to jump from one subroutine to another, and from there to





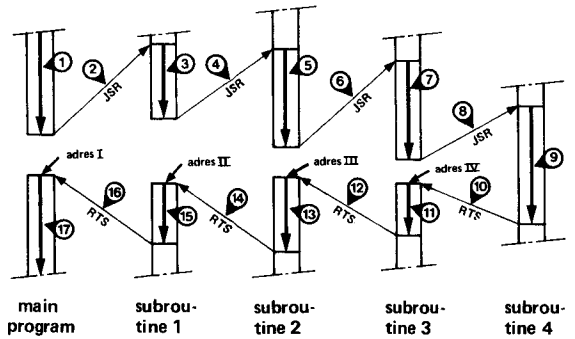
**Figure 11.** This figure illustrates the use of jump to and return from subroutine instructions. A subroutine can be 'jumped to' from any part of the main program.

another and so on. This process can be likened to 'Russian Dolls' where one doll fits inside a slightly larger one and they fit inside a slightly larger one again etc. When referring to computers and subroutines this process is called 'nesting'.

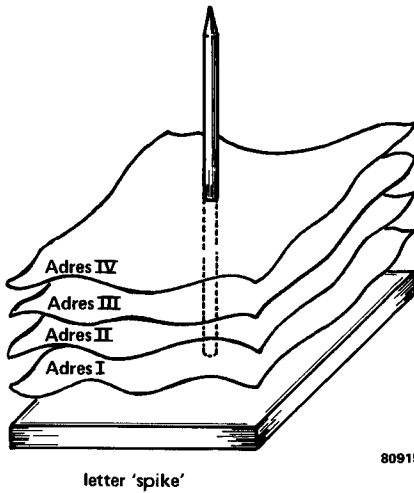
As with the Russian Dolls, there is a limit to how many times this can be done. This limit is called the 'nest depth'. Another important thing to remember when nesting subroutines is that for every JSR instruction there *must* be an equivalent number of RTS instructions. If the number of instructions is not equal, the computer will stop at the end of a subroutine and the program will 'crash'.

For every jump instruction there has to be a return instruction, and for every jump address there has to be a return address. The computer itself keeps a record of all the return addresses by means of the stack. This is an area of memory where the return addresses (two bytes per address) are arranged in a certain order. The return addresses are placed on the stack in the order they are encountered and they are removed in the reverse order (last in, first out).

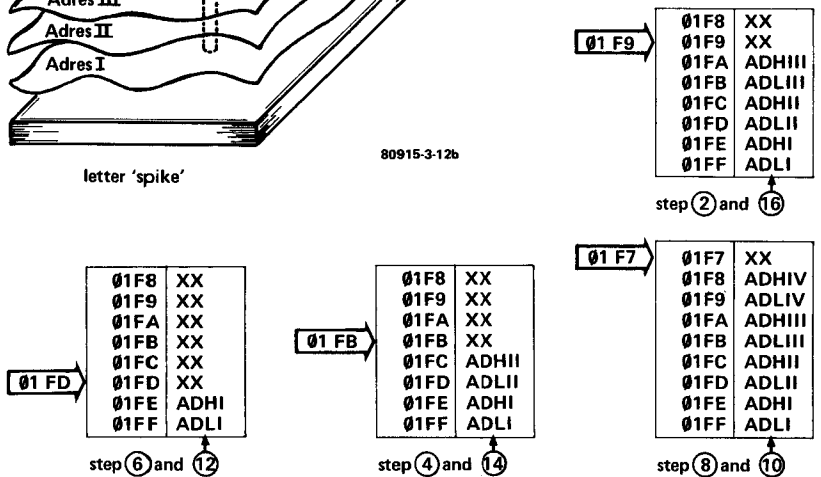
Figure 12 illustrates the nesting of subroutines. The program jumps to each of the four subroutines in turn, and the last one (subroutine 4) is completed first. The remaining three subroutines are completed in the reverse order before the computer returns to the main program. This can be likened to the 'letter spike' shown in figure 12b. The sheets of paper have to be removed in the reverse order that they were placed on the



80915-3-12a



80915-3-12b



80915-3-12c

Figure 12. The nesting of subroutines (12a) can be likened to the use of a letter 'spike' (12b). The memory map (12c) shows the contents of the stack and stack pointer for each jump and return instruction.

'spike'. It can also be seen from figure 12a that the nest depth is almost unlimited.

The Junior Computer uses the 256 memory locations of page one as the stack. This means that up to 128 return addresses can be stored between 01FF and 0100 (01FF is the 'bottom' of the stack).

Also in figure 12, a memory map shows that the return addresses are loaded onto the stack from bottom to top (compare this with figure 12b!). This means that the first return address is loaded into memory locations 01FF and 01FE. The low order address byte is loaded into location 01FF and the high order address byte into 01FE. We can also see that the stack pointer points to the first empty location in the stack. The only thing that is not shown is that the return address is determined by the contents of the program counter before the jump to the subroutine. This can be seen, however, in the program shown in figure 13.

The program runs straight through (that is without subroutines) up to and including address 0215. When it reaches address 0216 the op-code (20) for the JSR instruction is found. The start address of the subroutine is then

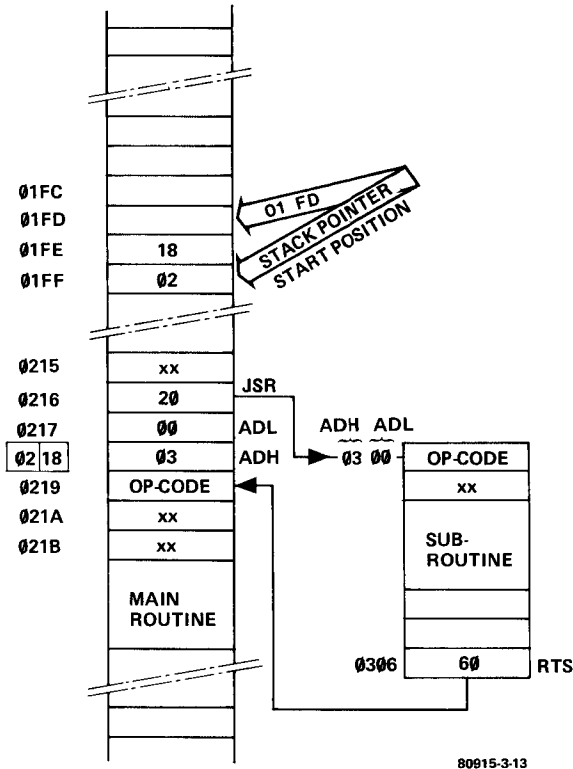


Figure 13. Section of program illustrating the use subroutines and the function of the stack and stack pointer.

found in locations 0217 (ADL = 00) and 0218 (ADH = 03). The present contents of the program counter are then 'pushed' onto the stack in page 1 of memory. Therefore, the low order byte, 02, will be loaded into address 01FF and the high order byte, 18, will be loaded into address 01FE. The computer will then jump to the start address of the subroutine (0300) and continue from there until the instruction RTS (op-code 60) is discovered in address location 0306. This instruction utilises 'implied' addressing (still to be covered) which means that the return address will be found on the top of the stack. This return address is then retrieved from the stack and replaced in the program counter. Before jumping back to the main program, however, the contents of the PC are incremented by one. The actual return address, therefore, becomes the stack address plus one!

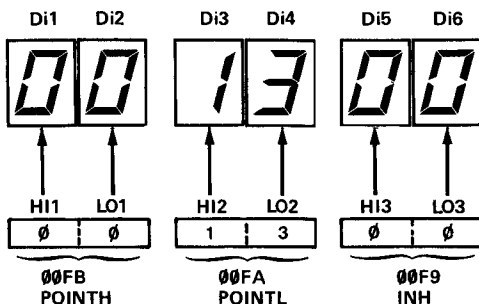
### Some more fingerwork

We feel that it is high time to start entering things into the computer and actually see these things happening. In the process we will also learn about a few interesting 'tricks' contained in the monitor program.

Firstly, we want to develop a program that displays the value of each of the keyboard switches in hexadecimal form. The various keys will be assigned the following values:

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

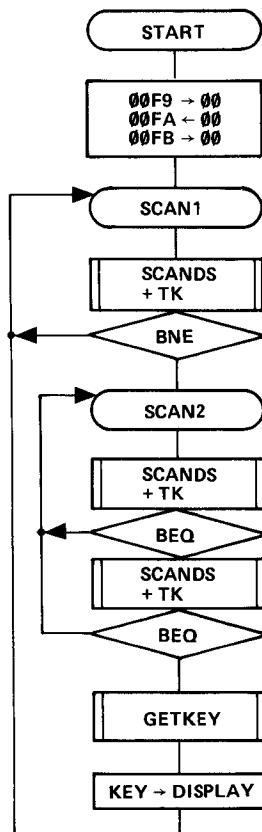
The values assigned to keys 0 . . . F are quite straightforward, the rest are more or less arbitrary. The object of the exercise is to produce the corresponding hexadecimal value of a particular key on the two centre displays. Before we actually enter the program to do this, it is worthwhile taking a closer look at the monitor program. This contains a short routine called SCANDS which displays the contents of address locations 00F9, 00FA



80915-3-14

Figure 14. The contents of three display 'buffers' can be displayed as shown here by utilising a routine called SCANDS contained in the monitor program.

and 00FB as shown in figure 14. Each display will show half of the particular byte. The routine SCANDS first obtains the four most significant bits of address 00FB, converts them into seven-segment code and passes this information to the left-hand display. The four least significant bits of the same byte are similarly decoded and this information is passed on to the second display. The contents of locations 00FA and 00F9 are indicated on the remaining four displays in the same manner. The next thing is to determine whether or not a key is being depressed. This can be accomplished with the routine TK (Test Key), which is called up by the SCANDS routine. As soon as a key is depressed, its value has to be displayed. Again, the monitor program contains a routine called GETKEY which does exactly this. A rough flow chart for the program is shown in figure 15. As the two centre displays are to show which key is being depressed, the remaining



80915-3-15

Figure 15. Rough flow chart of the program to detect, identify and display the value of a particular key. The program uses subroutines that are contained in the monitor program.

displays must always show 00. Therefore, the first operation is to store 00 in address locations 00FB and 00F9. The next section of the program (SCAN1) contains the SCANDS + TK subroutine with which the contents of locations 00F9, 00FA and 00FB are shown on the display. This subroutine also checks to see whether a key is being depressed. Via the conditional branch instruction, BNE, the program will always jump back to the start of SCAN1 if no key is pressed. As soon as a key is depressed, however, the program will carry on to SCAN2. This section of the program also contains the subroutine SCANDS + TK, but this time the program will check to see whether the key has been released. This check is actually carried out twice as a form of 'software debouncing'. The last section of the program contains the GETKEY subroutine. This routine ensures that the key is given the correct hexadecimal value which is then stored in address location 00FA. The program then jumps back to SCAN1 and waits until another key is depressed. A more detailed flow chart is shown in figure 16. As can be seen, there is

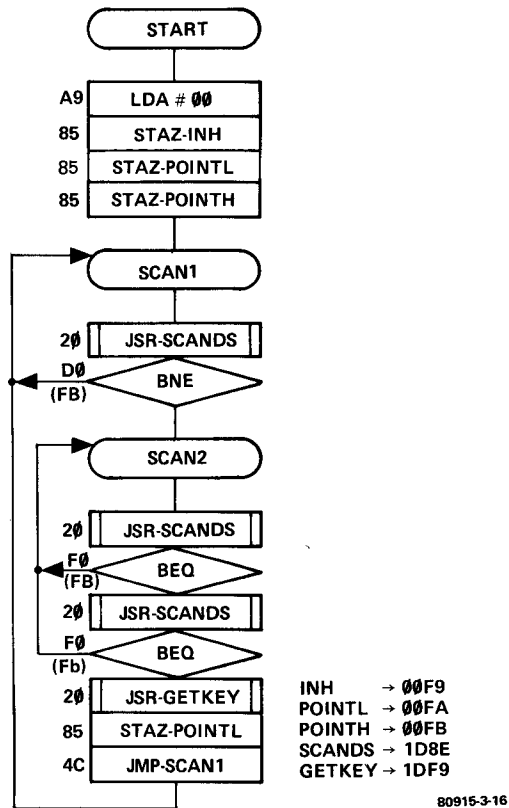


Figure 16. Detailed flow chart of the program given in figure 15.

not that much difference between this and the rough flow chart. The correct instructions, together with their op-codes, have been included next to the relevant symbols. The 'return from subroutine' instructions are not included in the main program, they are handled by the monitor program. As always, before entering the program, the displacement values have to be calculated (and noted!):

AD				xxxx	xx	
1	F	D	5	1FD5	D8	
GO				0000	00	
0	B	0	8	0B08	<span style="border: 1px solid black; padding: 2px;">FB</span>	→ note displacement
1	0	0	D	100D	<span style="border: 1px solid black; padding: 2px;">FB</span>	→ note displacement
1	5	0	D	150D	<span style="border: 1px solid black; padding: 2px;">F6</span>	→ note displacement
RST						
AD						
0	2	0	0	0200	xx	
DA		A	9	0200	A9	LDA #
+		0	0	0201	00	
+		8	5	0202	85	STAZ-
+		F	9	0203	F9	INH
+		8	5	0204	85	STAZ-
+		F	A	0205	FA	POINTL
+		8	5	0206	85	STAZ-
+		F	B	0207	FB	POINTH
+		2	0	0208	20	JSR-
+		8	E	0209	8E	} SCANDS + TK
+		1	D	020A	1D	
+		D	0	020B	D0	BNE
+		F	B	020C	FB	
+		2	0	020D	20	JSR-
+		8	E	020E	8E	} SCANDS + TK
+		1	D	020F	1D	
+		F	0	0210	F0	BEQ
+		F	B	0211	FB	
+		2	0	0212	20	JSR-
+		8	E	0213	8E	} SCANDS + TK
+		1	D	0214	1D	
+		F	0	0215	F0	BEQ
+		F	6	0216	F6	
+		2	0	0217	20	JSR-
+		F	9	0218	F9	} GETKEY
+		1	D	0219	1D	
+		8	5	021A	85	STAZ-

+		F	A	021B	FA	POINTL
+		4	C	021C	4C	JMP-
+		0	8	021D	08	} SCAN 1
+		0	2	021E	02	
AD				021E	02	
0	2	0	0	0200	A9	start address
GO				0000	00	program runs as long as no key is depressed
GO				0013	00	key value GO (!!!)
6				0006	00	key value 6
AD				0010	00	key value AD
DA				0011	00	key value DA
B				000B	00	key value B

and so on – ad nauseum:

RST

Once the start address (0200) has been entered, the GO key is depressed twice. The first time to set the program running, and the second time as an example – the value 13 will appear on the centre displays.

## Implied addressing

The term ‘implied addressing’ is used to describe instructions that identify one of the programmable registers. These are single byte instructions and are used, for instance, to transfer data from one register to another, or to set/reset various flags etc. There are twenty-five instructions of this type (some of which we have already discovered):

BRK op-code 00 BReAK  
 CLC op-code 18 CLear Carry  
 CLD op-code D8 CLear Decimal mode  
 CLI op-code 58 CLear Interrupt flag  
 CLV op-code B8 CLear oVerflow flag  
 DEX op-code CA DEcrement X register by one  
 DEY op-code 88 DEcrement Y register by one  
 INX op-code E8 INcrement X register by one  
 INY op-code C8 INcrement Y register by one  
 NOP op-code EA No OPeration  
 PHA op-code 48 PusH Accumulator onto stack  
 PHP op-code 08 PusH Processor status register onto stack  
 PLA op-code 68 PulL Accumulator from stack  
 PLP op-code 28 PulL Processor status register from stack  
 RTI op-code 40 ReTurn from Interrupt  
 RTS op-code 60 ReTurn from Subroutine  
 SEC op-code 38 SEt Carry flag  
 SED op-code F8 SEt Decimal flag  
 SEI op-code 78 SEt Interrupt flag



TAX op-code AA Transfer Accumulator to X register  
TAY op-code A8 Transfer Accumulator to Y register  
TSX op-code BA Transfer Stack pointer to X register  
TXA op-code 8A Transfer X register to Accumulator  
TXS op-code 9A Transfer X register to Stack pointer  
TYA op-code 98 Transfer Y register to Accumulator

We can now go on to describe them in more detail.

### *SED and CLD*

The 6502 instruction set makes it possible to calculate in decimal as well as in binary. After the instruction SED (SEt Decimal flag) the computer will operate in the decimal mode. This means that the carry flag is set whenever the result of a calculation exceeds 99 (hexadecimal 10011001) and not FF (hexadecimal 11111111). When the decimal flag is reset (with the instruction CLD) the computer will continue to operate in the binary mode.

A useful tip: If the computer is required to operate in the binary mode, it is advisable to start the (section of) program with the instruction CLD. This will ensure correct operation – it is quite easily forgotten that the flag was set elsewhere in the program!

Note: The computer will always be in the binary mode after a reset (RST key) operation.

### *NOP (do nothing)*

No matter how proficient a computer programmer becomes there is always the possibility of leaving out an important instruction in the middle (or more usually at the beginning) of a program. This means, of course, that the program will not run and a certain amount of 'de-bugging' becomes necessary. The wise programmer, however, will always include a proliferation of NOP instructions scattered randomly around the program before the final (working) version is drawn up. By doing this the extra instructions can be inserted where required without having to re-enter the whole program. If, on the other hand, some instructions are found to be superfluous they can be replaced with NOP instructions without affecting the rest of the program. It is not, therefore, such a 'do nothing' instruction as may appear at first sight.

### *Push-Pull-Transfer*

All implied instructions whose mnemonics begin with a P or a T infer that data transfer must take place between the internal registers. There are many reasons why such instructions are necessary. For instance, imagine that we are using the X register in the main program and we then want to jump to a subroutine which also uses the X register. Before jumping to the subroutine the contents of the X register have to be stored in a safe place so that they can be returned to the X register once the subroutine has been completed. The most obvious place to store the contents of the X register is, of course, the stack. The actual procedure could be as follows:

TXA transfer contents of X register to accumulator  
 PHA push contents of accumulator onto stack  
 JSR-xxxx jump to subroutine (and do what you like with the X register)  
 RTS return from subroutine  
 PLA replace the contents of the stack in the accumulator  
 TAX replace contents of accumulator in X register

The following could also have been done:

STX-SAVX store the contents of the X register in memory location SAVX

JSR-xxxx jump to subroutine (X register free for use)

RTS return from subroutine

LDX-SAVX replace the contents of SAVX in the X register

Although the former method looks longer on paper, it actually uses less (instruction) bytes and is therefore preferable. It can be seen that if the contents of the A, X, Y and P registers had to be saved, the second method would be much longer still.

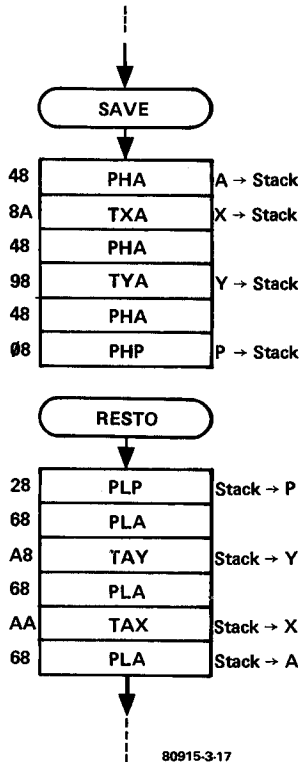


Figure 17. These two subroutines contained in the monitor program save and restore the contents of various registers.

To cut corners even further, the monitor program contains subroutines to perform the above operation. The subroutine SAVE places the contents of the various registers onto the stack and the subroutine RESTO replaces the contents of the stack into the relevant registers.

Figure 17 shows the two subroutines. The SAVE routine first pushes the contents of the accumulator onto the stack followed by the contents of registers X, Y and P (via the accumulator). The RESTO routine does exactly the opposite in that it first replaces the contents of the P register followed by the Y and X registers with the accumulator bringing up the rear.

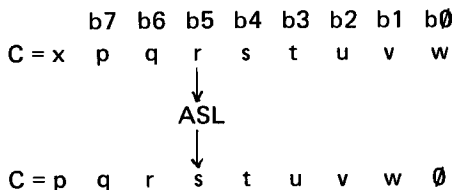
The stack pointer (internal register S) always points to the next following stack address. After calling up the monitor program (by depressing RST) the last address of page 1 (01FF) is automatically pointed to. It is also possible to change the start address of the stack pointer as follows:

```
LDX L 81      load the X register with 81
TXS          transfer the contents of the X register to the stack
```

The stack pointer will now point to address location 0181. Care must be taken when altering the start address of the stack as this could well mean that the program may run out of stack addresses. The availability of stack addresses can be tested by using yet another monitor subroutine, this time STKCHK. This routine ensures that whenever the highest stack address (0100) is exceeded, an error signal is shown on the display (EEEEEE). After an error, the JC enters a loop and will only return to the monitor program if the RST key is depressed. This subroutine is shown in figure 18 and is especially useful for larger programs. We will come back to it a little later on, but first, a bit more about implied addressing that has not yet been covered.

### *Shift and rotate instructions*

These four single-byte instructions are variations of the implied instructions and are used to manipulate the contents of the accumulator. The instructions ASL, LSR, ROL and ROR (the shift and rotate instructions) are used in all sorts of mathematical operations. The first of the four instructions, ASL — Arithmetic Shift Left (op-code 0A) — moves each bit of the byte in the accumulator one position to the left:



The bit positions are indicated along the top. Each bit value has been assigned one of the letters p . . . w for clarity. As can be seen, all of the bits have moved one place to the left. The extreme right-hand bit has become zero and the value of the extreme left-hand bit determines the state of the carry flag c is set if p was high and reset if p was low. The N-flag becomes set if bit seven (q) was high and the Z-flag is set if the

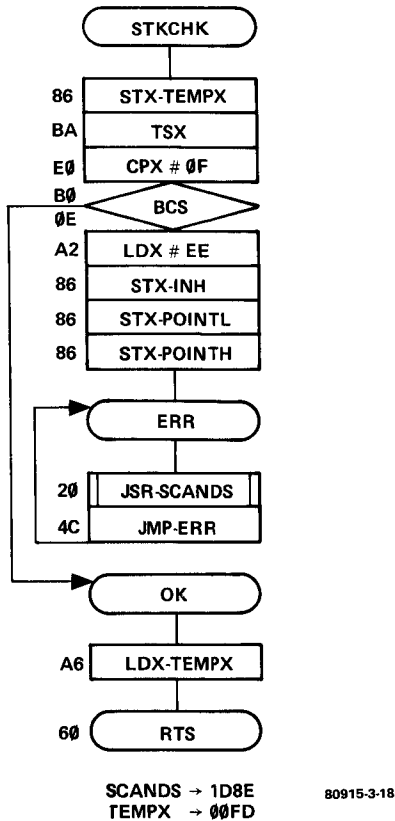
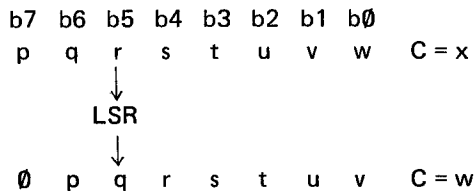


Figure 18. The monitor routine STKCHK which tests for the availability of further stack addresses. An error indication is given if there is no more room on the stack.

accumulator contents become 00000000 – eight consecutive ASL instructions!

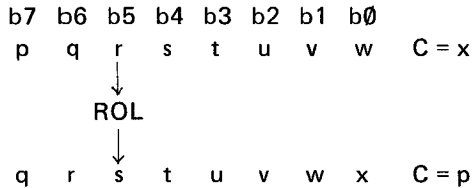
LSR-Logical Shift Right (op-code 4A). This instruction causes all the bits in the accumulator to be shifted one position to the right:



This time all the bits have been shifted one place to the right. The extreme left-hand bit becomes zero and the value of the extreme right-hand bit (w)

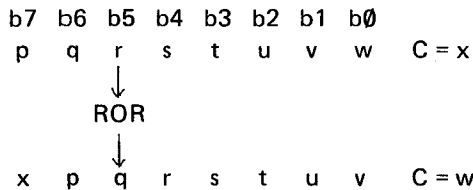
determines the state of the carry flag. The N-flag is not set by this instruction. Again, the Z-flag will only be set if the value in the accumulator becomes 00000000.

ROL-ROtate Left (op-code 2A). This instruction is similar to ASL in that all the bits are again moved one position to the left:



The difference between ROL and ASL is that this time the initial value of the carry bit is shifted into b0. For all the other bits the result is the same as that for the ASL instruction.

ROR-ROtate Right (op-code 6A). Once more each bit is moved to the left:



The difference between this instruction and LSR is that here the initial value of the carry bit replaces b7. For all the other bits the result is the same as for the LSR instruction.

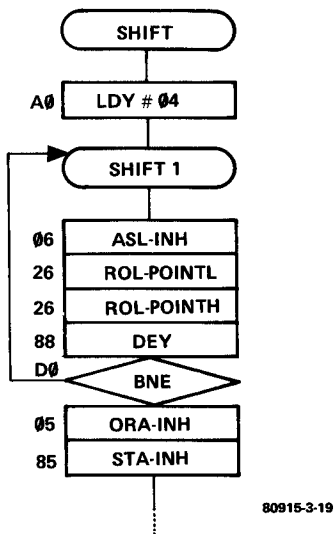
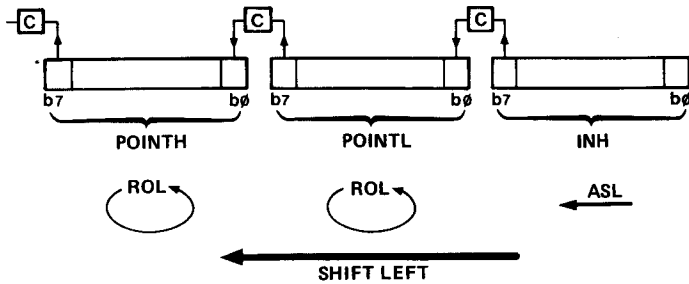
The difference between shifting and rotating is that the value of one of the end bits is lost when using shift instructions, but is kept when rotate instructions are used.

### *An example of shifting and rotating*

We already know that when entering data into the Junior Computer the information on the displays is shifted along from right to left. We also know from figure 14 that the information to be displayed is stored in and processed by three buffers — memory locations 00F9 . . . 00FB. The information to be displayed is shifted and rotated as shown in figure 19.

As soon as a key is depressed the SHIFT routine in the monitor program is performed. This moves the display information four bits to the left, which is equivalent to one display digit. Therefore, after the SHIFT routine, the five digits to the right will move one position to the left, the extreme left-hand digit will be lost and the new digit will appear on the right-hand side. This new digit is, of course, the same one that caused the SHIFT routine to be performed in the first place.

The actual operation of the routine is as follows. After the ASL instruction b0 of the INH byte becomes zero and the content of the carry bit is replaced with b7. The next instruction causes bit is replaced with b7. The next instruction causes the contents of byte POINTL to be rotated to the left. The carry bit is shifted into b0 of POINTL and b7 of POINTL is



**Figure 19. Operation of the monitor routine SHIFT which moves the display information one digit to the left.**

shifted into the carry bit. The same thing happens to POINTH during the next ROL instruction. So far, b0 of the INH byte has been replaced with zero and b7 of POINTH has been placed in the carry bit. This whole procedure is repeated four times which means that the carry bit is lost after the next ASL instruction. The result is that all the bits in the three bytes are shifted four places to the left and the four high order bits of POINTH are lost. The last two rectangles in the flow chart of figure 19 are for the newly entered digit.

*The following program uses the above instructions and certain monitor routines.*

### Decimal addition

There are many ways to add two decimal numbers: we can buy a calculator for a few pounds, a pencil and paper for a few pence; or we can take the difficult way out and design a 'software calculator'. As you have probably guessed, we are going to take the latter course of action so that we can learn how to use some of these new instructions.

We want to add two numbers as follows:

$$\begin{array}{cccccccc} X & X & X & X & X & X & + & Y & Y & Y & Y & Y & Y & = & Z & Z & Z & Z & Z & Z \\ \text{first number} & & & & & & & \text{second number} & & & & & & & \text{result} & & & & & & \end{array}$$

There are, however, a few rules:

When entering the numbers they must be displayed from right to left. The extreme left-hand digit must represent the most significant number (hundreds of thousands) and the extreme right-hand digit the least significant number (ones). The display must be cleared before a number can be entered.

The keys 0 . . . 9 are required to enter the digits. The DA key (11) will perform the '=' operation and the 'clear' function will be performed by the AD key (10). The actual addition will take place upon pressing the '+' key. If the result of the addition is larger than 999999 an error signal will have to be displayed. Finally, if a number is entered incorrectly the 'clear' key must be able to 'erase' it and then the user will be able to enter the correct number.

Figure 20 shows the flow chart for the addition of two decimal numbers. It may appear rather complicated at first sight, but it can be broken down into the nine subroutines given in figure 21.

Twelve memory locations are required to store the miscellaneous data and their 'names and addresses' are listed below:

POINTH 00FB	POINTL 00FA	INH 00F9	display-buffers
B12 0002	B11 0001	B10 0000	buffers for the first number
B22 0005	B21 0004	B20 0003	buffers for the second number
R2 0008	R1 0007	R0 0006	buffers for the result

The program (figure 20) starts with CLEAR1. This part of the program simply enters 00000000 into the display and number buffers (see also figures 21d, 21e and 21f). In the next section (FIRST) the program jumps to the subroutine KEYDIS (figure 21c), which uses the SCANDS and GETKEY routines contained in the monitor program. We are already familiar with these routines: the keyboard scan and the software debouncer. If a key is being pressed the routine GETKEY will determine which one it is and its value will be held in the accumulator upon returning from KEYDIS. A test is then carried out via the CMP # 10 instruction to see whether the CLEAR (AD) key was pressed and if so, the program will jump back to CLEAR1. If the CLEAR key was not depressed a further test is carried out (CMP # 12) to see whether it was the '+' key. If not, we

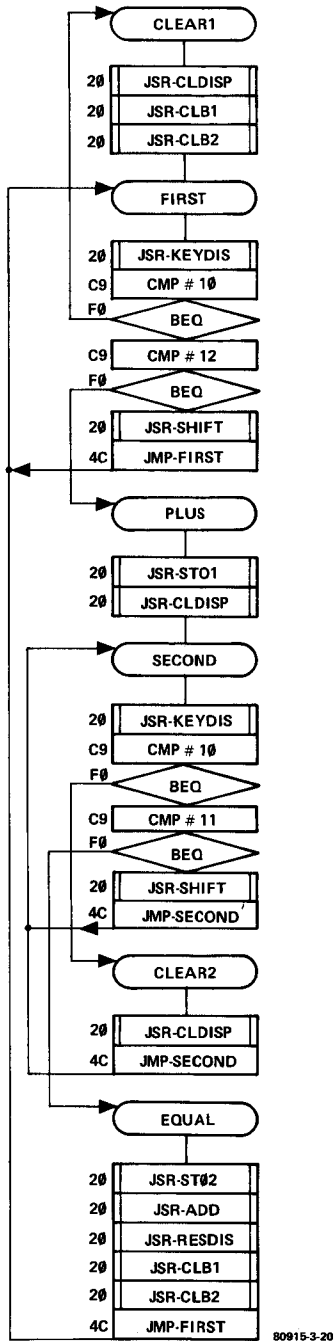


Figure 20. The main program for the addition of two 6-digit decimal numbers.



can presume it was one of the keys 0 . . . 9 in which case the program will jump to the SHIFT routine (figure 21a) to display the particular digit.

When the '+' key is depressed the program will move on to the third section (PLUS). This section stores the value of the first number into its buffer via the STO1 subroutine (figure 21h) and clears the display ready for the second number via the CLDISP subroutine (figure 21f).

The following section of the program (SECOND) obtains the value of the second number in exactly the same way as FIRST. Again, the key is tested to see whether it was the CLEAR key, in which case the program jumps to the subroutine CLDISP. The only difference is that this time the '=' is also tested (CMP # 11) and if it is depressed the program will enter the final section (EQUAL). The contents of the display buffers are stored via STO2 (figure 21g). Then, via the ADD subroutine (figure 21b), the two decimal numbers are added together and the result is stored in the buffers R0 . . . R2. The result is then displayed via the RESDIS subroutine (figure 21i). At the same time the two number buffers are cleared via the subroutines CLB1 and CLB2, in case a new addition is to follow immediately.

From this program it can be seen that the use of subroutines makes programming work much easier and they also enable a program to be 'followed' without too much difficulty.

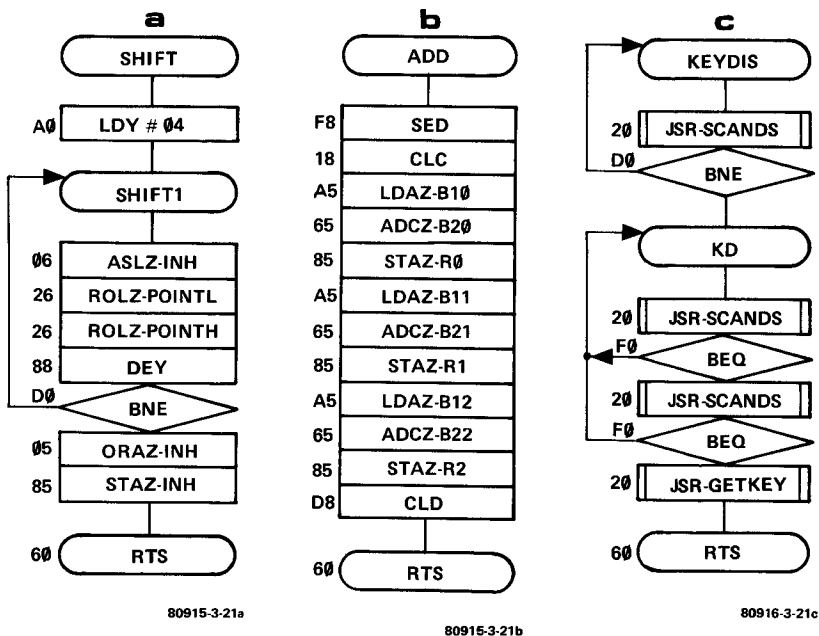
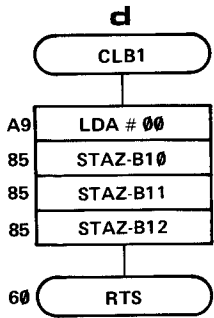
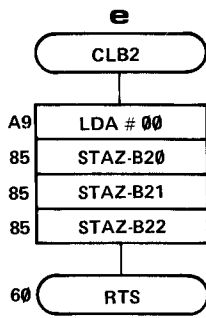


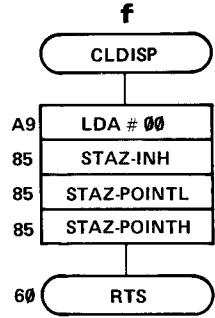
Figure 21. The nine subroutines required for the program (figure 20) to add two decimal numbers.



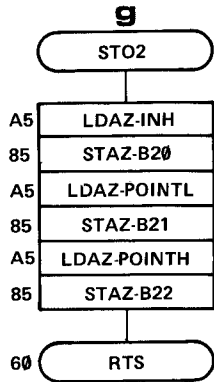
80915-3-21d



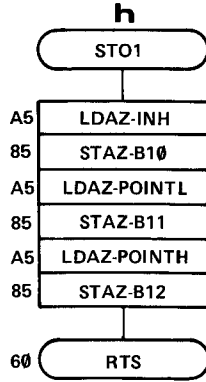
80915-3-21e



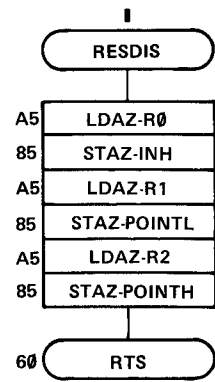
80915-3-21f



80915-3-21g



80915-3-21h



80915-3-21i

B10 → 0000  
 B11 → 0001  
 B12 → 0002

B20 → 0003  
 B21 → 0004  
 B22 → 0005

R0 → 0006  
 R1 → 0007  
 R2 → 0008

INH → 00F9  
 POINTL → 00FA  
 POINTH → 00FB

SCANDS → 1D8E  
 GETKEY → 1DF9

## Absolute indexed addressing

In this address mode, the contents of either the X or the Y index register are added to the 16-bit direct address provided by the second and third bytes of the instruction. The above may seem rather long-winded, but this address mode does simplify programming.

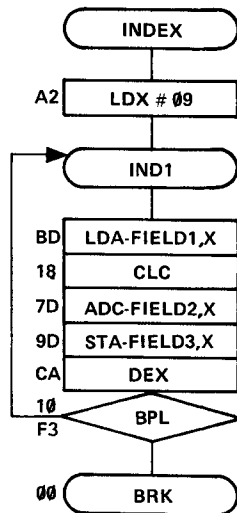
To recap, absolute address instructions consist of three bytes. The first for the actual instruction op-code and the following two bytes (ADL and ADH) for the direct address. Absolute indexed addressing is a variation of absolute addressing. Again, there are three bytes. The first for the actual instruction op-code, but this time the contents of the X or Y index

register are added to the direct address contained in the following two bytes before the instruction is executed. If, for instance, the contents of the X or Y register were xx and the contents of ADH and ADL were pp and qq respectively, the effective address would become ppqq + xx. To illustrate this address mode further we can examine the three tables shown below. A table is a set of consecutive address locations intended for the storing of data or results. Absolute indexed instructions are ideal when a table has to be filled with the results of an operation performed on two (or more) others:

TAB 1:	0120	TAB 2:	0011	TAB 3:	0300
0120	01	0011	10	0300	11
0121	02	0012	20	0301	22
0122	03	0013	30	0302	33
0123	04	0014	40	0303	44
0124	05	0015	50	0304	55
0125	06	0016	60	0305	66
0126	07	0017	70	0306	77
0127	08	0018	80	0307	88
0128	09	0019	90	0308	99
0129	0A	001A	A0	0309	AA

Tables 1 and 2 contain the numbers which have to be added together. Table 3 contains the results of the additions. Each table is denoted by its start address. The line being worked on is determined by the contents of the X register.

The flow chart for the program to perform this operation is shown in figure 22. The actual addition takes place in the section labelled IND1.



80915-3-22

**Figure 22. Program using absolute indexed addressing for the addition of numbers contained in two tables with the result stored in a third.**

The accumulator is first loaded with the contents of a given memory location in TAB 1. After the CLC instruction (clear carry for binary addition) the contents of the same line in TAB 2 are added (ADC) and the result is stored in the corresponding line of TAB 3 (STA).

The program starts by loading 09 into the X index register, the addition is then performed and the contents of the X register are reduced by one. The first number is therefore extracted from address location  $0120 + 09 = 0129$  of TAB 1, the second from location  $0011 + 09 = 001A$  of TAB 2, and the result of the addition is stored in location  $0300 + 09 = 0309$  of TAB 3. In other words, the program begins at the bottom line of the tables and works its way up to the top line.

The addition can also be performed using the 'old' method of absolute addressing:

- 1st byte: LDA-op-code
- 2nd byte: ADL data to be loaded
- 3rd byte: ADH data to be loaded
- 4th byte: CLC for binary addition

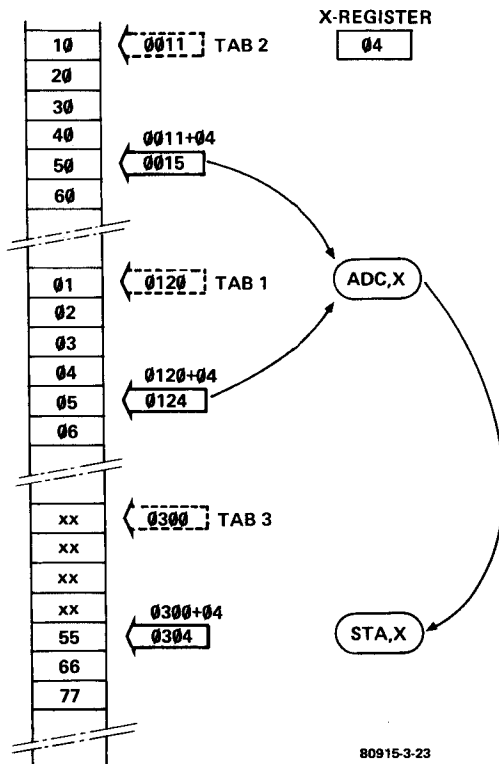


Figure 23. Memory map of the program in figure 22 when the value of the X register is 04.

5th byte: ADC-op-code  
 6th byte: ADL data to be added  
 7th byte: ADH data to be added  
 8th byte: STA-op-code  
 9th byte: ADL data to be stored  
 10th byte: ADH data to be stored

And this is only for one line of the tables!!

Using the above as an example, this means that  $10 \times 10 = 100$  memory locations are required for the full program. When compared with the program given in figure 22, which only requires 16 memory locations from start to finish, it can be seen that there is a valid reason for using absolute indexed addressing. Figure 23 shows the memory map of the program when the contents of the X register are 04.

Another example where absolute indexed addressing is indispensable is shown in the program (MOVE) in figure 24. The object of this exercise is to copy the contents of one address field (FROMAD, location 0172) to another (TOAD, location 03A0). The program, in fact, transfers a block of data from one address area to another. Initially, the X register is loaded with 04. After a data transfer has taken place the contents of the X register are reduced by one. The data at address  $0172 + 04 = 0176$  is loaded into the accumulator and then stored in address  $03A0 + 04 = 03A4$ . This continues until the X register becomes negative whereupon the program jumps back to the monitor. The program can be entered as follows:

AD				xxxx	xx	
0	2	0	0	0200	xx	
DA		A	2	0200	A2	LDX #
+		0	4	0201	04	
+		B	D	0202	BD	LDA-, X
+		7	2	0203	72	ADL of FROMAD
+		0	1	0204	01	ADH of FROMAD
+		9	D	0205	9D	STA-, X
+		A	0	0206	A0	ADL of TOAD
+		0	3	0207	03	ADH of TOAD
+		C	A	0208	CA	DEX
+		1	0	0209	10	BPL
+		F	7	020A	F7	displacement
+		4	C	020B	4C	JMP
+		3	3	020C	33	ADL of monitor routine
+		1	C	020D	1C	ADH of monitor routine
AD				020D	1C	
0	2	0	0	0200	A2	
GO				0200	A2	program start

Upon completion, the program jumps to the monitor and the start address is shown on the displays.

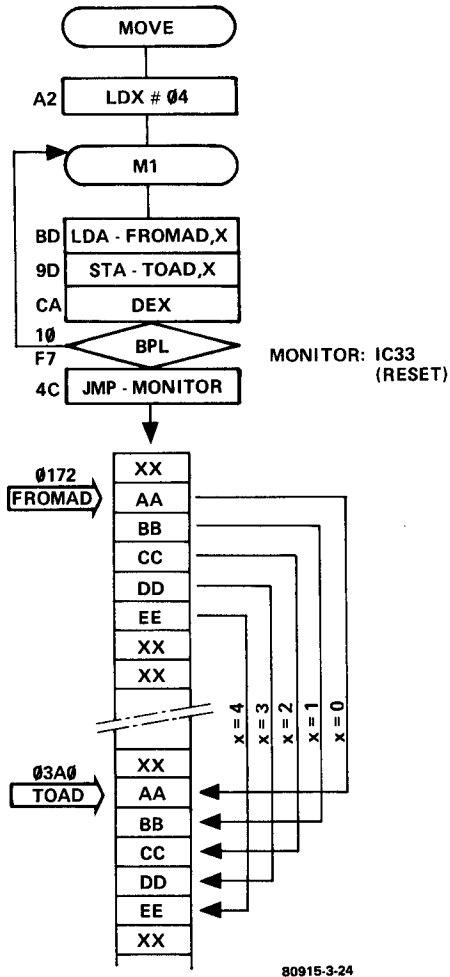


Figure 24. A program which copies a block of data from one area of memory to another.

Note: The X and Y registers are eight-bit registers. This means that a maximum of 256 data bytes can be re-located in this manner.

### Zero page indexed addressing

This is a further variation of absolute and absolute indexed addressing. In this instance, only the low order address byte (ADL) and the op-code have to be entered. The high order address byte is always the same (00).

The effective address is calculated from the data byte immediately following the instruction op-code plus the contents of the X or Y index register.

It should be noted that zero page indexed addressing is 'wraparound'. This means that if the sum of the index register and the low order address byte exceeds 255 (FF) the carry bit will be discarded. In other words, if the contents of the X or Y index register were 8B and the low order address byte was B1, the effective address would become 003C *not* 013C.

## Indirect addressing

The end of the tunnel is in sight! The last of the address possibilities is about to be described.

Instructions that use simple indirect addressing consist of three bytes. The first for the actual op-code, and the second and third provide a 16-bit address where the actual address can be found. This means that the indirect address can be located anywhere in memory. Sounds complicated? Not really.

As an example, let us assume that we wish to jump to another part of the main program, but we do not know the actual address that we want to jump to. We do know, however, that the actual jump address is stored in location 2B84. The op-code for the jump (JMP) instruction when using indirect addressing is 6C. Therefore, the full instruction would be 6C842B. If, for instance, locations 2B84 and 2B85 contained 06 and 1A respectively, the program would jump to address 1A06.

## Indirect indexed addressing

There are two forms of indirect indexed addressing: pre-indexed indirect addressing and post-indexed indirect addressing. In both cases page zero is used to compute the effective address and the instruction code, therefore, is two bytes long.

### *Post-indexed indirect addressing*

To recap, figure 25a shows how the accumulator is loaded with the contents of address location 021A using absolute indexed addressing (with the Y register). The portion of program looks like this:

```
B9 LDA- ABS,Y
1A ADL of address from which data is to be loaded
02 ADH of address from which data is to be loaded
```

As the Y register contains 00, the data contained in address location 021A + 00 = 021A (B3) is loaded into the accumulator. The essential feature of the above is that the address where the data is to be found must be known.

Post-indexed indirect addressing uses the Y index register to compute the effective address. The second byte of the instruction specifies a location in the first page of memory where an indirect address can be found. The

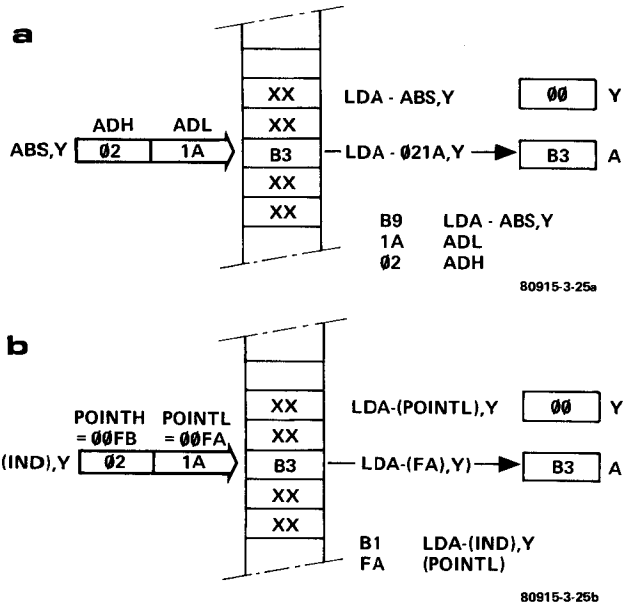
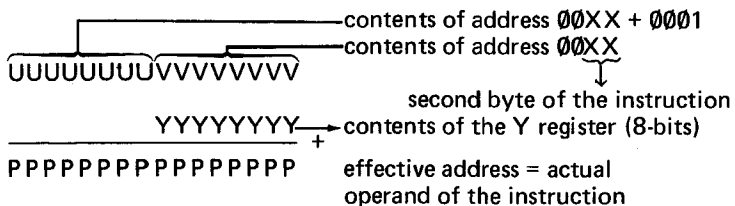


Figure 25. Comparison of the load instruction using absolute indexed addressing and post-indexed indirect addressing. The latter is one byte shorter!

contents of the `Y` register are then added to this indirect address to provide the effective address. Figure 25b shows that the same results can be obtained by using post-indexed indirect addressing, but with one less memory location being used. The actual instruction is in fact:

```
B1 LDA - (IND),Y
FA (POINTL)
```

The effective address is calculated as follows. The memory locations `00FA` and `00FB` (`POINTL` and `POINTH`) contain the `ADL` and `ADH` of the indirect address respectively. By instructing the computer to look at address `00FA` the complete indirect address (`021A`) is found automatically. The contents of the `Y` index register (in this case `00`) are then added to this indirect address to provide the effective address. Once again, the contents of `021A` (`B3`) will be loaded into the accumulator. This can also be represented by the following:





where any carry produced by adding bytes V and Y is added to byte U. The actual addition is carried out in the Y index register.

### *Block transfer*

We have already seen that it is possible to move up to 256 bytes from one memory location to another. We are now able to extend this facility and write a program which is able to transfer up to 255 blocks of 256 bytes! This gives a total of 65,280 bytes. The program also illustrates the use of post-indexed indirect addressing instructions.

Before we explain the actual program in detail let us take a look at the memory map shown in figure 26. Here we can see that two data blocks consisting of 256 bytes each are to be moved to a different area of memory.

The first address of the first block is 0200 and the end address is 02FF. This address is also found in two memory locations on page zero; BEG = 00 and BEG + 1 = 02. This first data block has to be moved to the area A800 . . . A8FF. The first address of this area is also found in two locations of page zero: MOV = 00 and MOV + 1 = A8. The various address locations on page zero have been assigned labels as follows:

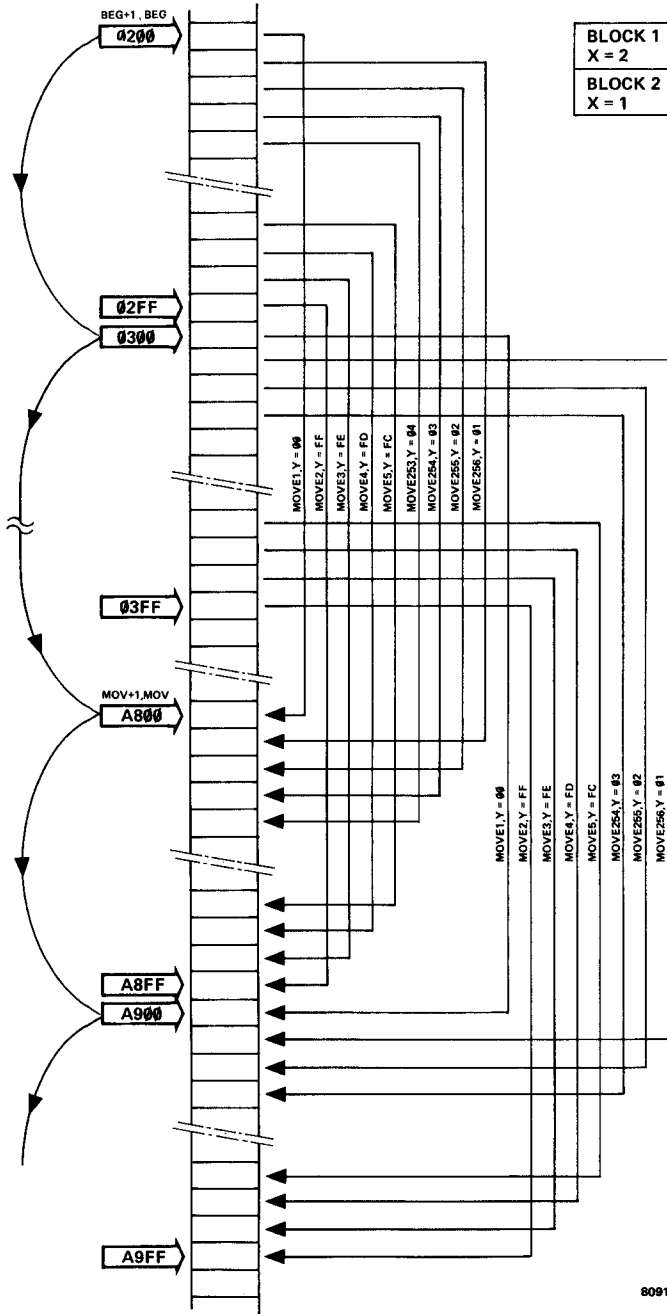
BEG : location 0000. Contents are called FRADL and are equal to 00  
BEG + 1 : location 0001. Contents are called FRADH and are equal to 02  
MOV : location 0002. Contents are called TOADL and are equal to 00  
MOV + 1 : location 0003. Contents are called TOADH and are equal to A8  
BLOCKS : location 0004. Contents are called N and are equal to 02 (number of blocks to be moved)

The labels are logical enough; FR stands for FRom and TO is self explanatory; ADL (low order byte) and ADH (high order byte) should be familiar by now.

The actual program for moving data blocks is not as complicated as you might expect. It consists of two (sub)routines whose flow charts are shown in figure 27. The start of the program (DEFMOV) simply stores the correct (indirect) address information into the locations 0000 . . . 0004 on page zero. Once all this information has been stored the program jumps to the subroutine BLMOVE to effect the transfer.

The first thing this subroutine does is to load the X register with the contents of BLOCKS (02). The X register is in fact used as a block counter. The Y register is used to inform the computer how far the transfer in a given block has progressed. Initially, the Y register is loaded with 00.

In the section called LOOP the processor loads the accumulator with the contents of 0200 + 00 (Y = 00) and stores that data in location A800 + 00. The Y register is then decremented (= FF) and the data from location 0200 + FF is stored in location A800 + FF. The Y register is again decremented (= FE) and the process continues until the Y register contains 00. As soon as this happens the contents of BEG + 1 and MOV + 1 are incremented ready for the next block of data and the contents of the X register are decremented. The program then jumps back



80915-3-26

Figure 26. Two data blocks of 256 bytes each are to be moved to a different section of memory. The actual program is shown in figure 27.

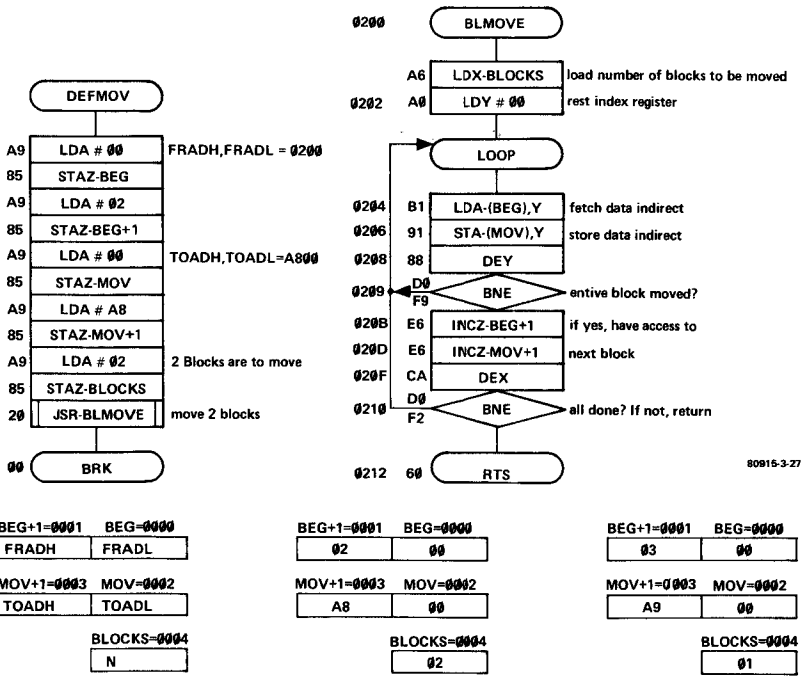


Figure 27. Complete program for the transfer of up to 255 data blocks of 256 bytes. Post-indexed indirect addressing is used.

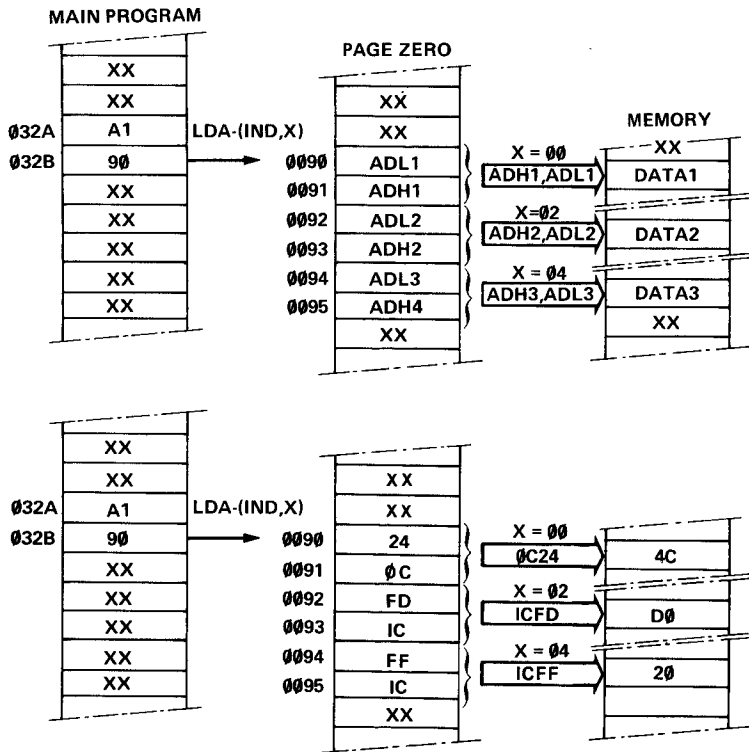
to LOOP and the second block of data is moved. Once the complete second block has been moved, the contents of BEG + 1 and MOV + 1 are once again incremented and the contents of the X register decremented. Now that the X register contains zero the processor will return to the main program and stop.

*Pre-indexed indirect addressing*

Pre-indexed indirect addressing uses the X register. This time, as its name suggests, the second byte of the instruction code is added to the contents of the X index register *before* the effective address is found. When using pre-indexed indirect addressing, 'wraparound' addition is performed once more. This means that when the contents of the X register are added to the second byte of the instruction code, any carry will be discarded. Again page zero is used for this address mode.

If, for instance, the second byte of the instruction was 7C and the contents of the X index register were 89, the indirect address would be 007C + 0089 = 0005 (not 0105) and the effective address would be found at locations 0005 (ADL) and 0006 (ADH).

The main differences between the two forms of indirect indexed addressing are summarised below.



80915-3-28

**Figure 28. Example of pre-indexed indirect addressing. The effective address is contained in the location (on page zero) pointed to by the second byte of the instruction plus the contents of the X index register.**

- When using pre-indexed indirect addressing the contents of the X register are added to the second byte of the instruction code to access a memory location on page zero. The effective address is contained in this (and the following) page zero location.
- When using post-indexed indirect addressing the second byte of the instruction code points directly to a memory location on page zero. The contents of the Y register are then added to the indirect address found there to provide the effective address.

An example of pre-indexed indirect addressing is given in figure 28. Three sections of memory are each shown twice (one on top of the other). The top three give a general view of the overall picture while specific address and data information has been filled in at the bottom. The left-hand section is part of the main program, the centre section is page zero, and the right-hand section is an area of memory where the data are to be found.

At address 032A of the main program we find the instruction A190,

LDA-(IND,X), which tells the computer that the indirect address will be found at location 0090 plus the contents of the X register. If the contents of the X register are 00 the effective address will be found in location 0090 (ADL1) and 0091 (ADH1). Care must be taken when using this form of addressing for if the contents of the X register were 01, the indirect address would be contained in locations 0091 and 0092. These two addresses contain ADH1 and ADL2 respectively, which would not do at all! This can be overcome by ensuring that the X register contains 00, 02 or 04 (for the purposes of this example).

If the contents of the X register were 02, the effective address would be found in locations 0092 and 0093 (ADL2 and ADH2) and the information at the effective address (DATA2) would be loaded into the accumulator. If the X register contained 04, DATA3 would be loaded into the accumulator (effective address would be found in locations 0094 and 0095). The main reason for using this form of addressing is that page zero can be used as a 'look up table' for various pointers. Each of the pointers will indicate a specific address where the data to be worked on can be found.

### Interrupting a running program: NMI, IRQ and RESET

An interrupt is a signal that causes the computer to stop whatever it was doing and perform a special (predetermined) task. This task is similar to a subroutine and once it has been completed the computer will return to the main program exactly where it left off as if nothing had happened.

An everyday example: imagine that you are sitting in your favourite armchair with a good book and the telephone rings. You are disturbed and the following interrupt routine takes place: you mark the the page you were reading and put the book down, you then answer the telephone (service the interrupt). When the conversation is over you replace the receiver, pick up the book and find the page you were reading previously (return from interrupt).

When the computer receives an interrupt request it stores essential information such as the contents of the program counter (return address) and the status register etc. before actually jumping to the interrupt routine. Upon completion of the routine this information is retrieved and the computer will return to the main program. The interrupt routine is initiated by an external signal and not by any instructions in the actual program. There are two methods of interrupting the program via hardware;

1. Interrupt request, IRQ. When the interrupt flag (I) in the status register is '0' and the interrupt request input of the microprocessor (pin 4) is taken low the computer will service the interrupt request. If, however, the interrupt flag is '1' the command will be ignored. This flag can be set or reset with the following instructions:  
CLI (op-code 58) where I = 0; Interrupt enable  
SEI (op-code 78) where I = 1; Interrupt disable
2. Non-Maskable Interrupt, NMI. Regardless of the state of the interrupt flag, an interrupt will be serviced when the NMI input (pin 6) of the microprocessor is taken low.

The interrupt signals IRQ and NMI, along with RES (still to be covered),

can be termed 'hardware instructions', or more precisely 'hardware jump instructions'. As we can see from the above, the IRQ 'instruction' is conditional and the NMI 'instruction' is unconditional.

#### *NMI operation*

When the NMI input is taken low the processor investigates the contents of locations FFFA and FFFB. These locations contain the start address of the interrupt routine. In the example shown in figure 29 this start address is 0200. The low order byte is contained in FFFA and the high order byte in FFFB. The *NMI-vector* is very similar to the address pointers mentioned earlier. The NMI-vector simply 'points' to the start address of the interrupt routine.

The interrupt routine is worked on in exactly the same way as a normal subroutine. Nesting of routines is also possible. Once the interrupt routine has been completed the instruction RTI (ReTurn from Interrupt-op-code 40) will cause the processor to jump back to the main program.

#### *IRQ operation*

Following an Interrupt ReQuest the processor will examine the state of the interrupt flag (I) in the status register. If this flag is set (= 1) the request will be ignored and the computer will carry on with the main program. If, on the other hand, the interrupt flag is reset (= 0), the processor examines the contents of locations FFFE and FFFF. These are the bytes reserved for the *IRQ-vector* and which contain the start address of the IRQ interrupt routine. In the example of figure 29 this start address is shown as 24C3. The high order byte is contained in location FFFF and the low order byte in FFFE. Again, nesting of (sub)routines is possible and the interrupt routine ends with the RTI instruction.

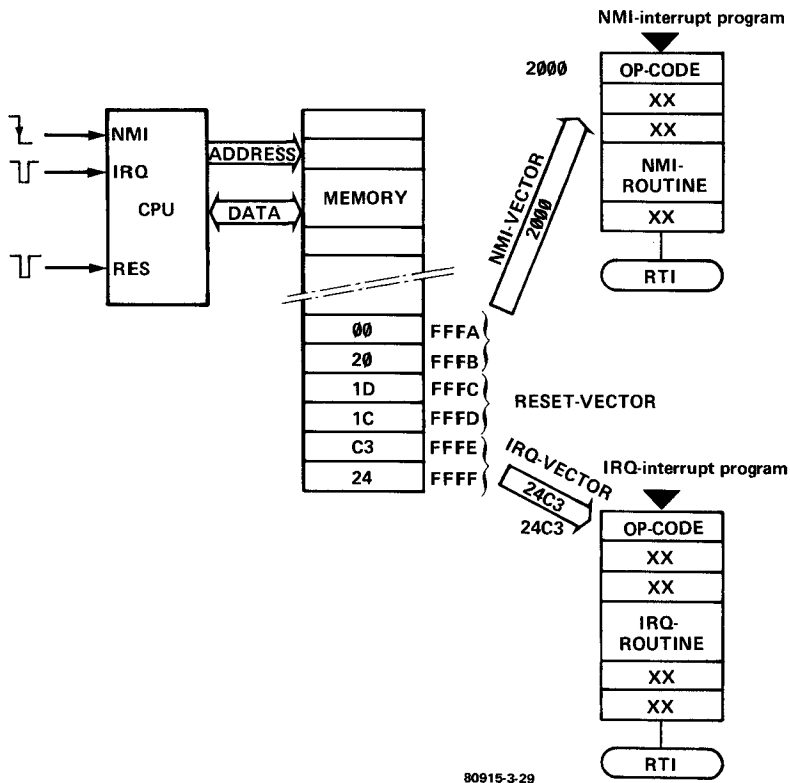
As soon as an interrupt request is acknowledged the interrupt flag is set so that further interrupt requests are inhibited.

With both forms of interrupt request the contents of all the internal registers are pushed onto the stack. Once the interrupt routine has been completed (RTI) all the information is pulled off the stack and replaced in the various registers.

#### *Reset*

The operation of the computer can be affected by a third 'hardware instruction'. When the reset input (pin 40) of the microprocessor is taken low the contents of locations FFFC and FFFD are examined. These locations (as expected) are the *reset vector* and in the case of the Junior Computer contain the address 1C1D. This is the start address of the reset routine contained in the monitor program. In other words, each time the reset key is operated the processor will effectively jump to this routine.

All three vectors are situated on page FF which is the highest possible page. Observant readers may have realised that, due to the incomplete addressing of the JC, this page cannot be addressed directly. However, the address lines required to decode this page are (from left to right) A15 . . . A8. Upon examination of the circuit diagram of the Junior Computer (chapter 1) we find that address lines A13, A14 and A15 are

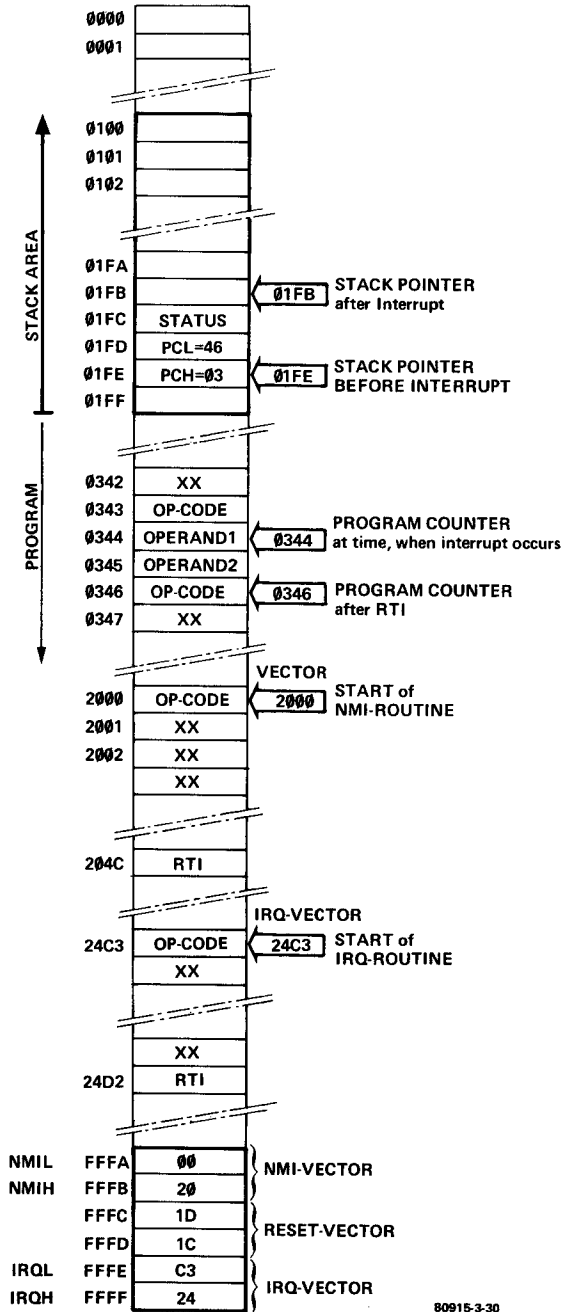


**Figure 29.** Illustration of the operation when an interrupt signal is received by the microprocessor.

not used. This means that these address lines have no influence on the actual memory locations to be addressed. By not using these three address lines the highest addressable page becomes 1F and the highest memory location becomes 1FFF. The processor is therefore 'fooled' into looking at locations 1FFA . . . 1FFF for the various vectors. These locations are, of course, part of the monitor program held in the EPROM and are therefore permanently pointing at certain address locations.

A further example of what happens when the processor receives one of the interrupt requests (NMI and IRQ) is shown in the memory map of figure 30. Here, page 01 is used as the stack and page 03 contains the main program. The IRQ vector is pointing to 24C3, the start of the IRQ routine, and the NMI vector is pointing to 2000, where the NMI routine starts.

Imagine that the main program is half way through the instruction at location 0343 when the NMI input goes low. The computer will not service the interrupt until the instruction being worked on has been completed. The high order byte of the program counter (PCH) is then pushed onto the stack and the stack pointer is incremented. The low



80915-3-30

Figure 30. A memory map of the procedure for dealing with interrupt requests.



order byte of the program counter (PCL) is then pushed onto the stack and the stack pointer again incremented. The stack pointer will now be pointing at address 01FC, which is where the contents of the status register are stored. After this procedure the stack pointer will point at location 01FB.

The next phase in the operation is the jump to the start of the NMI routine, which in this case is address 2000. At the same time the interrupt flag is set so that any subsequent interrupt request (IRQ) will be ignored. If the interrupt was caused by an IRQ 'instruction' instead of an NMI, a very similar sort of thing happens. Again, the contents of the program counter and the status register are pushed onto the stack. The contents of locations FFFE and FFFF are examined to obtain the start address of the IRQ routine and the interrupt flag is set. If, however, the NMI input of the processor goes low during the operation of the IRQ routine, the computer will store the contents of the program counter and status register once more and jump to the NMI routine. Once the NMI routine has been completed the processor will jump back to the IRQ routine and finish that off before returning to the main program. This is a very similar sort of action to the nesting of subroutines.

As soon as the RTI instruction is encountered in either of the interrupt routines, the contents of the stack are replaced into the relevant registers in the reverse order to which they were pushed onto the stack. In other words, (in figure 30) the contents of the stack pointer are incremented (to 01FC) and the information held there is restored into the status register. The stack pointer is incremented once more and the low order byte of the return address is replaced in the program counter etc. Once all the old values have been restored the processor will continue the main program from where it left off.

If the contents of more registers are to be saved, this can be accomplished with the various push instructions. To restore the data, of course, the pull instructions are required. Interrupt (or sub) routines could well start in the following manner:

```

SAVE: PHA  push the contents of the accumulator onto the stack (then
          increment the contents of the stack pointer)
      TXA  transfer the contents of the X register into the accumulator
      PHA  push the contents of the accumulator onto the stack
          (increment stack pointer)
      TYA  transfer the contents of the Y register into the accumulator
      PHA  push the contents of the accumulator onto the stack
          (increment stack pointer)
      . . . . first instruction of the interrupt routine
      .
      .
      .
      .
      .
      .
      .
RESTO PLA  replace the top of the stack into the accumulator (then
          decrement the contents of the stack pointer)
  
```

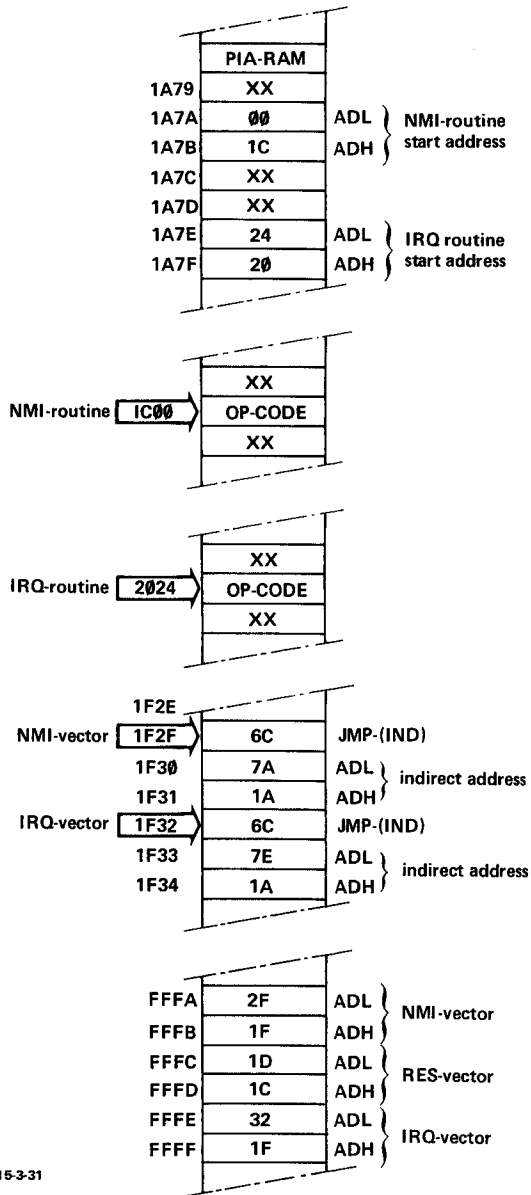


Figure 31. The servicing of interrupts via the indirect jump instruction. The contents of the indirect address point to the effective address and are programmable.

TAY place the contents of the accumulator in the X register  
 PLA replace the top of the stack in the accumulator (increment)

stack pointer)  
 TAX place the contents of the accumulator in the X register  
 PLA replace the top of the stack in the accumulator (de-  
 crement stack pointer)  
 END RTI return from the interrupt routine

Under these circumstances it is essential that the relevant interrupt vector is pointing to the start of the SAVE routine and not to the start of the actual interrupt routine. It is also important to remove information from the stack in the reverse order to which it was placed on the stack (and replace it in the correct registers or memory locations).

Figure 31 shows how the indirect jump instruction can be used with interrupt routines. In this instance the computer examines the contents of locations FFFA and FFFB to determine the start address of the NMI interrupt routine which appears to be address 1F2F. This is in fact the address of an indirect jump instruction which directs the program to addresses 1A7A and 1A7B. This is a section of memory which is contained in the PIA (see chapter 1). Therefore, by loading these last locations with the start address of the interrupt routine any number of different routines can be 'jumped' to.

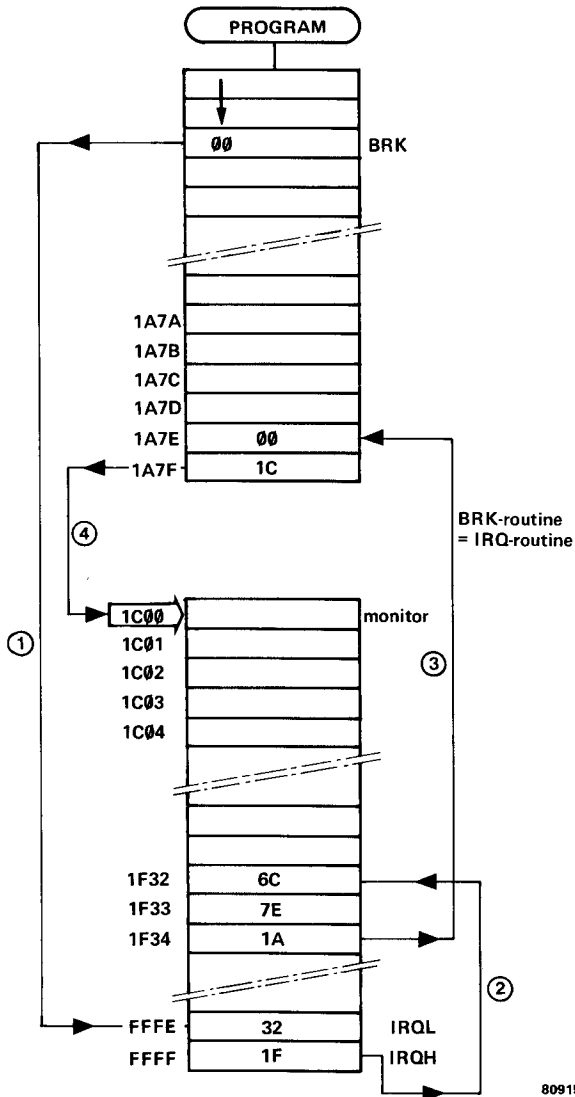
As shown in the example of figure 31, when the NMI input is taken low the computer first examines the contents of locations FFFA and FFFB where it finds the address 1F2F. The instruction (6C7A1A) at that address informs the computer to continue operation from the address contained in location 1A7A. This address is in fact 1C00, the start of the monitor routine. If the IRQ input was taken low the processor would perform a similar operation and continue the program from the address contained in locations 1A7E and 1A7F (2024).

All of the above is contained in the monitor program with the result that when developing (or running) a program in RAM the contents of locations 1A7A/1A7B and 1A7E/1A7F can be altered so that when an interrupt occurs virtually any routine can be run.

## The BRK instruction

Up to now, all interrupt routines have been initiated by an external influence. A software interrupt, the BRK instruction, is also possible. We have seen this instruction at the end of many program examples and it is time to discover exactly how the BRK instruction works.

Effectively, it is exactly the same as the IRQ 'instruction'. The processor examines the contents of locations FFFE and FFFF to determine the address of the next instruction. In the example in figure 32, after the processor encounters the BRK instruction in the main program, the address 1F32 is obtained from the interrupt vector. The indirect jump instruction at address 1F32 causes the computer to continue operation from the address contained in locations 1A7E and 1A7F which in the example is 1C00. Therefore, after encountering the BRK instruction the processor jumps to the monitor program. The break flag (B) in the status register is set when a software interrupt occurs. This flag remains 0 during normal (hardware) interrupts.



80915-3-32

**Figure 32.** The BRK instruction is a form of 'software interrupt'. This example shows the four jumps that are made before the BRK-routine (= IRQ-routine) is actually performed.

This concludes the chapter on the various address possibilities. Chapter four will expand on what we have learned and will provide some interesting programming examples.

# A simple beginning

## Programming without headaches

**This is the last chapter of the first book on the Junior Computer. The second book will provide some complex program examples and techniques. Before book two can be fully appreciated however, it is advisable to have a taste of what real programming is about. This chapter describes a few interesting programs to enable the JC owner to become more accustomed to programming and operating his/her machine.**

The monitor routines, I/O programming, hexadecimal editing and assembling are all described in detail in book two. Before you can run, however, you have to learn to walk – nobody starts their first piano lesson with a piece by Chopin!! With chapter three firmly implanted in your mind we can move on to higher things.

The program examples given in this chapter are accompanied by detailed flow charts and the actual keystrokes required to enter them into the JC. Some more practical tips are also given.

### **Fingerwork for the decimal addition program**

Before a program can be of any use it has to be entered into the memory of the Junior Computer. This program requires a total of 594 keystrokes to fill 196 memory locations – and that is without making any mistakes! Once this has been completed successfully the program can be started and we can perform the decimal addition of two six digit numbers.

The actual program was mentioned in chapter 3. Figures 14 (the 'name and address' of each of the display buffers), 20 (overall flow chart of the program), and 21 (detailed flow charts of the nine subroutines) should be referred to.

The construction of detailed flow charts is a very important phase when developing a program. All decisions etc. can be directly translated into

instructions and thus the op-codes and the number of bytes required per instruction can be worked out. In some instances even displacement values can be filled in, but more on that later.

To start with, let us have a look at the program given in figure 1 which consists of the following sections:

locations 0200 ... 0248: main program (figure 20)  
locations 0249 ... 0258: SHIFT subroutine (figure 21a)  
locations 0259 ... 026E: ADD subroutine (figure 21b)  
locations 026F ... 0281: KEYDIS subroutine (figure 21c)  
locations 0282 ... 028A: CLB1 subroutine (figure 21d)  
locations 028B ... 0293: CLB2 subroutine (figure 21e)  
locations 0294 ... 029C: CLDISP subroutine (figure 21f)  
locations 029D ... 02A9: STO2 subroutine (figure 21g)  
locations 02AA ... 02B6: STO1 subroutine (figure 21h)  
locations 02B7 ... 02C3: RESDIS subroutine (figure 21i)

### *Program preparation*

The first instruction (at address location 0200) is going to be a jump to subroutine (JSR) instruction. The actual start address of the subroutine is not always known to begin with. This is because the total length of the main program and the other subroutines has yet to be determined. This brings us to two general and very practical rules:

- 1. Time is well spent if a 'listing' of the program is made (on paper) before it is entered into the computer. This should be done after the detailed flow chart has been drawn and before the actual program entry.*
- 2. Enough space should be left for data and displacement values that have not yet been calculated. These 'spaces' can be filled in later.*

The second is very important because:

- 3. Programs should be contained in a single area of memory. This also applies to subroutines.*

These rules are related to the fact that the contents of the program counter are incremented after each instruction. If the processor comes across an empty space, it will either not know what to do, or will certainly misinterpret the intended instruction with (usually) disastrous results. Memory locations can be skipped, by accident, by pressing the + key twice inadvertently. Care must, therefore, be taken when a program is being entered by hand.

Gaps in the program can be filled with the NOP instruction (No Operation-op-code EA). This same instruction can also be entered at various locations on purpose in case extra instructions are required later on. There is nothing more time consuming than having to completely re-enter a long program.

Mistakes can occur at any stage during the development of a program. It is important therefore to be able to check the final version before the

program is actually run. This can be accomplished by the JC itself.

### *Program verification*

The control keys AD, DA and + can be used to check and (if necessary) alter the contents of specific memory locations. A particular location can be examined as follows:

key AD xxxx

where xxxx is the actual address of the memory location to be examined. This address will appear on the left-hand side of the display and the data contained there will be shown on the two right-hand digits. To examine the following memory location the + key is operated. The address displays will then show xxxx + 0001 and the data displays will give the contents of this second location. The complete program can be examined in this way. If a correction has to be made to the data contained in a particular address location, it is important to ensure that the location is the right one. Once this has been confirmed the DA key can be operated and the correct data can be entered. By operating the + key the data in the following location can be altered if desired, or else locations can be 'skipped' by repeatedly pressing the + key. As can be seen, there are adequate provisions for verifying and altering the contents of memory locations, should the need arise (and more often than not it does!).

Various program sections and subroutines are usually given a 'label'. In figure 1 the labels are shown inside a rectangle on the left-hand side (comment section). When writing out the program the various program sections can be referred to by their label rather than their actual address. As mentioned earlier, at this stage the actual address may not be known. These addresses together with displacement and data values can be filled in once the total number of memory locations required for the program (section) has been worked out.

Not only is it important that a program occupies a fixed number of (consecutive) memory locations, but it is also important that program sections do not overlap. It is imperative that the program does not run into the memory area reserved for the monitor program.

There is no rule that subroutines must follow straight on from where the main program finishes, which is the case in figure 1. It is always possible to have a number of unused memory locations situated between the main program and the subroutine. This is especially true of short programs. On the other hand, when developing longer programs it is advisable to use memory space as economically as possible.

How many memory locations can we actually use?

*4. The standard Junior Computer can directly access 1 k of RAM, which means that all locations from 0000 . . . 03FF can be addressed. This gives a total of four pages of 256 bytes each:*

*page 00: 0000 . . . 00FF*

*page 01: 0100 . . . 01FF*

*page 02: 0200 . . . 02FF*

*page 03: 0300 . . . 03FF*

key			address	data	comments
RST			xxxx	xx	
AD			xxxx	xx	
0	2	0	0	0200	xx
DA		2	0	0200	20 JSR- <b>CLEAR1</b>
+		9	4	0201	94 ADL of CLDISP
+		0	2	0202	02 ADH of CLDISP
+		2	0	0203	20 JSR-
+		8	2	0204	82 ADL of CLB1
+		0	2	0205	02 ADH of CLB1
+		2	0	0206	20 JSR-
+		8	B	0207	8B ADL of CLB2
+		0	2	0208	02 ADH of CLB2
+		2	0	0209	20 JSR- <b>FIRST</b>
+		6	F	020A	6F ADL of KEYDIS
+		0	2	020B	02 ADH of KEYDIS
+		C	9	020C	C9 CMP #
+		1	0	020D	10 with 10
+		F	0	020E	F0 BEQ
+		F	0	020F	F0 go to CLEAR1
+		C	9	0210	C9 CMP #
+		1	2	0211	12 with 12
+		F	0	0212	F0 BEQ
+		0	6	0213	06 go to PLUS
+		2	0	0214	20 JSR-
+		4	9	0215	49 ADL of SHIFT
+		0	2	0216	02 ADH of SHIFT
+		4	C	0217	4C JMP-
+		0	9	0218	09 ADL of FIRST
+		0	2	0219	02 ADH of FIRST
+		2	0	021A	20 JSR- <b>PLUS</b>
+		A	A	021B	AA ADL of STO1
+		0	2	021C	02 ADH of STO1
+		2	0	021D	20 JSR-
+		9	4	021E	94 ADL of CLDISP
+		0	2	021F	02 ADH of CLDISP
+		2	0	0220	20 JSR- <b>SECOND</b>
+		6	F	0221	6F ADL of KEYDIS
+		0	2	0222	02 ADH of KEYDIS
+		C	9	0223	C9 CMP #
+		1	0	0224	10 with 10
+		F	0	0225	F0 BEQ
+		0	A	0226	0A go to CLEAR2
+		C	9	0227	C9 CMP #
+		1	1	0228	11 with 11
+		F	0	0229	F0 BEQ
+		0	C	022A	0C go to EQUAL
+		2	0	022B	20 JSR-
+		4	9	022C	49 ADL of SHIFT
+		0	2	022D	02 ADH of SHIFT
+		4	C	022E	4C JMP-
+		2	0	022F	20 ADL of SECOND
+		0	2	0230	02 ADH of SECOND



+		2	0	0231	20	JSR-	<b>CLEAR2</b>
+		9	4	0232	94	ADL of CLDISP	
+		0	2	0233	02	ADH of CLDISP	
+		4	C	0234	4C	JMP-	
+		2	0	0235	20	ADL of SECOND	
+		0	2	0236	02	ADH of SECOND	
+		2	0	0237	20	JSR-	<b>EQUAL</b>
+		9	D	0238	9D	ADL of STO2	
+		0	2	0239	02	ADH of STO2	
+		2	0	023A	20	JSR-	
+		5	9	023B	59	ADL of ADD	
+		0	2	023C	02	ADH of ADD	
+		2	0	023D	20	JSR-	
+		B	7	023E	B7	ADL of RESDIS	
+		0	2	023F	02	ADH of RESDIS	
+		2	0	0240	20	JSR-	
+		8	2	0241	82	ADL of CLB1	
+		0	2	0242	02	ADH of CLB1	
+		2	0	0243	20	JSR-	
+		8	B	0244	8B	ADL of CLB2	
+		0	2	0245	02	ADH of CLB2	
+		4	C	0246	4C	JMP-	
+		0	9	0247	09	ADL of FIRST	
+		0	2	0248	02	ADH of FIRST	
N.B. End of main program.							
+		A	0	0249	A0	LDY #; subroutine	<b>SHIFT</b>
+		0	4	024A	04	04 in Y-index register	
+		0	6	024B	06	ASLZ	<b>SHIFT1</b>
+		F	9	024C	F9	INH (00F9)	
+		2	6	024D	26	ROLZ	
+		F	A	024E	FA	POINTL (00FA)	
+		2	6	024F	26	ROLZ	
+		F	B	0250	FB	POINTH (00FB)	
+		8	8	0251	88	DEY	
+		D	0	0252	D0	BNE	
+		F	7	0253	F7	go to SHIFT1	
+		0	5	0254	05	ORAZ	
+		F	9	0255	F9	OR INH	
+		8	5	0256	85	STAZ	
+		F	9	0257	F9	INH (00F9)	
+		6	0	0258	60	RTS return to main program	
+		F	8	0259	F8	SED decimal arithmetic subroutine	<b>ADD</b>
+		1	8	025A	18	CLC	
+		A	5	025B	A5	LDAZ	
+		0	0	025C	00	ADL of B10 (0000); B10 in accumulator	
+		6	5	025D	65	ADCZ	
+		0	3	025E	03	ADL of B20 (0003); accumulator = B10 + B20	
+		8	5	025F	85	STAZ	
+		0	6	0260	06	ADL of R0; accumulator → R0	
+		A	5	0261	A5	LDAZ	
+		0	1	0262	01	ADL of B11 (0001); B11 in accumulator	
+		6	5	0263	65	ADCZ	

+	0	4	0264	04	ADL of B21 (0004); accumulator = B11 + B21
+	8	5	0265	85	STAZ
+	0	7	0266	07	ADL of R1 (0007); accumulator → R1
+	A	5	0267	A5	LDAZ
+	0	2	0268	02	ADL of B12 (0002); B12 in accumulator
+	6	5	0269	65	ADCZ
+	0	5	026A	05	ADL of B22 (0005); accumulator = B12 + B22
+	8	5	026B	85	STAZ
+	0	8	026C	08	ADL of R2 (0008); accumulator → R2
+	D	8	026D	D8	CLD back to binary
+	6	0	026E	60	RTS return to main program
+	2	0	026F	20	JSR-subroutine <b>KEYDIS</b>
+	8	E	0270	8E	ADL of SCANDS } monitor
+	1	D	0271	1D	ADH of SCANDS } (1D8E)
+	D	0	0272	D0	BNE
+	F	B	0273	FB	go to KEYDIS
+	2	0	0274	20	JSR- <b>KD</b>
+	8	E	0275	8E	ADL of SCANDS } monitor
+	1	D	0276	1D	ADH of SCANDS } (1D8E)
+	F	0	0277	F0	BEQ
+	F	B	0278	FB	go to KD
+	2	0	0279	20	JSR-
+	8	E	027A	8E	ADL of SCANDS } monitor
+	1	D	027B	1D	ADH of SCANDS } (1D8E)
+	F	0	027C	F0	BEQ
+	F	6	027D	F6	go to KD
+	2	0	027E	20	JSR-
+	F	9	027F	F9	ADL of GETKEY } monitor
+	1	D	0280	1D	ADH of GETKEY } (1DF9)
+	6	0	0281	60	RTS return to main program
+	A	9	0282	A9	LDA #; subroutine <b>CLB1</b>
+	0	0	0283	00	00 → accumulator
+	8	5	0284	85	STAZ
+	0	0	0285	00	accumulator → B10 (= 00)
+	8	5	0286	85	STAZ
+	0	1	0287	01	00 → B11
+	8	5	0288	85	STAZ
+	0	2	0289	02	00 → B12
+	6	0	028A	60	RTS return to main program
+	A	9	028B	A9	LDA #; subroutine <b>CLB2</b>
+	0	0	028C	00	00 → accumulator
+	8	5	028D	85	STAZ
+	0	3	028E	03	00 → B20
+	8	5	028F	85	STAZ
+	0	4	0290	04	00 → B21
+	8	5	0291	85	STAZ
+	0	5	0292	05	00 → B22
+	6	0	0293	60	RTS return to main program
+	A	9	0294	A9	LDA #; subroutine <b>CLDISP</b>

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

**Figure 1. The complete listing for the decimal addition program. It requires 196 memory locations and 594 key operations.**

A number of locations on page zero, however, are reserved for the temporary storage of data by certain monitor routines. These are the 31 locations from 00E1 . . . 00FF. We have already encountered some of them; 00F9 (INH), 00FA (POINTL) and 00FB (POINTH) are used as display buffers. The locations 00EF . . . 00F5 are reserved for the contents of all the internal registers of the microprocessor by the SAVE routine. The rest will be dealt with in book 2.

The whole of page 01 is used as a stack by the processor. However, it is very rare that a program requires the storage of 128 address locations. If the stack is not needed this page can be used to hold the actual program or subroutines. If a certain amount of the stack is required, it is imperative that any program located on page 01 does not run into the stack area for obvious reasons.

There are certain areas which cannot be used for programming purposes.

*5. The 1k EPROM of the Junior Computer contains the monitor program. This uses addresses 1C00 . . . 1FFF. Therefore, data cannot be entered into pages 1C, 1D, 1E and 1F. The monitor routines contained in these pages can however be used by the main program.*

An example of the latter can be seen in figure 1. The subroutine KEYDIS (026F . . . 0281) uses the monitor routines SCANDS and GETKEY.

*6. The 128 bytes of RAM (1/8k) contained in the PIA can be used for programming. This RAM is located at 1A00 . . . 1A7F. Thus, a total of half a page (the first half of page 1A) is available to the user.*

As with page zero, however, there are exceptions. The four locations 1A7A . . . 1A7F are reserved for the NMI and IRQ vectors (see chapter 3). These locations can only be used if the program does not require any interrupt routines.

Certain memory locations on the second half of page 1A (1A80 . . . 1AFB) are required for the operation of the PIA itself. This will be discussed in greater detail in book 2. Under certain conditions these locations can also be used for normal program operation.

### *Displacement*

We have already discovered that we can use the JC to calculate displacement values. By using the monitor routine BRANCH (start address 1FD5) we simply have to enter the low order byte of the address where the jump or branch instruction is located followed by the low order address byte of the location to be jumped (or branched) to. All the displacement values for the program given in figure 1 can be found as follows:

RST	AD				xxxx	xx
1	F	D	5	1FD5	D8	
GO				0000	00	
0	E	0	0	0E00	F0	F0 in location 020F
1	2	1	A	121A	06	06 in location 0213

2	5	3	1	2531	0A	0A	in location 0226
2	9	3	7	2937	0C	0C	in location 022A
5	2	4	B	524B	F7	F7	in location 0253
7	2	6	F	726F	FB	FB	in location 0273
7	7	7	4	7774	FB	FB	in location 0278
7	C	7	4	7C74	F6	F6	in location 027D

RST

Note: If any of the control keys are operated in between displacement calculations the display will show 000000.

As mentioned in chapter three, the monitor program does not have to be used to calculate displacement values. It should be remembered however, that when working out displacement values the hard way, the calculation is made from the low order byte of the address location immediately following the complete jump or branch instruction.

### Initialisation

This has also been mentioned briefly in chapter 3. The initialisation procedure for the program in figure 1 is as follows:

RST		AD		xxxx	xx
1	A	7	A	1A7A	xx
DA		0	0	1A7A	00
+		1	C	1A7B	1C
+				1A7C	xx
+				1A7D	xx
+		0	0	1A7E	00
+		1	C	1A7F	1C

Locations 1A7E and 1A7F have been loaded with 00 and 1C respectively. These memory locations now contain the effective start address (1C00) of the monitor program. This means that if an interrupt request (IRQ) is received, or if a BRK instruction (op-code 00) is encountered, the processor will jump straight back to the monitor program. The same effective address has been loaded into locations 1A7A and 1A7B so that if a non-maskable interrupt occurs (NMI), or if the ST key is depressed, the processor will again jump to the monitor program.

If, of course, there are no BRK instructions in a particular program, and no interrupts are expected, these memory locations need not be loaded.

Once the initialisation is complete and the program has been entered correctly, we can do some calculations:

AD	0	2	0	0	(enter start address)	
GO					(start program)	
2	4	5	6		002456	
+					000000	
4	1	3	2		004132	
DA	(=	=)			006588	
AD	(=	CLEAR)			000000	
1	9	8	5	3	1	198531
+						000000

8	3	2	7	0	2	832702
DA						031233
AD						000000

In the last instance the result was greater than 999999 (overflow) so only the six least significant digits will be displayed.

### **'Throwing a six' with the Junior Computer**

Dice are very cheap and electronic dice cannot be called expensive. This, however, should not deter you from using the Junior Computer for your game of ludo!

The dice is rolled by depressing the + key. As soon as this key is released the roll will end and the display will show the number of moves you can make. The program counts from 0 to 7, but will only display the numbers 1 to 6. The two centre displays will show the actual number while the other four will show 'FF'.

The simplified flow chart of the program is shown in figure 2. The monitor routines SCANDS, TK and GETKEY are also utilised by this program. The program starts by entering FFFFFFF into the display. The next section of the program (SCAN1) determines whether or not a key is depressed, debounces it and tests to see if it was the + key. If the + key was operated the program moves on to COUNT where the actual number to be displayed is generated. This section of the program is repeated until the + key is released.

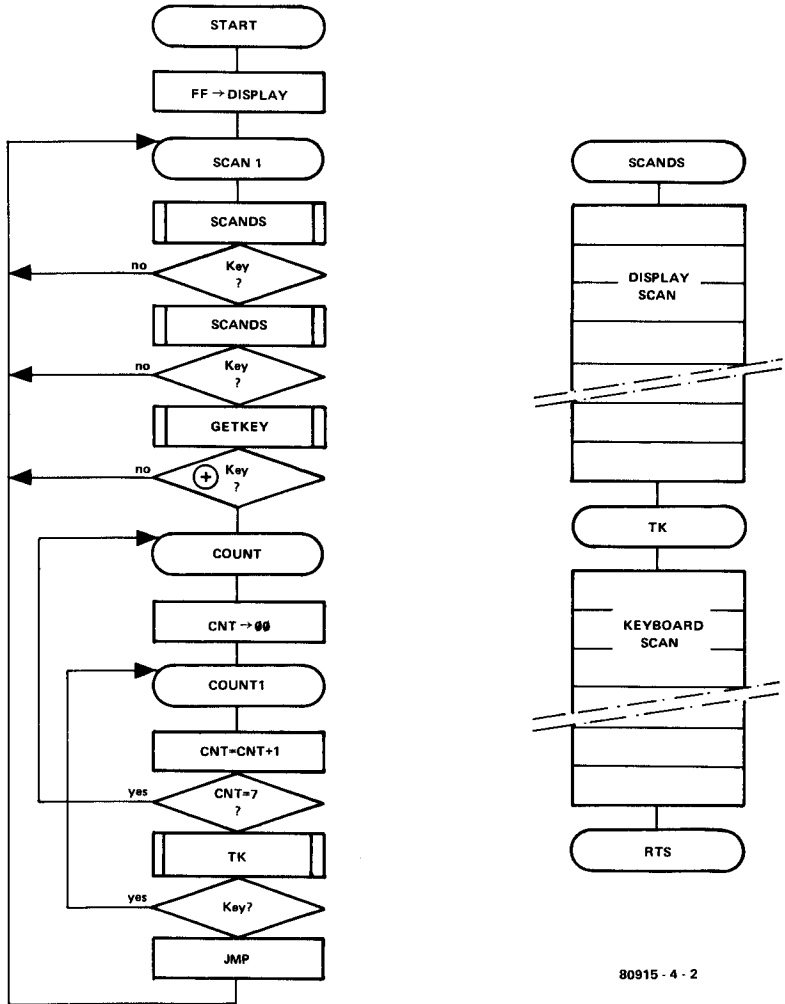
The actual operation of the COUNT section can be seen more clearly in the detailed flow chart of figure 3. POINTL is first loaded with 00 (counter reset) and continually incremented until the value reaches 07 (COUNT1). As soon as this is the case, the counter is reset once more and the procedure continues (for as long as the + key is held down). This operation is performed so fast that there is absolutely no way of knowing what the final value is actually going to be.

The complete listing of the program is given in figure 4. The start address is 0200 as before. A number of comments have been added to help explain the various steps. Once the start address has been entered (AD 0200 GO) the dice can be thrown to your heart's content.

### **Determination of instruction length via software**

As we know (by now) instructions are one, two or three bytes long. The first byte is required for the op-code and the second and third (if present) are needed to determine the data or effective address information.

The op-code consists of two hexadecimal numbers. The first table at the back of this book gives a complete listing of all 256 possible combinations of a two digit hexadecimal number. Where applicable, the table also gives the appropriate mnemonic and address mode. This table is also shown in condensed form in figure 5. The most significant 'nibble' (= four bits of a byte) is shown on the left-hand side and the least significant nibble along the top. The table consists of 29 single byte instructions, 74 double byte instructions, 48 triple byte instructions and 105 empty spaces.

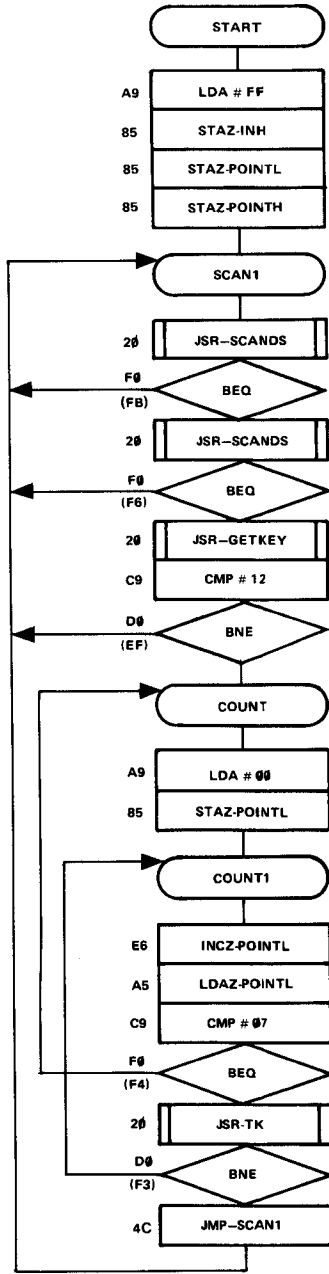


**Figure 2. The simplified flow chart for the 'dice' program. Various monitor routines are used to simplify matters.**

We wish to develop a program that determines whether a particular data byte is:

- the op-code for a single byte instruction
- the op-code for a double byte instruction
- the op-code for a triple byte instruction
- non-existent op-code

This program is more than just an educational example. Practically all



80915 - 4 - 3

Figure 3. Detailed flow chart of the 'dice' program.



	key	AD	address	data	comments
RST		AD	xxxx	xx	
0	2	0 0	0200	xx	
DA		A 9	0200	A9	A9 LDA IMM <b>START</b>
+		F F	0201	FF	FF → accumulator
+		8 5	0202	85	STAZ
+		F 9	0203	F9	FF → INH (00F9)
+		8 5	0204	85	STAZ
+		F A	0205	FA	FF → POINTL (00FA)
+		8 5	0206	85	STAZ
+		F B	0207	FB	FF → POINTH (00FB)
+		2 0	0208	20	JSR- <b>SCAN1</b>
+		8 E	0209	8E	ADL of SCANDS (monitor)
+		1 D	020A	1D	ADH (address 1D8E)
+		F 0	020B	F0	BEQ
+		F B	020C	FB	displacement for branch to SCAN1
+		2 0	020D	20	JSR-
+		8 E	020E	8E	ADL of SCANDS (monitor)
+		1 D	020F	1D	ADH (address 1D8E)
+		F 0	0210	F0	BEQ
+		F 6	0211	F6	displacement for branch to SCAN1
+		2 0	0212	20	JSR-
+		F 9	0213	F9	ADL of GETKEY (monitor)
+		1 D	0214	1D	ADH (address 1DF9)
+		C 9	0215	C9	CMP IMM
+		1 2	0216	12	with 12
+		D 0	0217	D0	BNE
+		E F	0218	EF	displacement for branch to SCAN1
+		A 9	0219	A9	LDA IMM <b>COUNT</b>
+		0 0	021A	00	00 → accumulator
+		8 5	021B	85	STAZ
+		F A	021C	FA	00 → POINTL (00FA)
+		E 6	021D	E6	INC Z <b>COUNT1</b>
+		F A	021E	FA	POINTL + 1 → POINTL
+		A 5	021F	A5	LDA Z
+		F A	0220	FA	POINTL → accumulator
+		C 9	0221	C9	CMP IMM
+		0 7	0222	07	with 07
+		F 0	0223	F0	BEQ
+		F 4	0224	F4	displacement for branch to COUNT
+		2 0	0225	20	JSR-
+		B 1	0226	B1	ADL } of TK (monitor)
+		1 D	0227	1D	ADH } (address 1DB1)

```

+           D   0       0228 D0   BNE
+           F   3       0229 F3   displacement for branch to COUNT1
+           4   C       022A 4C   JMP-
+           0   8       022B 08   ADL  }
+           0   2       022C 02   ADH  } of SCAN1
AD
0   2   0   0       0200 A9   start address
GO
+           FF04 FF   the dice is cast!
+           FF01 FF
+           FF06 FF
+           FF02 FF
etc.

```

Figure 4. The complete listing of the 'dice' program.

assemblers and editors have a subroutine which is very similar to this program (more on this in book 2).

The program for the 'instruction meter' uses a new subroutine called LENACC. The detailed flow chart of this subroutine is given in figure 6. This is the actual 'body' of the program. Upon entry to the subroutine the accumulator contains the (instruction) byte to be worked on. The subroutine ends by storing the length of the instruction in the location called BYTES.

During the subroutine the X index register contains the information concerning the length of the instruction: X = 00 means no op-code; X = 01 means a single byte instruction; X = 02 means a double byte instruction; and X = 03 means a triple byte instruction. Towards the end of the sub-

Figure 5. This table is a condensed version of the instruction code table shown at the back of the book. The column information (the four least significant bits) plays an important role in the program that determines the length of the various instructions.

		Least significant four bits							
		0	1	2	3	4	5	6	7
Most significant four bits	0	BRK (1)	ORA (IND,X) (2)						
	1	BPL (2)	ORA (IND,Y) (2)						
	2	JSR (3)	AND (IND,X) (2)						
	3	BMI (2)	AND (IND,Y) (2)						
	4	RTI (1)	EOR (IND,X) (2)						
	5	BVC (2)	EOR (IND,Y) (2)						
	6	RTS (1)	ADC (IND,X) (2)						
	7	BVS (2)	ADC (IND,Y) (2)						
	8		STA (IND,X) (2)						
	9	BCC (2)	STA (IND,Y) (2)						
	A	LDY # (2)	LDA (IND,X) (2)	LDX # (2)					
	B	BCS (2)	LDA (IND,Y) (2)						
	C	CPY # (2)	CMP (IND,X) (2)						
	D	BNE (2)	CMP (IND,Y) (2)						
	E	CPX # (2)	SBC (IND,X) (2)						
	F	BEQ (2)	SBC (IND,Y) (2)						

routine the section LENEND simply stores the contents of the X index register into memory location BYTES.

The Y index register is loaded with the value of the least significant four bits of the entered byte. This corresponds to the columns shown in figure 5. The Y register functions as an index for the loading of data contained in the look-up table, LENTBL, into the X register. This table contains the length of the associated instructions for each column of figure 5 (00, 01, 02 or 03).

Observant readers will be saying that there will be no problems for columns 1, 3, 5, 6, 7, 8, B, D and F in which the lengths of the instructions are the same, but what about the other seven columns which contain instructions of different lengths? For these columns the table, LENTBL, will not function fully – some extra work has to be done.

### Complex columns

If we ignore the empty spaces, we are left with only double byte instructions in columns 2 and 4, single byte instructions in column A and triple byte instructions in columns C and E. These empty spaces can be eliminated fairly simply, see later on.

The only awkward columns remaining are 0 and 9. Column 0 consists mostly of double byte instructions except for: BRK, RTI and RTS (single byte); JSR (triple byte). Column 9, on the other hand, contains an assortment of double and triple byte instructions.

At the start of the subroutine LENACC the X register is loaded with 01. This is so that the three single byte instructions in column 0 can be filtered out. This is done with three compare and branch instructions in a row. If the input byte is one of the instructions BRK, RTI or RTS the program will jump to LENEND and store the value of the X register (01) in the address location BYTES. If the input byte was not one of the above instructions the X index register is loaded with 03 and a test is carried out to see if it is a JMP instruction. If so, the location BYTES is loaded with 03 (three byte instruction).

The next section of the subroutine filters out the triple byte instructions from column 9. This is done by ANDING the contents of the accumulator

		Least significant four bits								
		B	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)		0
	1	CLC (1)	ORA ABS,Y (3)				ORA ABS,X (3)	ASL ABS,X (3)		1
	2	PLP (1)	AND # (2)	ROL A (1)		BIT ABS (3)	AND ABS (3)	ROL ABS (3)		2
	3	SEC (1)	AND ABS,Y (3)				AND ABS,X (3)	ROL ABS,X (3)		3
	4	PHA (1)	EOR # (2)	LSR A (1)		JMP ABS (3)	EOR ABS (3)	LSR ABS (3)		4
	5	CLI (1)	EOR ABS,Y (3)				EOR ABS,X (3)	LSR ABS,X (3)		5
	6	PLA (1)	ADC # (2)	ROR A (1)		JMP IND (3)	ADC ABS (3)	ROR ABS (3)		6
	7	SEI (1)	ADC ABS,Y (3)				ADC ABS,X (3)	ROR ABS,X (3)		7
	8	DEY (1)		TXA (1)		STY ABS (3)	STA ABS (3)	STX ABS (3)		8
	9	TYA (1)	STA ABS,Y (3)	TXS (1)			STA ABS,X (3)			9
	A	TAY (1)	LDA # (2)	TAX (1)		LDY ABS (3)	LDA ABS (3)	LDX ABS (3)		A
	B	CLV (1)	LDA ABS,Y (3)	TSX (1)		LDY ABS,X(3)	LDA ABS,X (3)	LDX ABS,Y (3)		B
	C	INY (1)	CMP # (2)	DEX (1)		CPY ABS (3)	CMP ABS (3)	DEC ABS (3)		C
	D	CLD (1)	CMP ABS,Y (3)				CMP ABS,X (3)	DEC ABS,X (3)		D
	E	INX (1)	SBC # (2)	NOP (1)		CPX ABS (3)	SBC ABS (3)	INC ABS (3)		E
	F	SED (1)	SBC ABS,Y (3)				SBC ABS,X (3)	INC ABS,X (3)		F

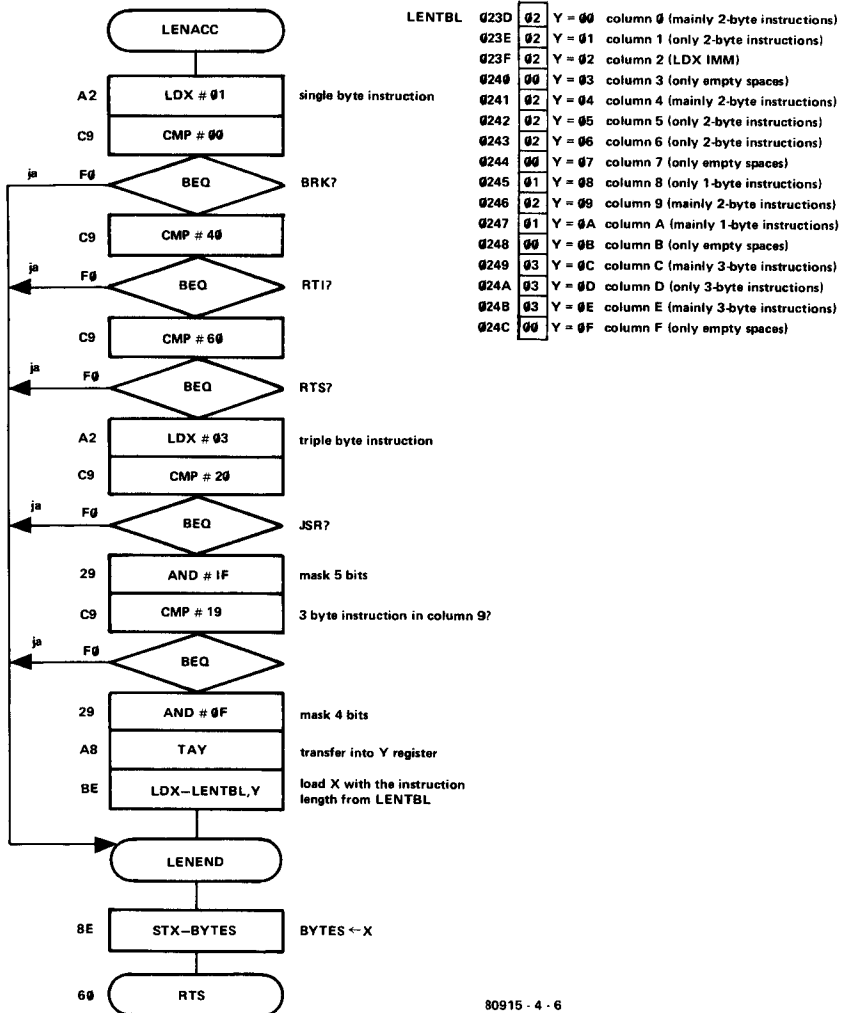


Figure 6. The subroutine LENACC ensures that the correct instruction length information is loaded into the location BYTES.

with 1F (masking). The AND function produces the following results:

$$0 \text{ AND } x = 0 \quad 1 \text{ AND } x = x$$

When the contents of the accumulator are ANDed with 1F the result is:

contents of accumulator : xxxxxxxx

AND with 1F : 00011111

result in accumulator : 000xxxxx

We can see that after the masking process the right-hand five bits remain

unchanged. The four least significant bits are required later in the program to determine which column the input byte belongs to.

It can be seen that the two and three byte instructions in column 9 are on even and odd rows respectively. By masking the input byte as above the fifth bit from the right will also be even (0) or odd (1) depending on the row being examined. The four right-hand bits of column 9 will always be equal to 9 (hexadecimal). Therefore, the total value of the complete byte will be 09 or 19 for even and odd rows respectively. The next stage of the program simply tests for the value of 19 (odd row, triple byte instruction) and if true the contents of the X register (still 03 at this time) are again placed in the location BYTES.

All the complex bytes have now been filtered out. The next step is to AND the accumulator contents with 0F. This removes the (now unwanted) fifth bit leaving the remainder unaltered. The accumulator now contains 0000xxxx. This value is then transferred to the Y index register to determine which column the instruction belongs to. The X register is then loaded with the corresponding instruction length from the table LENTBL. Finally (at LENEND), this value is placed in the location BYTES.

#### Decoding the empty spaces

All the non-existent (zero byte) instructions in columns 3, 7, B and F are decoded by the LENACC subroutine (figure 6). The remaining 41 are

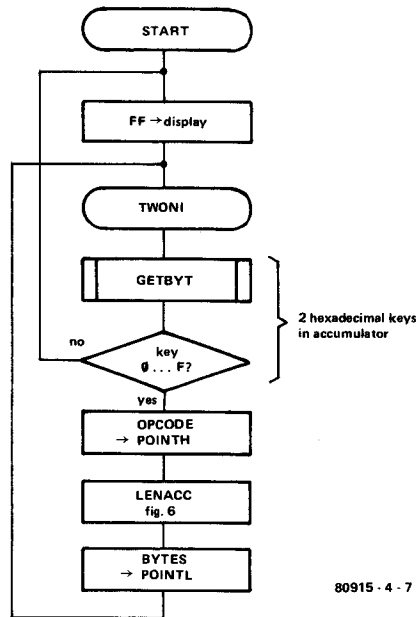
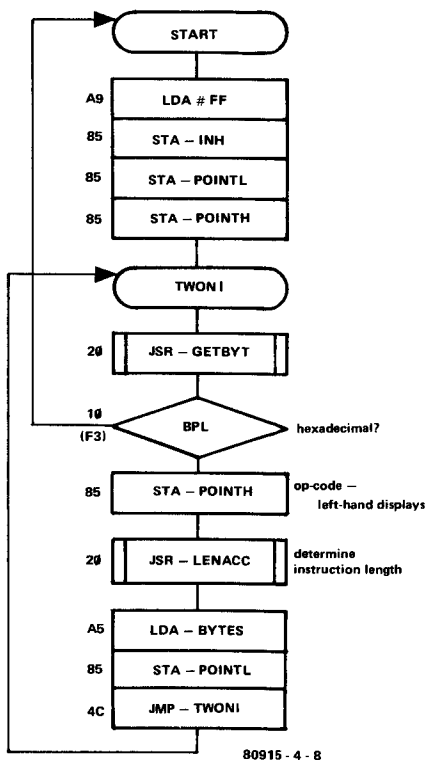


Figure 7. The simplified flow chart of the 'instruction meter' program.



**Figure 8. Detailed flow chart of the 'instruction meter' program.**

simply ignored. To be able to decode all of these the LENACC subroutine can be expanded as follows:

The X index register needs to be loaded with 00. Twenty-six consecutive compare immediate and branch instructions are then required to check whether the entered byte corresponds to one of the empty locations in columns 0, 4, 9, A, C or E. If so, then the branch instruction must direct the program to LENEND. If not then the X register must be loaded with 02 and a test performed to see if the entered byte corresponds to the lonely instruction in column 2 (LDX IMM). If this is the case the program should be directed to the TAY instruction of figure 6.

The simplified flow chart of the 'instruction meter' program is given in figure 7. A more detailed version is shown in figure 8, while the complete program listing is given in figure 9.

Once the program has been started, the display buffers are loaded with FF. A test is then carried out to check whether a key is depressed and if so whether it is a control key or a hexadecimal key. This operation is performed by the monitor subroutine GETBYT which in turn calls up the monitor routine SCANDS.

As soon as two of the hexadecimal keys, 0 . . . F, have been depressed the resulting byte is transferred to POINTH via the accumulator. The sub-routine LENACC is then performed and, upon its return, the contents of BYTES are stored in POINTL. The program then jumps back to the GETBYT (and SCANDS) routine to display the first result and prepare for a second entry.

The start address of the program is once again 0200. The main routine (figure 8) runs from address location 0200 up to and including 0218. The sub-routine LENACC occupies the area 0219 . . . 023C and the following 16 locations (023D . . . 024C) are reserved for the table (LENTBL). The memory location BYTES is situated on page zero (location 00F6). This is simply because the monitor program also contains an instruction length routine (OPLN, start address 1E5C) which uses this location to store the result.

### *A spot of homework*

We have already mentioned that not all of the empty spaces in figure 5 are decoded. We have also mentioned how they can be decoded. As an exercise and a test of your programming expertise we leave it to you to fill in the necessary instructions and make the required modifications to the program. There is no need to send us your results — the Junior Computer will tell you whether you were correct or not!

		key		address	data	comments
RTS	AD			xxxx	xx	
0	2	0	0	0200	xx	start address
DA		A	9	0200	A9	LDA IMM
+		F	F	0201	FF	FF → accumulator
+		8	5	0202	85	STAZ
+		F	9	0203	F9	accumulator → INH (00F9)
+		8	5	0204	85	STAZ
+		F	A	0205	FA	accumulator → POINTL (00FA)
+		8	5	0206	85	STAZ
+		F	B	0207	FB	accumulator → POINTH (00FB)
+		2	0	0208	20	JSR-
+		6	F	0209	6F	ADL of GETBYT (monitor
+		1	D	020A	1D	ADH address 1D6F)
+		1	0	020B	10	BPL; 2 keys pressed?
+		F	3	020C	F3	if not, go back to START
+		8	5	020D	85	STAZ (byte in accumulator)
+		F	B	020E	FB	accumulator (= op-code) → POINTH (00FB)
+		2	0	020F	20	JSR-
+		1	9	0210	19	ADL } of LENACC
+		0	2	0211	02	ADH }

+	A	5	0121	A5	LDAZ
+	E	0	0213	E0	BYTES 00E0 → accumulator
+	8	5	0214	85	STAZ
+	F	A	0215	FA	accumulator → POINTL (00FA)
+	4	C	0216	4C	JMP ABS
+	0	8	0217	08	ADL } of TWONI
+	0	2	0218	02	ADH } of TWONI
+	A	2	0219	A2	LDX IMM; subroutine
					LENACC
+	0	1	021A	01	01 → X; 1-byte instruction
+	C	9	021B	C9	CMP IMM
+	0	0	021C	00	with 00
+	F	0	021D	F0	BEQ BRK?
+	1	A	021E	1A	if so, go to LENEND
+	C	9	021F	C9	CMP IMM
+	4	0	0220	40	with 40
+	F	0	0221	F0	BEQ RTI?
+	1	6	0222	16	if so, go to LENEND
+	C	9	0223	C9	CMP IMM
+	6	0	0224	60	with 60
+	F	0	0225	F0	BEQ RTS?
+	1	2	0226	12	if so, go to LENEND
+	A	2	0227	A2	LDX IMM
+	0	3	0228	03	03 → X; 3-byte instruction
+	C	9	0229	C9	CMP IMM
+	2	0	022A	20	with 20
+	F	0	022B	F0	BEQ JSR?
+	0	C	022C	0C	if so, go to LENEND
+	2	9	022D	29	AND IMM
+	1	F	022E	1F	mask 5 bits
+	C	9	022F	C9	CMP IMM
+	1	9	0230	19	with 19
+	F	0	0231	F0	BEQ; 3-byte instructions in column 9?
+	0	6	0232	06	if so, go to LENEND
+	2	9	0233	29	AND IMM
+	0	F	0234	0F	mask 4 bits
+	A	8	0235	A8	TAY; right-hand nibble → Y
+	B	E	0236	BE	LDX ABS,Y; (Y + 1) length of instruction → X
+	3	D	0237	3D	ADL } of LENTBL
+	0	2	0238	02	ADH } (look up table)
+	8	E	0239	8E	STX ABS
					LENEND
+	E	0	023A	E0	ADL BYTES



+		0	0	023B	00	ADH BYTES
+		6	0	023C	60	RTS return to main program
+		0	2	023D	02	column 0 ; Y = 00 <span style="border: 1px solid black; padding: 2px;">LENTBL</span>
+		0	2	023E	02	column 1 ; Y = 01
+		0	2	023F	02	column 2 ; Y = 02
+		0	0	0240	00	column 3 ; Y = 03
+		0	2	0241	02	column 4 ; Y = 04
+		0	2	0242	02	column 5 ; Y = 05
+		0	2	0243	02	column 6 ; Y = 06
+		0	0	0244	00	column 7 ; Y = 07
+		0	1	0245	01	column 8 ; Y = 08
+		0	2	0246	02	column 9 ; Y = 09
+		0	1	0247	01	column A ; Y = 0A
+		0	0	0248	00	column B ; Y = 0B
+		0	3	0249	03	column C ; Y = 0C
+		0	3	024A	03	column D ; Y = 0D
+		0	3	024B	03	column E ; Y = 0E
+		0	0	024C	00	column F ; Y = 0F
AD						
0	2	0	0			start address
GO						run
		A	9	A902	FF	
		0	3	0300	FF	
		D	2	D202	FF	
		9	E	9E03	FF	
		D	5	D502	FF	
		etc.				

Figure 9. The complete listing of the 'instruction meter' program — see figures 6, 7 and 8.

# 1. Instruction codes in numerical order

Table of the complete set of instruction op-codes in numerical order, 00 . . . FF.  
The unused op-codes have also been listed.

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

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

**Note:** The first three letters after the op-code are the actual mnemonic for the instruction. The possible address mode is indicated by the following letter(s).

- IMM: immediate addressing
- ABS: absolute addressing
- Z: zero page addressing
- A: accumulator addressing
- (IND,X): pre-indexed indirect addressing
- (INDO, Y): post-indexed indirect addressing
- Z,X: zero page indexed addressing (using the X index register)
- Z,Y: zero page indexed addressing (using the Y index register)
- ABS,X: absolute indexed addressing (using the X index register)
- ABS,Y: absolute indexed addressing (using the Y index register)
- IND: indirect addressing
- REL: relative addressing
- IMP: implied addressing

## 2. Instruction listing



The 56 instructions of the 6502 microprocessor in alphabetical order. Each individual instruction can be used in a number of different address modes. This gives an actual total of 151 instruction possibilities.

mnemonic + explanation	address mode (n)	hexa-decimal op-code	number of clock pulses (N)	number of bytes	flags affected
<b>ADC</b> Add memory to accumulator with carry $A + M + C \rightarrow A$ (1)	IMM	69	2	2	NV ---- ZC
	ABS	6D	4	3	
	Z	65	3	2	
	(IND,X)	61	6	2	
	(IND),Y	71	5	2	
	Z,X	75	4	2	
	ABS,X	7D	4	3	
ABS,Y	79	4	3		
<b>AND</b> "AND" memory with accumulator $A \wedge M \rightarrow A$ (1)	IMM	29	2	2	N ---- Z -
	ABS	2D	4	3	
	Z	25	3	2	
	(IND,X)	21	6	2	
	(IND),Y	31	5	2	
	Z,X	35	4	2	
	ABS,X	3D	4	3	
	ABS,Y	39	4	3	
<b>ASL</b> Shift left one bit (accu or memory) $C \leftarrow \boxed{7} \text{ } \text{ } \leftarrow \emptyset$	ABS	0E	6	3	N ---- ZC
	Z	06	5	2	
	A	0A	2	1	
	Z,X	16	6	2	
	ABS,X	1E	7	3	
<b>BCC</b> Branch on carry clear (2) Branch on $C = 0$	REL	90	2	2	-----
<b>BCS</b> Branch on carry set (2) Branch on $C = 1$	REL	B0	2	2	-----
<b>BEQ</b> Branch on result zero (2) Branch on $Z = 1$	REL	F0	2	2	-----
<b>BIT</b> Test bits in memory: $A \wedge M$ $M_7 \rightarrow N; M_6 \rightarrow V$	ABS	2C	4	3	$M_7 M_6$ ---- Z -
	Z	24	3	2	
<b>BMI</b> Branch on result minus (2) Branch on $N = 1$	REL	30	2	2	-----

mnemonic + explanation	address mode (n)	hexa-decimal op-code	number of clock pulses (N)	number of bytes	flags affected
<b>BNE</b> Branch on result not zero (2) Branch on Z = 0	REL	D0	2	2	-----
<b>BPL</b> Branch on result plus (2) Branch on N = 0	REL	10	2	2	-----
<b>BRK</b> Force break forced interrupt	IMP	00	7	1	---1-1-- B  I
<b>BVC</b> Branch on overflow clear (2) Branch on V = 0	REL	50	2	2	-----
<b>BVS</b> Branch on overflow set (2) Branch on V = 1	REL	70	2	2	-----
<b>CLC</b> Clear carry flag 0 → C	IMP	18	2	1	-----0 C
<b>CLD</b> Clear decimal mode; 0 → D	IMP	D8	2	1	----0--- D
<b>CLI</b> Clear interrupt flag; 0 → I	IMP	58	2	1	-----0-- I
<b>CLV</b> Clear overflow flag; 0 → V	IMP	B8	2	1	0----- V
<b>CMP</b> Compare memory and accumulator A-M	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	C9 CD C5 C1 D1 D5 DD D9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N-----ZC
<b>CPX</b> Compare memory and index X X-M	IMM ABS Z	E0 EC E4	2 4 3	2 3 2	N-----ZC
<b>CPY</b> Compare memory and index Y M-Y	IMM ABS Z	C0 CC C4	2 4 3	2 3 2	N-----ZC

mnemonic + explanation	address mode (n)	hexa-decimal op-code	number of clock pulses (N)	number of bytes	flags affected
<b>DEC</b> Decrement memory by one M-1 → M	ABS Z Z,X ABS,X	CE C6 D6 DE	6 5 6 7	3 2 2 3	N-----Z-
<b>DEX</b> Decrement index X by one X-1 → X	IMP	CA	2	1	N-----Z-
<b>DEY</b> Decrement index Y by one Y-1 → Y	IMP	88	2	1	N-----Z-
<b>EOR</b> "Exclusive or" memory with accumulator A ∨ M → A  (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	49 4D 45 41 51 55 5D 59	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N-----Z-
<b>INC</b> Increment memory by one M + 1 → M	ABS Z Z,X ABS,X	EE E6 F6 FE	6 5 6 7	3 2 2 3	N-----Z-
<b>INX</b> Increment index X by one X + 1 → X	IMP	E8	2	1	N-----Z-
<b>INY</b> Increment index Y by one Y + 1 → Y	IMP	C8	2	1	N-----Z-
<b>JMP</b> Jump to new location (PC + 1) → PCL (PC + 2) → PCH	ABS IND	4C 6C	3 5	3 3	-----
<b>JSR</b> Jump to new location saving return address PC + 2 ↓ (PC + 1) → PCL (PC + 2) → PCH	ABS	20	6	3	-----

mnemonic + explanation	address mode (n)	hexa-decimal op-code	number of clock pulses (N)	number of bytes	flags affected
<b>LDA</b> Load accumulator with memory M → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	A9 AD A5 A1 B1 B5 BD B9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
<b>LDX</b> Load index X with memory M → X (1)	IMM ABS Z Z,Y ABS,Y	A2 AE A6 B6 BE	2 4 3 4 4	2 3 2 2 3	N - - - - - Z -
<b>LDY</b> Load index Y with memory M → Y (1)	IMM ABS Z Z,X ABS,X	A0 AC A4 B4 BC	2 4 3 4 4	2 3 2 2 3	N - - - - - Z -
<b>LSR</b> Shift right one bit (memory or accumulator) 0 → <span style="border: 1px solid black; padding: 0 2px;">7</span> 0 → C	ABS Z A Z,X ABS,X	4E 46 4A 56 5E	6 5 2 6 7	3 2 1 2 3	0 - - - - - ZC N
<b>NOP</b> No operation	IMP	EA	2	1	- - - - -
<b>ORA</b> "OR" memory with accumulator AVM → A	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	09 0D 05 01 11 15 1D 19	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
<b>PHA</b> Push accumulator on stack A ↓	IMP	48	3	1	- - - - -
<b>PHP</b> Push processor status on stack; P ↓	IMP	08	3	1	- - - - -
<b>PLA</b> Pull accumulator from stack A ↑	IMP	68	4	1	N - - - - - Z -

mnemonic + explanation	address mode (n)	hexa-decimal op-code	number of clock pulses (N)	number of bytes	flags affected
<b>PLP</b> Pull processor status from stack; P↑	IMP	28	4	1	(reset)
<b>ROL</b> Rotate one bit left (memory or accumulator) 	ABS Z Z,X ABS,X A	2E 26 36 3E 2A	6 5 6 7 2	3 2 2 3 1	N - - - - - ZC
<b>ROR</b> Rotate one bit right (memory or accumulator) 	ABS Z A Z,X ABS,X	6E 66 6A 76 7E	6 5 2 6 7	3 2 1 2 3	N - - - - - ZC
<b>RTI</b> Return from interrupt PC↑; P↑	IMP	40	6	1	(reset)
<b>RTS</b> Return from subroutine PC↑; PC + 1 → PC	IMP	60	6	1	- - - - -
<b>SBC</b> Subtract memory from accumulator with borrow (3) A-M-C → A	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	E9 ED E5 E1 F1 F5 FD F9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - ZC
<b>SEC</b> Set carry flag 1 → C	IMP	38	2	1	- - - - - 1
<b>SED</b> Set decimal mode	IMP	F8	2	1	- - - - 1 - - - D
<b>SEI</b> Set interrupt disable; 1 → I	IMP	78	2	1	- - - - 1 - - I
<b>STA</b> Store accumulator in memory A → M	ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	8D 85 81 91 95 9D 99	4 3 6 6 4 5 5	3 2 2 2 2 3 3	- - - - -

mnemonic + explanation	address mode (n)	hexa-decimal op-code	number of clock pulses (N)	number of bytes	flags affected
<b>STX</b> Store index X in memory; X → M	ABS Z Z,Y	8E 86 96	4 3 4	3 2 2	-----
<b>STY</b> Store index Y in memory; Y → M	ABS Z Z,X	8C 84 94	4 3 4	3 2 2	-----
<b>TAX</b> Transfer accumulator to index X A → X	IMP	AA	2	1	N-----Z-
<b>TAY</b> Transfer accumulator to index Y A → Y	IMP	A8	2	1	N-----Z-
<b>TSX</b> Transfer stack pointer to index X S → X	IMP	BA	2	1	N-----Z-
<b>TXA</b> Transfer index X to accumulator X → A	IMP	8A	2	1	N-----Z-
<b>TXS</b> Transfer index X to stack pointer X → S	IMP	9A	2	1	N-----Z-
<b>TYA</b> Transfer index Y to accumulator Y → A	IMP	98	2	1	N-----Z-

Notes:

- (1) Add 1 to N if page boundary is exceeded.
- (2) Add 1 to N if jump occurs to the same page;  
add 2 to N if jump occurs to another page
- (3) Borrow = NOT carry (C)

IMM: immediate addressing

ABS: absolute addressing

Z: zero page addressing

A: accumulator addressing

IMP: implied addressing

(IND,X): pre-indexed indirect addressing

(IND),Y: post-indexed indirect addressing

Z,X: zero page indexed addressing (X index register)

Z,Y: zero page indexed addressing (Y index register)

ABS,X: absolute indexed addressing (X index register)

ABS,Y: absolute indexed addressing (Y index register)

REL: relative addressing

IND: indirect addressing



### 3. Hex dump of the monitor program

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

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

This is the complete (condensed) listing of the monitor program contained in the EPROM (IC2) in hexadecimal form. To be precise, only the machine code is listed. The first column of the table consists of addresses, whilst all the remaining figures represent data. For example, the data byte 85 is contained in location 1C00. The next figure, F3, is the contents of the following location (1C01).

## 4. Pin assignment of the connectors

### The pin assignment of the expansion connector

For practical reasons the connector has been rotated 90°. The a-connections are the ones closest to the board. Seen from the side therefore the numbers will run from right to left. The blocks above 32c and below 1c indicate the polarizing notches enabling the male plug to be connected correctly.

32a	ground	32a	o	o	32c	32c	ground
31a	RAM-R/W	31a	o	o	31c	31c	NC
30a	$\Phi$ 1	30a	o	o	30c	<del>30c</del>	
29a	K1	29a	o	o	29c	29c	R/ $\bar{W}$
28a	NC	28a	o	o	28c	28c	K2
27a	$\Phi$ 2	27a	o	o	27c	27c	NC
26a	A1	26a	o	o	26c	26c	A0
25a	A3	25a	o	o	25c	25c	A2
24a	A5	24a	o	o	24c	24c	A4
23a	A7	23a	o	o	23c	23c	A6
22a	A9	22a	o	o	22c	22c	A8
21a	A11	21a	o	o	21c	21c	A10
20a	A13	20a	o	o	20c	20c	A12
19a	A15	19a	o	o	19c	19c	A14
18a	-5 V	18a	o	o	18c	18c	K3
17a	K4	17a	o	o	17c	17c	+12 V
16a	16c	16a	o	o	16c	16c	
15a	K5	15a	o	o	15c	15c	K6
14a	K7	14a	o	o	14c	14c	S0
13a	NC	13a	o	o	13c	13c	NC
12a	IRQ	12a	o	o	12c	12c	NMI
11a	NC	11a	o	o	11c	11c	NC
10a	D7	10a	o	o	10c	10c	D6
9a	D5	9a	o	o	9c	9c	D4
8a	D3	8a	o	o	8c	8c	D2
7a	D1	7a	o	o	7c	7c	D0
6a	NC	6a	o	o	6c	6c	NC
5a	RES	5a	o	o	5c	5c	RDY
4a	ground	4a	o	o	4c	4c	ground
3a	NC	3a	o	o	3c	3c	NC
2a	NC	2a	o	o	2c	2c	NC
1a	+5 V	1a	o	o	1c	1c	+5 V

64 206

ROMH

## The pin assignment of the port connector

NC	30	o	o	31	NC
NC	28	o	o	29	NC
PB3	26	o	o	27	NC
PB1	24	o	o	25	PB2
PB7	22	o	o	23	PB0
PB5	20	o	o	21	PB6
NC	18	o	o	19	PB4
NC	16	o	o	17	+5 V
NC	14	o	o	15	NC
NC	12	o	o	13	NC
PA7	10	o	o	11	NC
PA5	8	o	o	9	PA6
PA3	6	o	o	7	PA4
PA1	4	o	o	5	PA2
+5 V	2	o	o	3	PA0
			o	1	ground