This is a book for people who want to know how their Spectrum ticks. It will guide you, even if you have no knowledge of electronics, to an understanding of what is going on inside the case of your computer.

The book is in two sections; the first deals with the fundamental principles behind computer design. The second part takes a look at how the Sinclair Spectrum is structured and, in particular, at how the screen display, keyboard and sound facilities are implemented.

The book includes practical and helpful programs that illustrate the points discussed. You are gently introduced to the world of machine code programming with a monitor program and 'hands on' experiments.

Inside your Spectrum will take curious beginners to the point where they can tackle those forbidding projects with a clear understanding of how their computer works.

Jeff Naylor has written several commercial computer games, including some for the Spectrum. He is also a regular contributor to Popular Computing Weekly. Diane Rogers has worked mainly in theatre and television stage management. Her contribution to the partnership has been more literary than technical.

GB £ NET +006.95

ISBN 0-946408-35-1

00695

9 780946 408351

SUNSHINE

ISBN 0 946408 35 1

£6.95 net

JEFF NAYLOR AND DIANE ROGERS INSIDE YOUR SPECTRUM SUNSHINE

inside your spectrum

an introductory guide to the hardware

jeff naylor & diane rogers

inside your spectrum

an introductory guide to the hardware

jeff naylor & diane rogers

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2R 3LD

British Library Cataloguing in Publication Data
Naylor, Jeff
Inside your Spectrum
1. Sinclair ZX Spectrum (Computer)
I. Title II. Rogers, Diane
001.64'04 QA76.8.S625

ISBN 0-946408-35-1

Cover design by Grad Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset, printed and bound by Trintype, Nene Litho and Woolnough Bookbinding, Wellingborough, Northants

CONTENTS

Contents in detail

Preface

Although some experience of programming the Spectrum in BASIC would be helpful, this book makes no demand on its readers in terms of previous knowledge of computers or electronics: all that is required is a desire to learn. More advanced readers may find that the earlier chapters are a useful clarification of essential background and a preparation for the more complex issues dealt with later on.

The greatest benefit will be felt if you read with your computer close to hand, so that experimenting is naturally encouraged. Don't be afraid to wander on your own: the best remembered facts are those discovered for yourself.

Warning: When entering the programs in this book, please be careful. The quality of print from computer printers means that some characters look very similar — commas, in particular, may look very like full stops. Please ensure that you key in the correct characters.

Part 1

First Principles

CHAPTER 1
Electronics of the Digital Kind

When you discover that the apparently inscrutable performance of modern computers is achieved by using 'digital electronics', do not be panicked into assuming that this will make them any harder to understand than a conventional electronic device such as a radio. Indeed, if you have tried to comprehend electronics at some time in the past and struggled to understand such things as sine waves, modulation, and capacitance, then take heart: the theory behind digital techniques requires less in the way of abstract concepts or knowledge of mathematics. It is size (a great deal crammed into a small package) and speed of operation that make modern computers so powerful.

So what is the difference between digital electronics and the more conventional kind? Broadly speaking, a digital circuit is only concerned with the presence or absence of electricity; in other words, whether parts of itself are ON or OFF. The exact amount of electricity present, providing it falls within certain limits, is unimportant.

Conventional circuits tend to be much more precise. For example, in a hi-fi system the quantities of electricity flowing out to the loudspeakers reflect exactly the changing levels of sound that the circuit is trying to imitate. In a badly designed system, the levels of electricity may not be controlled accurately enough, resulting in distortion.

Charles Babbage is sometimes known as the father of modern computing because he designed the first 'Analytic Engine' in the 1830s. It was indeed an engine; with plungers, levers and cogs interacting in a visible, tangible way. Comprehending the behaviour of a mechanical device is easy on the brain because there is nothing abstract to understand: pulling this lever operates that plunger because they are attached to each other by a wire which we can see.

In order to come to terms with an electronic device, however, it is common to envisage a picture of electricity behaving like water, for instance, or even little men rushing around. There is nothing to be ashamed of in using these analogies, but they are not accurate enough to cope with complex models. A circuit consisting of a battery, bulb and switch can be resolved in these terms, but try to apply your analogy to a television set and not only will you fail to grasp how it works, you may begin to doubt the behaviour of water, or even of little men!

The nature of electricity

In order to have a picture of the nature of electricity, it is first necessary to appreciate a few simple facts about matter. All of these would take some time to prove, and in the final analysis you would still be taking my word for it, just as I am taking someone else's. Let us, therefore, accept that matter, whether it is gas, plastic or orange juice, is made up of extremely small particles called 'atoms', a simple definition of which is 'the smallest particle that can exist'. In the past the definition would have gone on to say that atoms are indivisible, but we now know that this is not the case (however, nuclear fission is not the concern of this book). There are well over a hundred types of atoms, and some are more common than others.

Materials which consist of only one type of atom are called 'elements': they include such familiar substances as carbon, iron and oxygen. The atoms of one element differ from those of another by their size and structure, varying from the simple hydrogen atom to heavyweights such as uranium. Other substances consist of different types of atom grouped together, sometimes as a simple mixture of elements, but often as the result of the atoms bonding together and forming what is known as a 'molecule'. Two atoms of hydrogen, for example, locked together with one atom of oxygen make up a molecule of water.

Through a long process of educated guesswork, followed up by experiments to prove their theories true, scientists have established a picture of what makes up an atom. It is in the construction of the different types of atom that the key to electricity lies. Each atom consists of a nucleus of a number of 'protons' and 'neutrons', which is the so-called 'indivisible core', the prising open of which leads to the science of particle physics.

Around this core orbit 'electrons'. Each type of atom has a different requirement of these to make it ideally balanced: hydrogen has a meagre one, whilst others can have dozens. However, some atoms can sustain a small imbalance in their construction, and either host additional electrons or relinquish some of their normal quota. As you have probably guessed from their name, it is these electrons that lead to all electrical phenomena from lightning to digital watches.

If we consider the behaviour of electrons in a battery, an environment which is easy to visualise, we can begin to understand how electricity can be made to be useful. Batteries are constructed in such a way as to be a source of electricity. They contain two areas, one with a surplus of electrons, the other with a deficiency. Some batteries, such as those found in cars, can be recharged when the imbalance between the two areas has equalled out; others, such as torch cells, contain chemicals which cannot easily be rejuvenated.

The two areas are connected to terminals in the outside of the case.

The terminal which leads to the area containing the surplus of electrons is called the 'negative' terminal, and it is marked with a minus sign. At first glance this may seem illogical but, as it is how the early experimenters labelled their batteries, the tradition is too well established to be tampered with now. This negative terminal is the jumping-off point for the electrons which are eager to get across to the other terminal, marked with a plus sign and called 'positive'.

You are probably aware that one of the units used in the measurement of electricity is the 'volt': this can be thought of as the pressure on the electrons to move away from their present, overcrowded, host atoms to find a more welcoming home. Electrons and the nucleus of atoms behave in a similar way to magnets: the north pole of a magnet is attracted to the south pole of another magnet, and vice versa, whilst two north poles have the reverse effect, actually repelling each other. Electrons don't like other electrons to be present: if an atom is hosting too many electrons they tend to put pressure on each other to go away and, if the opportunity occurs, the surplus tend to leave. Do not think of voltage as the number of spare electrons. The latter is an indication of the capacity of the battery, and therefore relates to the length of time the battery would keep working. Voltage is the force with which an electron is trying to escape.

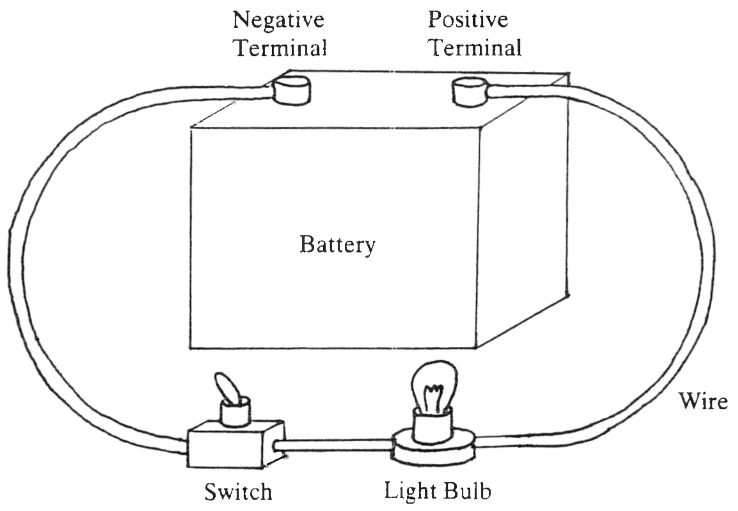


Figure 1.1

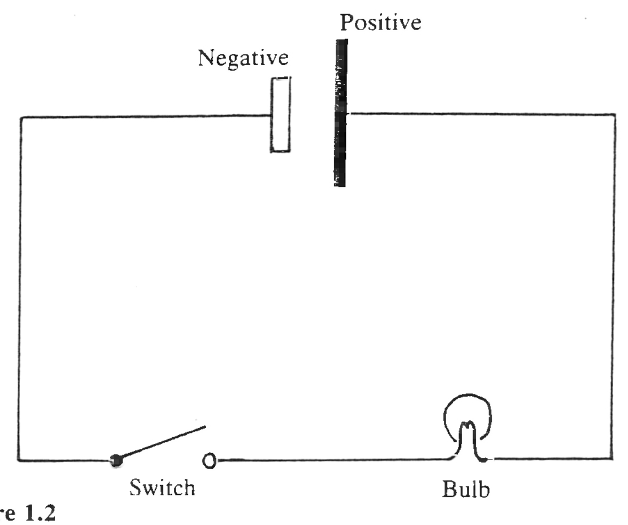


Figure 1.2

Analogue circuits

It is time to look at our first circuit. If you study **Figure 1.1** you should recognise some of the components. **Figure 1.2** is the same thing drawn as a circuit diagram, using symbols which make the drawing easier and the overall picture simpler to understand, once you know what the symbols are meant to represent.

As you can see, the battery is connected to the other parts of the circuit by wire. Electrons cannot normally travel through air, except when the voltage is very high, as in the case of lightning. Some materials do allow the relatively easy movement of free electrons and are said to be 'conductive': if a substance only allows the flow grudgingly, causing the electrons to work hard for their passage, then it is called 'resistive'. In fact, all conductors have some resistance, but in the case of the connecting wire made of metal (probably copper) the resistance is so small as to be irrelevant.

The light bulb contains a particular type of wire, known as the 'filament', which has sufficient resistance to ensure that the electrons, jostling past the atoms of the filament, cause it to heat up until it glows and gives off light. In order to prevent the wire burning up, it is encased in a glass bubble which is filled with inert gases so that the filament has nothing with which to react. It is the amount of flow through the bulb which determines how much work is done and therefore how much light the filament gives off. This flow is called 'current' and we measure it in 'amps'.

The final component is the switch, the operation of which is accurately expressed by its symbol — it provides a mechanical gap in the conductor which can be made good when the switch is closed. When the switch is open, the entire voltage difference of the circuit is present between its two contacts.

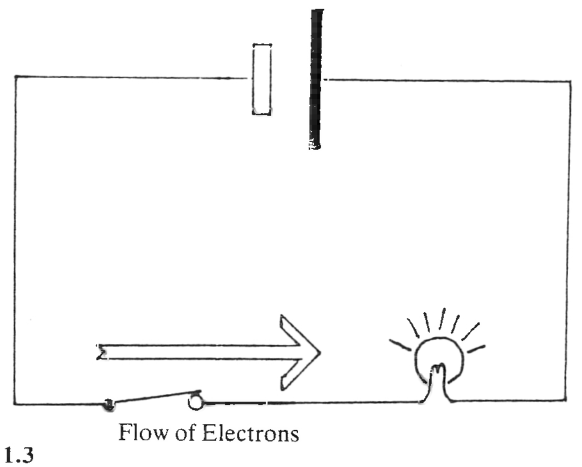


Figure 1.3

Figure 1.3 shows what happens when the switch is closed. The voltage present causes electrons to travel along the wire, through the now closed switch, until they reach the bottleneck caused by the resistive nature of the bulb's filament. After passing through the filament, the electrons are on the home straight and finally they arrive at their goal, the positive terminal of the battery, where they find an atom which is not overcrowded around which they can orbit.

By now you should have a picture of how the electric circuit in a torch behaves. Perhaps you may like to compare it with a simple water analogy, which at this level of complexity holds true.

The negative terminal of the battery can be though of as a tank of water stored in a loft. Voltage relates to the height of the tank and therefore the pressure that is trying to force water through the pipes. The amount of water in the tank is the capacity of the battery. If there is a stopcock it will act like a switch, cutting off the flow. Any narrow pieces of pipe would restrict the flow in the same manner as a bulb limits current.

You will be able to see that the three parameters of our simple circuit, voltage, current and resistance, are related to each other. Connect a bulb with a higher resistance into the circuit, and the flow of current will be smaller; use a battery with a higher voltage and more flow will ensue.

Their relationship is fixed in mathematical terms as 'Ohm's law'. This states that the current in amps flowing through a circuit is equal to the voltage, in volts, divided by the resistance of the circuit, measured in 'ohms'. This means that we can calculate any one of these values if we know the other two. Nowhere in this book will you be expected to perform such mathematics, but I trust that when I use the words voltage, current and resistance, I can safely visualise comprehending faces rather than blank stares.

Digital circuits

Now we can neatly side-step much electrical theory and move on to more practical matters. In glancing through any modern electronics catalogue, you are likely to find a large section devoted to 'digital integrated circuits': these go under names such as TTL and CMOS, which do little to promote understanding. CMOS stands for complementary metal oxide semiconductor, and this indicates how they are made rather than what they do. Transistor transistor logic is a little more helpful in explaining TTL. These components are the building blocks of logic circuits: some of them are already prefabricated into fairly complex circuits themselves. Even the largest computer components use the same building blocks within a single package — there are simply more of them.

In order to follow the processes involved in a digital circuit, it is necessary to know something about 'semiconductors'. These are substances which do not occur in nature. They are manufactured from highly-refined materials such as germanium and silicon to have particular, very useful, properties. The simplest device we can build from semiconducting materials is a 'diode', which has two terminals and passes current in only one direction. Even more useful is the 'transistor', the resistance of which varies when a voltage is applied at a *third* terminal. If we revert to the water analogy, the transistor is like a tap, and the voltage present at the third terminal can be thought of as a hand on the tap, controlling the flow.

In analogue circuits the precision of these devices is critical, as the amount of flow allowed through the transistor is proportional to the control voltage. In digital circuits, however, the next stage in the circuit is only concerned with whether or not there is voltage present, so the transistors can be less precise in their manufacture — they are only acting as electrically-controlled switches.

I have often talked about things being on or off, and I should also mention some other terms. If a positive voltage is present then that part

of the circuit is said to be 'high', or at 'logic level one': conversely, the negative side of the circuit or supply is said to be 'low', or at 'logic level zero'. There is another possibility, which occurs when we are examining an area not connected to either terminal of the voltage supply, and this is said to be 'floating'.

It will be useful to describe in detail the function of one logic chip, the '7409 quad two-input AND gate'. Sometimes the names of these parts can indicate what they do and in this case one clue lies in the word 'gate'.

A 'logic gate' is a point where a decision is made. Do not assume that any thought is involved — given the same set of circumstances the same gate will always behave in the same way. The 7409 contains four such gates (hence the 'quad' in its name) and each gate has two input terminals and one output terminal.

It is called an AND gate because it behaves in the following manner. If its first input is high *and* and its second input is also high, then the output will be high. If either, or both inputs are low, then the output will be low.

You may be wondering why there are four gates in one package. Simply, the cost is so low that the manufacturers might as well make the most of the space. The chip has 14 pins connecting it to the outside — four sets of three as access to the gates, and two for the voltage supply. The 74 series runs on a 5 volt positive supply, the negative terminal being called 'ground', or GRN for short.

		Input One	
		Low	High
Input Two	Low	Low	Low
	High	Low	High

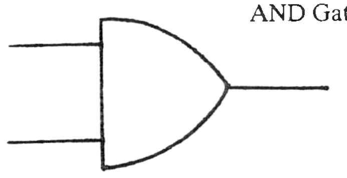


Table 1.1

A useful technique in the study of logic is the drawing up of a truth table. This is a method of assessing how a logic circuit operates and it is worthwhile making use of a simple one in order to define the operation of an AND gate.

Take a look at **Table 1.1**. Across the top of the table are the possible states of input one of an AND gate, and down the side those of the second input. Follow the 'input two low' across to where it meets the 'input one low' column and you read 'low' — the state of the output when both inputs are low.

		Input One	
		Low	High
Input Two	Low	Low	High
	High	High	High

OR Gate

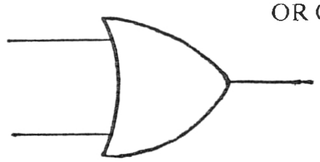


Table 1.2

Now study **Table 1.2**. This describes the behaviour of an OR gate; where the output is high if either input one *or* input two is high. With **Tables 1.3** and **1.4** we come across two new words: 'NAND' and 'NOR'. However, these are only AND and OR with an N prefix, which stands for 'not'. If you made a truth table for an AND gate but you lied each time you wrote an answer, the final result would describe the operation of a NAND gate. One way of achieving this electronically would be by using a circuit called an 'inverter' attached to the output of an AND gate. This is one of the simplest logic devices available as it has only one input and output — as its name implies, the output is an inverted reflection of its input, it swops high for low and vice versa. We can also pass the output of an OR gate through an inverter to create a NOR gate.

Now take a good look at **Figure 1.4.** Spend a little time working out what will happen when it is turned on, and when the switches are pressed. The circuit represents a latch, or simple 'flip-flop', and it shows how a pulse, in the form of someone pressing the switch for a moment (ie a logic high being applied briefly to one input of gate one) achieves a

		Input One	
		Low	High
Input Two	Low	High	High
	High	High	Low

NAND Gate

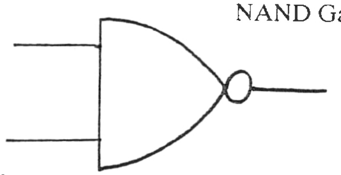


Table 1.3

		Input One	
		Low	High
Input Two	Low	High	Low
	High	Low	Low

NOR Gate

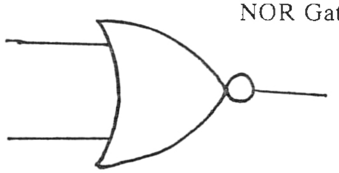


Table 1.4

permanent result. The circuit consists of two NOR gates, two switches, and two components that greatly impede the flow of current, namely resistors.

Study how the switch and resistor are arranged. If the switch is open, there will be zero volts at the point where the wire leading to one input of the gate is connected to the switch and resistor. Close the switch, and

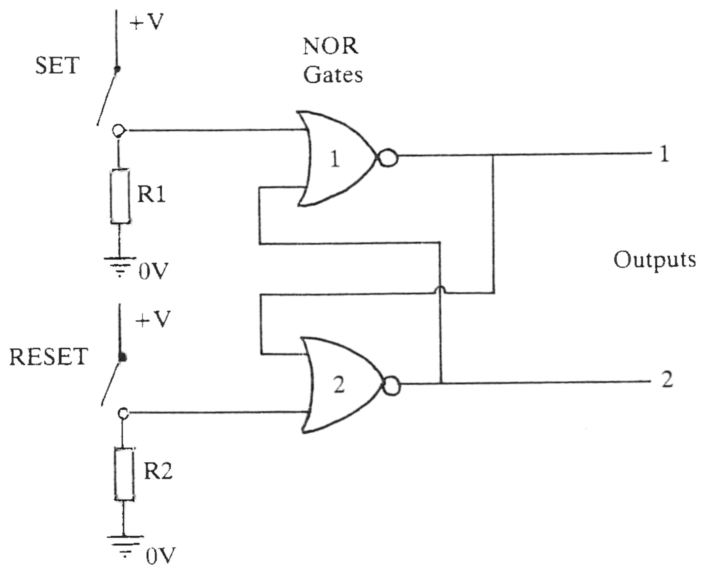


Figure 1.4

current will flow through the switch and resistor so that the point connected to the gate will be at a high voltage.

Let's follow the operation of the gates. When first connected and before either switch is closed, both inputs to gate one will be at zero volts. Check this occurrence against the NOR gate truth table and you can see that the results will be a high voltage at the output terminal. The inputs to gate two will be low in the case of the terminal connected to the switch and high for the one attached to output one, resulting in a low output at output two. This ties up with the second input to gate one, keeps it at a low voltage and so maintains the status quo, ie output one high, output two low.

Now, imagine what happens when you close the switch marked 'set'. The first NOR gate now has a high and a low on its inputs, resulting in a low output. This also affects gate two, as it now has two low inputs, so its output goes high. This has an effect on gate one, by making both inputs high; but, as you can see from the truth table, its output remains low. We have a new, but stable situation with the two output values swopped over.

Let's see what happens when we reopen the set switch. Gate one has one low and one high input, so the output remains low. The second gate

is not affected and the outputs remain the same. In order to make the outputs flip back, we need to press the reset switch. Try tracing the logic levels that will result. This circuit would be said to remember which switch was pressed last — if it was 'set' then output one will be high. Extra circuits can be added to make this device work as a store: it can actually remember a logic level, a property which, as you will see later, is very useful indeed.

CHAPTER 2
Numbers and Data

In the days of telegraph, communication along the wires was in digital form, with morse code using dots and dashes rather than ones and zeros. The process was slow and cumbersome as the information which was to be sent had to be coded, passed on as a series of dots and dashes, and decoded at the other end of the wire. If only something more complex than the on/off signal could be sent, it would all be a lot quicker: for example, by transmitting 26 uniquely differing signals along the wire, the whole alphabet would be able to be represented.

Analogue and digital techniques would tackle this problem in different ways. An analogue solution might be to vary the length of the dashes, or to place a voltage of changing levels on the line — continue along this path and you will end up with the telephone.

The digital solution, however, is to add more lines. If one line can be at one or zero, then two lines can indicate any one of four conditions — both lines off, line one on and line two off, line one off and line two on, or both lines on. Three lines can offer eight permutations — the four available from two lines with the third line on or off in each situation.

Imagine a system of communication which consists of five wires connecting two devices able to code and decode information. You would be able to have 32 buttons, labelled with letters of the alphabet, for your message. A fairly simple digital circuit could then codify this into a unique five-line signal of ons and offs to transmit to the receiver. This would decode the message and it could respond by lighting a particular bulb. This kind of device could, in computer jargon, be called a 'five-bit parallel data transfer system': that is, it passes information (data) of five 'bits' (or ones and zeros) along five wires running in parallel.

Binary and hexadecimal numbers

Let us now consider the same view in a more theoretical way. How people think of numbers depends on their education. I was not introduced to the concept of numbers to a different base until I was too set in my ways to grasp it quickly, but modern maths now teaches early on the idea of numbers to bases other than ten. For those puzzled even by the word 'base', an explanation is required.

We have ten numbers that only need one figure to represent them, zero to nine. Each time we reach ten we add to the next column, and so ten is said to be the base. But why only ten figures? If the human race had evolved with 16 fingers and thumbs, we might have had six more single figures to represent the numbers 10 to 15, and then we should have worked to the base 16.

Table 2.1: Decimal to Hexadecimal Conversion

DEC	HEX	DEC	HEX
0	00	16	10
1	01	32	20
2	02	64	40
3	03	128	80
4	04	256	0100
5	05	512	0200
6	06	1024	0400
7	07	2048	0800
8	08	4096	1000
9	09	8192	2000
10	0A	16384	4000
11	0B	32768	8000
12	0C	65535	FFFF
13	0D		
14	0E		
15	0F		

Try writing out numbers, say from 1 to 30, using the base 16, with the letters A to F to represent 10 to 15, and compare your results with **Table 2.1.** You are using hexadecimal numbers, a numbering system often favoured by machine code programmers because it is easier to translate into 'binary', which is the way in which the computer itself handles numbers. As its method of representing data only permits its columns to contain two different figures, 0 and 1, the computer needs to use more columns in order to represent a number greater than 1. Instead of units, tens, hundreds, thousands and so on, the columns go up in value by the power of two, ie 2, 4, 8, 16, etc. The Spectrum, together with the majority of home computers, uses a data structure of eight such columns, and can therefore handle numbers up to 255: this is the decimal number which, when it is a binary number (or a number to the base of two) is represented as 11111111.

I've already said that each voltage (one or zero) is called a bit, which is short for binary digit. The columns or lines which convey the eight separate bits around the computer are known collectively as the 'data bus'. Eight bits of information grouped together to represent a single piece of data are called a 'byte'; and eight-bit micros handle one byte of data at a time. They can store, and communicate with other devices, using numbers in the range 0–255.

The value of using hexadecimal (hex for short) is that one byte can be expressed as a two-column number in the range 00 to FF. With a little practice, you should be able to convert mentally a binary number into a hex number and vice versa, so that you can visualise how the individual lines of the data bus are behaving without the inconvenience of reading and writing eight-column numbers. If you do find hex absolutely impossible, you can get away without using it, but you will still need to become familiar with binary.

I have included a program which can be used either to demonstrate the relationship between binary and decimal, or to calculate the binary form of a decimal number. This program, and all the others in this book, is marked extensively with 'remark statements' (REM), which you can omit if you want to save time.

Although this first program contains no machine code, later ones will do so and you may lose them completely if you run them with a typing error included. In order to prevent this, I strongly recommend that you save and verify anything that takes more than a minute to enter. As all the programs start from line 10, you can do this using the instruction 'Save "name" line 10': then they will run automatically when reloaded (see Manual, Chapter 20).

Program 2.1: Binary demonstration
```
  10 REM    Binary demonstration
  11 REM    program
 100 REM
 101 REM    Set up
 103 REM
 110 GO SUB 400: LET number=0: GO SUB 30
0
 115 REM
 116 REM    Menu
 117 REM
 120 PRINT  AT 20,1;"Press  "; INVERSE 1;
"C"; INVERSE 0;"alculate or  "; INVERSE 1
;"D"; INVERSE 0;"emonstrate"
 130 LET a$= INKEY$ : IF a$="" THEN  GO
TO 130
```

```
 140 IF a$="d" THEN  GO SUB 500: GO TO 1
20
 150 IF a$="c" THEN  GO SUB 600: GO TO 1
20
 160 GO TO 130
 300 REM
 301 REM    Display Binary Number
 302 REM
 310 LET power=128: LET remain=number
 320 FOR x=0 TO 7
 330 LET result=1: LET remain=remain-pow
er
 340 IF remain<0 THEN  LET remain=remain
+power: LET result=0
 350 LET z=x*4+1: LET color=4-2*result:
FOR y=3 TO 9: PRINT  AT y,z; PAPER color
;"  ": NEXT y
 360 PRINT  AT 12,z;result
 370 LET power=power/2: NEXT x
 380 LET a$="00"+ STR$ number: LET a$=a$
( LEN a$-2 TO  LEN a$): PRINT  AT 17,22;
a$
 390 BEEP .5,number/4: RETURN
 400 REM
 401 REM     Display Screen
 402 REM
 410 BORDER 4: PAPER 6: INK 1: CLS
 420 LET power=256: FOR x=0 TO 7: LET po
wer=power/2: LET z=x*4+1: PRINT  AT 1,z;
"d";7-x; AT 2,z; INK 3;power: PLOT 32*x+
7,96: DRAW 0,55: PLOT 32*x+24,96: DRAW 0
,55: NEXT x
 430 PRINT  AT 0,0;"("; AT 0,31;")": PLO
T 6,174: DRAW 244,0: PRINT  AT 0,12; INK
 7; PAPER 1;"DATA BUS"
 440 PRINT  AT 13,0;"("; AT 13,31;")": P
LOT 6,64: DRAW 244,0: PRINT  AT 14,10; I
NK 7; PAPER 1;"BINARY NUMBER"
 450 PRINT  AT 17,5;"Decimal Number--"
 460 RETURN
 500 REM
 501 REM     Demonstration
 502 REM
 510 PRINT  AT 20,1;"Press "; INVERSE 1;
```

```
"M"; INVERSE 0;"enu or hold down "; INVE
RSE 1;"F"; INVERSE 0;"reeze"
 520 FOR a=1 TO 255: LET number=a: GO SU
B 300
 530 IF  INKEY$ ="m" THEN  RETURN
 540 IF  INKEY$  <> "" THEN  GO TO 540
 550 NEXT a: RETURN
 600 REM
 601 REM     Calculate
 602 REM
 610 PRINT  AT 20,1;"Enter a value from
zero to 255"
 620 INPUT "Decimal Number ? ";number
 630 LET number= INT number: IF number>2
55 OR number<0 THEN  GO TO 620
 640 GO SUB 300: RETURN
```

When Program 2.1 is entered and safely saved, enter 'RUN'. You will be offered a choice of either a demonstration or a calculation.

The calculation mode asks for a number in the range 0–255, and then displays it in two forms. At the top of the screen, there are eight lines representing the data bus of your computer. If these are red, the voltage is high; if green (or the same shade as the border if you are using a monochrome TV), it is low. Across the screen below the lines is a string of zeros and ones, showing your chosen number in its binary form.

For the demonstration mode, the computer will cycle through the numbers, and you can freeze the display by holding down any key other than the menu key, 'M'.

Handling data

So now we have seen how an eight-bit micro deals with data. You may quite rightly argue that your Spectrum can handle words, and both very large and very small numbers. However, these are handled by the built-in software program — numbers, for example, are stored and manipulated as five separate bytes of data. Even the machine language which drives the computer sometimes uses a serial form of coding: certain machine instructions come as two or more bytes, with the first giving instructions to the computer to expect further data and how to treat it. These operations will be discussed later. Parallel data transfer can sometimes extend outside the computer itself: for example, some printers are connected in this manner. When long distances are involved, or speed of communication is less important, it is usual to revert to a serial method. The coding of data into a suitable form may be handled by the computer's host program.

We have looked at the handling of data in digital form. A similar method is used to direct the flow of that data within the computer. There is another 'bus' inside your Spectrum, called an 'address bus', and this holds a binary number to indicate which particular area the computer wishes to deal with. Each number is unique and can therefore be decoded by using logic circuits to activate the relevant address. (Remember the lighting of the bulb in the five-bit parallel data transfer system I mentioned earlier.) In the next chapter you will see how the computer manipulates data by using both the address and data buses.

CHAPTER 3
The Microprocessor

In the not too distant past, computers filled whole rooms. The component parts were housed in separate packages, according to their function. If you had been shown round such an installation, your attention may have been directed towards 'memory' in the big, grey boxes over there, or to 'punched-tape reader' here. Your guide may well have indicated one of the anonymous grey cabinets and said: 'that's where the actual work takes place; it's the Central Processing Unit'.

Computers have now shrunk in size, but if you look inside you can still point to one integrated circuit and say: 'that's where the work is done'. These 'CPUs' or 'microprocessors', contain all the components that used to be found in that large grey cabinet, etched on to a slice of silicon of tiny proportions.

Your Sinclair Spectrum contains a Z80 eight-bit microprocessor, a design developed by Zilog in the late 1970s. It is one of the most successful, particularly in the field of small business microcomputers, so I will be using it as a basis for my descriptions. Other microprocessors may be more or less sophisticated, but the general principles are the same. The enclosing of all the processing functions into one circuit encourages us to look at the microprocessor as a kind of 'black box', and just concern ourselves with its inputs and outputs. It is at this point that I intend to start.

The Z80 — the exterior

A Z80 microprocessor has 40 terminals. Whilst some of these are for inputting, and others for outputting, voltages, some of them can perform both functions.

To deal with the simplest first, let us study the two terminals (or pins) of the integrated circuit which provide it with its power. For its internal operations and also for outputting to other devices, it needs a supply of five volts positive applied to the pin called the Vcc terminal (supply voltage), relative to the GRN (ground) terminal. When this supply is present, a small current passes through the device, the amount of which varies according to the function being performed by the Z80.

Next there are the eight data pins, D0 to D7, through which the CPU can pass bytes of data to and from the outside world. These are bi-directional. On some occasions, the eight pins are pulled high or low by the CPU in order to impose a byte of data on the data bus to be used elsewhere. At other times, the Z80 reads in data placed on the bus by other devices through its data terminals and, when this is happening, these terminals are floating.

This type of terminal is very useful and therefore very commonly found on computer chips — it alows you to connect many things to the same data bus flowing along the lines. Chips that are built with such terminals are known as 'tri-state' devices.

Controlling data

Now that we have a method of passing data in and out, we need to control its destination. For this purpose, microprocessors are equipped with address lines allowing them to signal which one of their surrounding devices they wish to communicate with. The Z80 has 16 such address lines. One of the more sophisticated features of this CPU is that it is designed to address two separate forms of peripheral device in a different manner. These are 'memory devices', which store data, and 'ports', which interface with the outside world.

Two terminals called IORQ (input/output request) and MREQ (memory request) indicate with which class of device the CPU wishes to deal. If MREQ is at 0 volts, then the CPU addresses memory: the address lines hold a 16-bit binary number, revealing which memory slot the CPU intends to communicate with. Eight bits of data give us 256 possible combinations: this is 2 multiplied by itself 8 times, which is 2 raised to the power of 8, and it can be written as 2^8. 2 raised to the power of 16 comes out as 65,536, and this is normally the maximum of memory slots (or locations) that the Z80 can address, as 16 is the number of address lines. I say normally because, with additional circuitry and some clever tricks, some microcomputers (although not the Spectrum) are capable of switching between memory devices. This allows for more storage space.

It is worth a short digression in order to explain a piece of computer jargon which can be confusing. Advertisements and computer literature frequently describe a computer as having a certain amount of memory, for instance, 16K bytes. As K is the SI unit for a thousand (ie the internationally accepted standard) it would be natural to assume that 16K means 16,000 bytes. However, 1K of memory is used to represent 1024 bytes, this being the amount of memory that can be addressed using 10 bits (ie 2^{10}). This small discrepancy makes 64K, which is the normal maximum capacity of a Z80 based microcomputer, actually mean 65,536 bytes.

Two further pins called RD and WR (standing for 'read' and 'write') tell the memory circuits whether the Z80 requires data from memory, or whether it is placing a byte on the data bus which it requires to be stored at the location specified by the address bus. These pins are also used when the CPU is addressing a port. We have already seen that the CPU can address memory when the MREQ is at 0 volts. When IORQ is low, the read and write pins serve the same function they did for memory, applied now to a port, the address of which is held on the eight low address lines. This means that the maximum number of ports that the Z80 can handle is the by now familar number, 256; and, as with memory, IORQ and the address bus can be decoded to activate the desired circuits. However, it is possible to acquire extra space in the port addressing, and this will be dealt with under the section devoted to the keyboard.

The essence of a computer system is beginning to emerge but, in order for the CPU to spring into life, there are two remaining pins to consider. The first is named CLK, which stands for 'clock': this is the heartbeat of the processor. In order to make the Z80 function this pin must be pulled alternately low then high by an external voltage source. This is done by a circuit called an 'oscillator', which generates a varying voltage in a form known as a square wave. The output of the circuit is high for a fixed period of time and then falls low for the same duration, performing this task continually.

If you are interesting in seeing why the output is named as it is, enter and run Program 3.1. This plots a graph of voltage against time, but the speed of this simulation is very slow compared to the oscillator in your machine. The sequence of the voltage going high, remaining there, going low, and then starting to go high again, is called a 'cycle', and we measure the speed of alternating voltages as the number of cycles they perform in one second. The scientist who has this very fundamental unit of measurement named after him is 'Hertz'. Hi-fi buffs will know that the frequency of the sounds that we humans can hear occurs in the range of about 30 hertz to more than 16 kilohertz, or sixteen thousand cycles per second. The operating speed of the oscillator in your Spectrum is much higher — 3.5 megahertz (mega stands for million).

Program 3.1: Square wave

```
 10 REM        Square wave graph
 11 REM        plotting program
 12 REM
100 REM        Set up
101 REM
110 PAPER 0: INK 6: BORDER 0: CLS
120 PRINT "VOLTS = ": FOR x=1 TO 6: PRI
```

```
NT  AT x*3,0;6-x: NEXT x
 130 INK 3: PLOT 7,151: DRAW 0,-120: DRA
W 247,0: INK 6
 140 PRINT  AT 1,17;"Square Wave"
 150 DIM a$(9)
 200 REM
 201 REM       Plotting Loop
 202 REM
 210 FOR x=0 TO 400
 211 REM
 220 REM       Calculate volts
 221 REM
 230 LET volts=2.5+10* SIN (x/40)
 240 IF volts>5 THEN  LET volts=5
 250 IF volts<0 THEN  LET volts=0
 300 REM
 301 REM       Plot values
 302 REM
 310 PLOT  INK 7;8+(24*x/40),32+volts*24
 320 REM
 321 REM       Print values
 322 REM
 330 PRINT  AT 0,7;volts;a$
 340 PRINT  AT 21,15;x/40;a$
 350 REM
 351 REM       End loop
 352 REM
 360 NEXT x
 370 STOP
```

Each time the microprocessor senses that the clock input is being pulled from low to high, it performs the next task, so the clock acts as a time-keeper for the CPU, regulating its actions and prompting it into doing the next function. With all the complex circuitry that a microprocessor contains, why can it not regulate its own speed, or, even simpler, run as fast as it can? The answer is that the external clock signal gives circuit designers control over the CPU, and allows them to build a computer in which all the associated circuitry can keep pace with the main microprocessor. It also gives them the facility to stop the clock, as it were, to freeze all action until the oscillator is turned on again.

The remaining pin that I must mention is the 'reset' pin. If you attach it to zero volts it behaves exactly as its name implies it should — it stops the CPU from continuing whatever it is involved with and makes it recommence from a predetermined point.

Let's go through what happens when your Spectrum, or any other Z80-based microcomputer system for that matter, is turned on. All the circuits begin to receive the voltage they require in order to function. The oscillator circuit starts to apply the square wave clock signal to the CPU. However, a very simple electronic circuit ensures that, for the first few moments after the switch-on, the reset pin has zero volts applied to it. This causes the processor to start working from the correct place.

It is possible to witness what would happen if this was not done, by turning your machine off and on very quickly. If insufficient time is allowed for the reset signal to be generated, the computer may 'hang up' — it will display a peculiar picture on the screen and fail to respond to the keyboard. When the reset signal is generated correctly, however, it holds the reset pin at zero volts long enough for the CPU to sort itself out. The reset circuitry then allows the appropriate pin to rise to five volts, and the Z80 can start work.

After a reset, the Z80 always does the same thing — it puts the number zero on the address bus, so that all 16 address lines will be set to zero volts by the address pins, MREQ and RD. Now the Z80 has generated sufficient information to tell its associated circuitry that it requires the data stored in memory location zero to be placed on the data bus, and the memory circuits begin to perform this task. The processor waits for the next clock pulse (the low-to-high transition of the oscillator circuit). Then it assumes that the data on the data bus is that provided by the memory, and so reads it through its data pins. What the CPU has just fetched from memory location zero is its first machine code instruction and, still using the clock pulses as a timing reference, the processor goes about the business of performing whatever the instruction tells it to do.

The processes by which the Z80 can store data in memory and exchange information with circuits which are activated by the IORQ pin, should be assumed to be variations of the above description of how it can fetch data from its memory. The time has come to look at what is contained within the CPU itself. This is not as daunting as it may seem and, if you are interested in the nature of machine code, the internal structure of the Z80 is essential knowledge.

CHAPTER 4
Inside the Box

The inside of the Z80 microprocessor can be divided up into a number of different areas. The workings of some of these need not concern the person who wishes to write machine code programs, but an overall view of what goes on can be of great benefit if you want to discover what actually happens when the program is running.

The registers

The processor contains a number of registers. A register consists of eight flip-flop type circuits, each capable of storing one bit of information, connected in such a way so that together they can hold one byte of data — they are in effect the CPU's personal memory cells. The registers are all connected to an internal data bus. (NB. This is not the same data bus as the one with which the Z80 communicates with the outside world of memory and ports.) Whatever appears on these lines is not always transmitted to the data pins, although they are used to carry data to and from the pins when required.

Also connected to the internal data bus is circuitry called the 'arithmetic and logic unit', or ALU. It is in this that all the processing of bytes of data is carried out. Here the CPU can add or subtract two binary numbers, and perform logical procedures such as ANDing two bytes together. The whole operation is governed by complex circuitry called the 'control unit' — this deciphers the machine code instructions fetched from memory, and reacts accordingly.

Figure 4.1 merits considerable study — it represents the internal structure of the Z80 processor. The control unit can manipulate all the registers, as well as send the required signals to the external circuits and the bus control circuits. The 'instruction' register can only be used by the control unit; its function is to store the last instruction that was fetched from the external memory. All the other registers can be used by the programmer, but some are more versatile than others.

The PC register, the 'program counter', is used to store the *address* of the next instruction to be interpreted. When the CPU is reset, this register is loaded with the value zero, which is the location of the first instruction. When this has been fetched, the control unit automatically

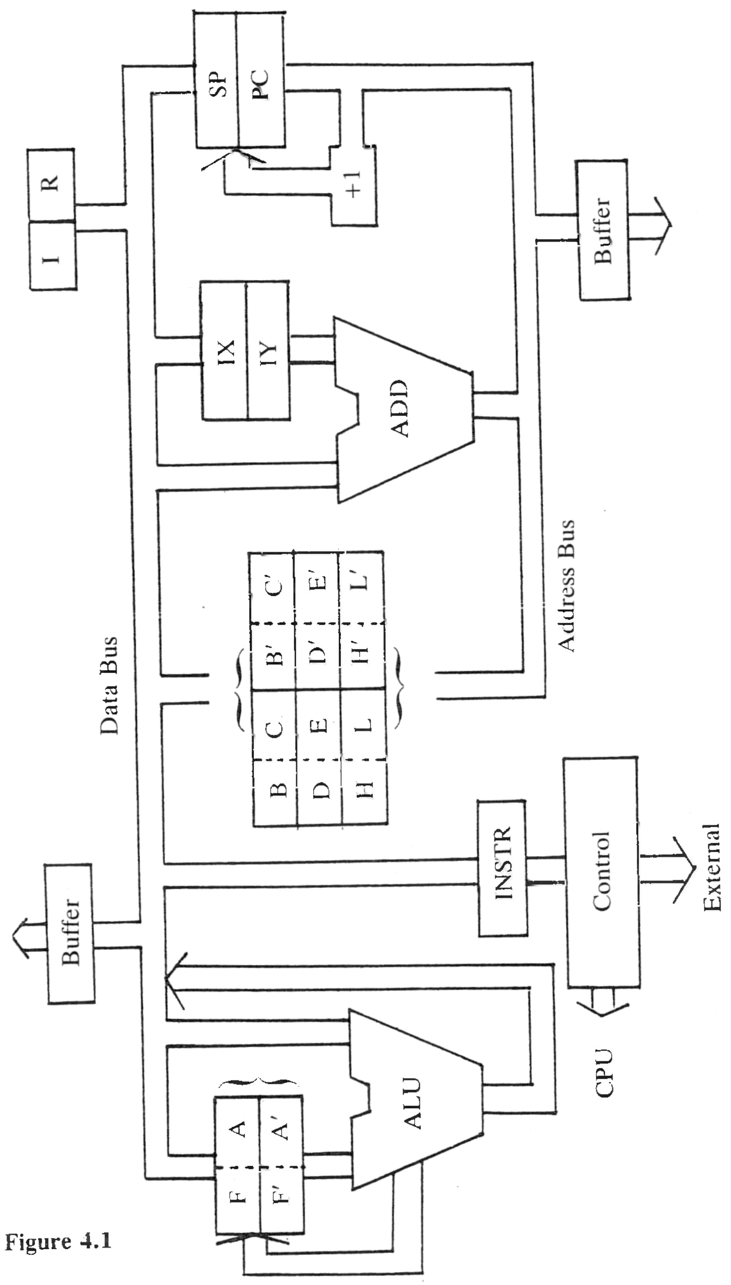


Figure 4.1

adds one to the value in the PC, thus storing the address from which it will receive subsequent data. Unless the programmer uses certain 'jump' instructions to reload the PC with a different value, machine code data is always read sequentially. The PC is a 16-bit register, so it can supply any one of 64K addresses to the address bus.

The main block of user registers comes in two groups, both of which use identical initials as names, but with one set distinguished by a ' suffix to indicate that it is the alternative group. Only one of these sets is active at any one time — special machine code instructions allow the user to determine which group is connected to the internal data bus in order to receive further instructions. To all intents and purposes, the alternative set can be thought of as useful storage space, particularly if you interrupt work on one task to carry out another (in which case the values held in the registers can be saved by switching over to the other set for use until you are ready to return to the first task).

The 'accumulator', or A register, is the most versatile of these user registers, because it can have all the arithmetic and logical functions applied to it. The value held in any other user register or stored in external memory can be added, subtracted, ANDed, ORed, or XORed with the A register, which will then contain the result.

Certain results of the ALU's calculations can be of particular interest, eg when the final product is zero. For its own purposes, as well as for those of the programmer, it notes electronically some of these occurrences and stores them as single bits of data in the F register, sometimes called by its full name, 'flags'. This conjures up a quaint analogy — imagine a small flag hoisted up a pole marked 'Z' every time a calculation results in zero. Amongst other things, the F register also records if the operation has resulted in a 'carry', ie if the answer has exceeded an eight-bit value.

The remaining six eight-bit general purpose registers, B, C, D, E, H and L, can be used as the temporary storage space of values which the CPU can therefore get at quickly. But they can also function as 16-bit registers by pairing up into BC, DE and HL — a facility which is very useful when you are manipulating address values. As pairs, they can have some limited arithmetic operations performed on them, with the result going to the HL register. Another role for the register pairs is as a memory pointer: this means that you can, for example, load the A register with the value held at a memory location, the address of which is held in the HL register.

There exist two special 16-bit registers of which the purpose is to hold important memory addresses — important, that is, to the programmer. He is able to load the IX or IY registers with a convenient address, and then have easy and quick access to memory addresses in the range 128 locations below, up to 127 above, those held in either of these 'index' registers. Two of the less frequently used registers are named I and R. The I, or 'interrupt vector', register can be used to change the behaviour of the CPU when it receives an interrupt, a subject I will deal with later. The R, or 'refresh', register is often used if the Z80 is worked in conjunction with 'dynamic memories'. Again, I will expand on this later.

The final register for us to consider is the SP, or 'stack pointer', register. This is another form of memory pointer and allows the use of some sophisticated programming techniques. Those of you familiar with BASIC programming will know of the 'subroutine', whereby the program can be made to jump to a particular line: it remembers where it came from and returns there when it encounters a RETURN instruction. The same principle can be used in Z80 machine code, with the return address stored at the memory location indicated by the stack pointer. This register has other uses which will feature in the machine code chapter.

You should now understand how the CPU exchanges data with the outside world, and have an idea of its internal structure. Let's go slowly through the procedure for executing a very simple program, which we wil assume is stored in external memory starting at address zero. Refer to **Table 4.1** to see the example values.

Table 4.1

Address	Contents
0	62
1	8
2	33
3	0
4	88
5	61
6	202
7	15
8	0
9	54
10	0
11	35
12	205
13	5
14	0
15	116

Executing a program

When the reset pin ceases to be held low by the reset circuit, the Z80, during the first two clock cycles, fetches a byte of data from the memory address held in its program counter. This it passes along its internal data bus to the instruction register where it is stored for use by the control unit. This data is the first machine code instruction of the program (or 'op' code, short for operation code) — the control unit decodes it and reacts accordingly.

In our example program, the first op code is 62 decimal, which tells the control unit to fetch the next piece of data stored in memory, and place it in the A register. Two more machine cycles occur whilst this instruction is 'understood'. Meanwhile, the control unit increases the value in the program counter by one, so that now it holds the value one. Incidentally, the control unit can perform concurrently other tasks related to servicing any dynamic memory circuits which may be attached, but this need not concern us.

Back in our example program the CPU, aware that what is stored at address one is not a further instruction but data, performs another fetching operation from memory and stores it in the A register, as instructed by the previous op code. It also increments the program counter again. Whatever data was stored at address one, in this case 8 decimal, is also stored in the A register. The CPU has completed the task set by the first instruction, so it fetches the next from memory, using the address, currently 2, stored in the PC. The next op code in the program is dutifully transferred to the instruction register and acted upon. It is 33 decimal, which tells the CPU to put the next two bytes of data into the L and H registers respectively. Now the L register contains whatever was stored at location 3 (0); and H stores the data from location 4 (88 decimal). Remember that one is added to the PC each time a byte is fetched.

We have now reached the instruction in location 5, which is collected in the same manner. This is 61 decimal, which represents the simple task of subtracting one from the contents of the A register: when this is performed, A holds the value 7 decimal. On to address 6, which contains 202 decimal. This is a conditional op code and could be written as, 'fetch the next two bytes of data from this program and IF the result of the last arithmetic or logic operation was zero, load the data fetched into the program counter'. The CPU is making its first 'decision', based on the contents of the flags register. We made the CPU perform arithmetic only one program step earlier, when it subtracted 1 from the contents of the A register. However, the result was 7 rather than zero. So, although the next two bytes are fetched from addresses 7 and 8, nothing is done with them as the Z flag is not set. We will see shortly what happens if it is.

The PC now contains 9, so the CPU loads the contents of this address, which is 54 decimal, into the instruction register. This in effect tells it to 'fetch the next byte from the program as data, and place it in the memory address represented by the contents of the HL register pair'. So the value zero is obtained from address 10, and the control unit goes about the task of storing it in memory.

The first step is to place the contents of the L register on the eight low address lines, and the contents of the H register on the eight high address lines. This means that the address bus holds the value 22528 decimal, which is 256 times the value in H (88), plus the contents of L, in this case zero. Then the control unit pulls the MREQ line low, activating the external memory devices, and it places on the data bus the value it has just fetched from the program memory area, which is zero. It waits a clock cycle for everything to respond and then pulls the WR, or write, line low. The external memory takes this as a signal that it should store the data on the data bus, at the address on the address bus. What is the result? The data which was held in memory location 22528, has now been over-written by the data zero.

With the PC now set at 11, the CPU has reached the instruction 35 decimal, which it fetches and reacts to by adding one to the value of the HL register. The data in address 12 is the 'unconditional' version of one we have encountered before, at address 6. This time the CPU is told to load the PC with the next two bytes. So, as locations 13 and 14 contain 5 and 0 respectively, these are fetched, and then the PC is loaded with the value 5. (The low part of the PC is loaded with the first byte fetched, and the high part with the second.)

Now where does the next instruction come from? The previous address in the PC has been over-written, so the next instruction comes, not from location 15, but from 5. The CPU has performed a jump instruction: it loads the op code from address five, which is 61, an instruction that it has performed before. The program has created a loop — it will continue to perform the op codes at locations 5 through to 12, but it will not do this indefinitely, due to the conditional instruction at address 6. Each time the loop is carried out, the value of the A register is reduced by one. After the eighth time that the op code 61 is performed, the A register will have fallen to zero, so that when the CPU checks the zero bit of the flags register, it will find that the previous operation has indeed resulted in zero.

This involves proceeding to the second part of the operation dictated by the op code at address 6 — if you refer back you wil see that, in the instance of the result being zero, the PC is loaded with 15. After executing the loop eight times, the program jumps to the instruction at 15. This happens to be the final one, and it tells the CPU that the program has been completed.

Use Table 4.1 to go through each step of the above program carefully until you are clear as to what is happening inside the processor. At this stage it is not important to understand in detail the program itself, but rather to visualise what steps are taken by the internal structure of the Z80 processor, in order to follow through the instructions it fetches from the area of external memory containing the program.

Well, does the program work? And what does it actually do? To answer these questions, why don't you try running it on your Spectrum? First of all, however, it must be loaded into memory. We cannot load data into the first 16K of memory as, on the Spectrum, this is filled with the machine's own program, so we will have to use an area higher up in the memory. Consequently, as the first instruction will not be at location zero, we cannot use the reset pin to direct the program counter to it. However, there is a way round this.

The Spectrum has the facility of running machine code from BASIC via the function USR. This is included in Program 4.1 in the form 'RANDOMIZE USR 28672', and it works in the following manner. The address the BASIC program has reached in its execution is stored at the address indicated by the stack pointer, and the program counter is loaded with the value 28672, so that the CPU starts running the program stored at this address. When the CPU finally encounters the op code 201 decimal, it reloads the original value of the PC from the area of memory pointed to by the SP. This ensures that the CPU returns to the BASIC program it had been running. BASIC now uses the value that is contained in the BC register of the CPU, and treats it as the result of the function USR.

The practical consequence of this is that if we state RAND USR and then a valid memory address in any BASIC program we write, the Spectrum goes to that address and runs any machine code program found there, until it reaches a RET machine code instruction (201 decimal). Then it seeds the random number generator with the value of the BC register. This is just a convenient way of calling up a routine: if we use PRINT USR instead, then the result from BC would be printed on the screen.

Now enter and save Program 4.1 When you run the program, it first of all POKEs the machine code held in the data statements into memory, starting at location 28672. Then it prompts you to press a key, and when you do so it jumps to the program and executes it. A thick black line should appear in the top lefthand corner of the screen. Note the speed with which this occurs: what has happened is that the machine code has poked zero values in seven memory locations, 22528 to 22535. I chose these particularly because they make up the first part of what is called the 'attributes file'. This controls what colours appear on the screen, and where, and is explained in the section of the book devoted to screen display. You may wish to try changing the value of the A register (the second number in the data statement), the value loaded into the attribute file (eleventh in the data statement), or even the memory pointer in HL itself (but you will need to delete line 60 before other values will work). If you make HL lower than 16348 you will not see the result. Any higher than 23296 and not only will nothing show, but you may cause a 'crash' by interfering with the running of the computer. This will not cause any damage, but you may have to switch the machine off and on again to regain control. Incidentally, this machine code program is not only very simple, it is also quite crude. But it does prove that we can bypass the Spectrum's operating system and exploit the power of machine code programming.

Table 4.2: Machine Code for Program 4.1

Address	Hex Code	Assembler
7000	3E08	LD A,08h
7002	210058	LD HL,5800h
7005	3D	DEC A
7006	CA0F70	JP Z,700Fh
7009	3600	LD (HL),00h
700B	23	INC HL
700C	C30570	JP 7005h
700F	C9	RET

Program 4.1: Machine Code Demo

```
  10 REM      Machine code demo
  11 REM
  20 REM         Set up
  21 REM
  30 PAPER 7: INK 0: BORDER 7: CLS
  40 REM
  41 REM         Check data
  42 REM
  50 RESTORE : LET sum=0: FOR x=0 TO 15:
 READ a: LET sum=sum+a: NEXT x
  60 IF sum <> 1183 THEN  PRINT "The dat
a statements are wrong."''"Please check
that you have"'"entered them correctly":
 STOP
  70 REM
  71 REM    Poke code into memory
  72 REM
  80 RESTORE : FOR x=28672 TO 28687: REA
```

```
D byte: POKE x,byte: NEXT x
  90 REM
  91 REM         Wait for key
  92 REM
 100 PRINT  AT 16,10;"PRESS A KEY"
 110 IF  INKEY$ ="" THEN  GO TO 110
 120 REM
 121 REM         Jump to routine
 122 REM
 130 RANDOMIZE  USR 28672
 140 REM
 141 REM         Returned
 142 REM
 150 PRINT  AT 16,8;"Program complete"
 160 STOP
 500 REM
 501 REM         Machine code
 502 REM
 510 DATA 62,8,33,0,88,61,202,15
 520 DATA 112,54,0,35,195,5,112,201
```

CHAPTER 5
RAM and ROM

In previous chapters, I have glibly mentioned 'external memory' without any further illumination. However, now that we have covered the main working processes of the Z80, it is time for a description. Memory is used by your Spectrum to store the data it needs in order to function, whether that data is the machine code it uses when it is switched on, the simple program from this book, or your latest copy of Space Invaders.

Types of memory

There are essentially two distinct types of memory used in microcomputers. 'Read only memory' (ROM) does not allow the microprocessor to alter its data — as its name implies, it can only be *read from*. The other is called 'random access memory' (RAM), which is a slightly confusing title because its main feature is the facility for having its contents altered under the control of the computer. Of course, the situation is not as simple as this sounds: within both RAM and ROM there are different types of memory devices. Let us start by looking at the type of ROM that your Spectrum uses to store the built-in machine code program which makes it function.

This ROM has 14 address bus pins, labelled A0 to A13. When the 'chip select' (CS) pin of the memory is pulled low, the contents of an eight-bit register inside the chip, corresponding to the address currently present on the address pins, are placed on the data pins of the memory device (and consequently on the data bus of the computer). As each different address is capable of accessing a separate eight-bit storage area, with 14-bit binary addresses the ROM is capable of storing 16484 (16K) different bytes of data. This is actually present in the form of simple connections made at the time of manufacture.

Imagine that the address is decoded by logic circuits in such a manner so as to force the output of one, and only one, logic gate high. If that address were designed to store the number 255, or 11111111 binary, then that gate's output would be connected to all the eight data pins of the ROM. If the data stored was supposed to be 129, which is 10000001 in binary, then the high voltage on the output of the gate would only be connected to data pins 0 and 7. The data lines are normally low, but the logic gates can pull them high so that the pins will reflect the data stored at an address pointed to by the information on the address bus. When a logic gate is not selected by its appropriate address, its ouput is floating, and therefore it has no effect on the data bus. When the CS pin is not activated by a logic 0 voltage level, then the whole output of the chip is floating.

You could calculate that, if every address held the data 255, then over 130,000 connections would need to be made. This is achieved by the manufacturers using a photographic process to etch the connections on to the silicon material of the chip itself. The template used in this process is very expensive, but once set up the cost of producing each individual chip is low, so these Mask Etched Read Only Memories are economical only for large production runs.

It is worth mentioning two other types of ROM that you may encounter. The Programmable Read Only Memory (PROM) is a device that comes of the production line completely blank. It is programmed by a machine generating high voltages which 'blow' the linking connectors in much the same way a fuse might behave if too great a voltage were applied to it. An improved version, the Erasable Programmable Read Only Memory (EPROM), stores data as electric charges. Not only can it be programmed, but also, if necessary, the data can be erased by the application of ultra-violet light through a small window on its upper surface.

Both of the above memory device types are used mainly for small production runs and prototypes, and they are often 'pin compatible' with normal ROMs. This simply means that they will fit the same sockets and have their pins in the same places. The early prototype Spectrum computers were fitted with EPROMs, so that they could be fully tested before the program contained in the EPROMs was committed to an expensive tooling-up procedure to produce the ROM which replaced them.

Why we have different types of memory

The most important aspect to notice about all ROMs is that they will retain the data they contain at all times, whether they are powered up or not, because, although they need a supply voltage to operate the decoding circuits they contain, the data is stored in a 'non-volatile' form. This is an essential facility for any computer that is to be turned on and off, so that when it is first switched on, there are always some instructions ready and waiting.

Memory which can contain only data that has been preprogrammed is

of no use when we want to utilise external memory to store data that the computer can change. Even the machine's own operating system, which controls the keyboard and screen and interprets the programs that are typed into it, needs more registers than the Z80 can supply — it must remember, for instance, where it last printed something on the screen. So computers are supplied with random access memory, with which they can write data to locations for later retrieval. The name refers to the fact that locations can be accessed for either reading or writing in any order. (Note that the term random access could be applied to the types of ROM I have mentioned, but it has come to be used only for the forms of memory that can be written to.)

The simplest type of RAM can be thought of as a ROM layout with address lines and decoding circuits but, instead of the links, each piece of information has a flip-flop arrangement of which the output represents each 'bit' of data. This output is routed to the appropriate data line when the correct address is fed on to the address bus. Also, the data bus is bi-directional so, with the aid of two pins to tell the chip whether to output or input information, the CPU can send data along with an address to the memory, and it will set the values of the appropriate flip-flops to those of the data sent.

This kind of memory is called 'static RAM', because as long as it is receiving a supply voltage it will maintain the pattern of bits that its internal circuitry is storing. However, this type of memory is expensive and difficult to make compact, so it will come as no surprise to you to learn that the Spectrum is equipped with a different breed, called dynamic memory. This stores data in the form of an electric charge which, unfortunately, begins to leak away after a short period of time, so the memory needs the opportunity to refresh the charges by receiving a request to read each bit before their values are lost.

As I hinted earlier, the Z80 has been equipped to provide refresh signals during the two clock cycles that occur after a memory fetch operation has been performed. The R register is used to provide sequential addresses for the memory circuits to use in conjunction with the refresh pin. From a user's point of view, there is no difference in the performance of dynamic or static random access memories.

My main task in this chapter has been to differentiate between RAM and ROM. The more sceptical amongst you may wish to demonstrate the differences by following this procedure. Find the contents of a ROM address, say, location zero, by entering PRINT PEEK 0. Now try to alter the value stored there to, for instance, 1, by entering POKE 0,1. Get the computer to PRINT PEEK 0 again, and you will discover that your efforts have been in vain. Repeat the above operation for the first RAM address, which is 16384: ie, PRINT PEEK 16384; POKE 16384,1;

PRINT PEEK,16384. Proof! Incidentally, if you have a 16K Spectrum, all addresses above 32678 will not react, as there are no memory locations in that area.

CHAPTER 6
Language for Machines

It is no use sitting down in front of your computer and typing 'Play me a cheerful tune': it simply won't know what you're talking about. You can write a program that will play any tune you like, but first of all you must learn how to make the computer understand you. In order to bridge the gap between machine code and human language, there have been developed over the years many intermediate languages that can be entered into a computer and understood by both parties.

BASIC, the language that your Spectrum understands, is short for Beginners' All-purpose Symbolic Instruction Code, and it is easier to grasp than any other computer language. Although 'professionals' are somewhat disparaging about it, it is flexible and universally accepted as the most popular language. You can if you wish buy other languages that are more elegant to use and faster to run on your machine but, unless you're a glutton for punishment, I suggest you stick with BASIC for now.

I trust that you are familiar with BASIC because it is not within the scope of this book to explore the finer points of writing programs. If you have yet to write your own BASIC, you will find that the Spectrum's manual explains this well. In addition, there are shelves full of books exploring programming techniques in your local computer shop: before we proceed, you should teach yourself BASIC.

Interpreting BASIC

So how does BASIC work? The Z80 only understands what the bytes of data fed into it from memory tell it to do, but not statements such as PRINT AT 2,5;"HELLO". Imagine that you have written two programs to beep out on the speaker cheerful and marching music: organise them as subroutines and you can type, for instance, GOSUB 100 for 'Happy Birthday'. If, at the top of the program, you write something such as

```
INPUT A$: IF A$="Play me a cheerful tune" THEN GOSUB 100:
STOP
```

then when you run the program and type in 'Play me a cheerful tune' your computer will appear to understand you. By adding further lines, perhaps 'Play me a tune to march to', you may even fool the lay observer into thinking that your machine has musical taste! However, your micro is interpreting your instructions not through any thoughtful process, but rather by simply comparing them with a list it holds, and responding if your demand tallies with one it recognises. The machine code program in your Spectrum contains a 'BASIC interpreter' that works along similar lines. When running, it uses the values of the 'keywords' it encounters to find where the machine needed to perform that task is stored in ROM, then it GOSUBs that routine. (In machine code the word CALL is used instead of GOSUB.) Before moving on to the rather daunting subject of machine code itself, it may help if I explain what goes on when we switch on a Spectrum and type in

```
PRINT AT 2,5; "Hello"
```

and then press ENTER.

Switch on. The Z80 performs a reset, then it executes the op codes stored, starting from location zero. These are very involved: setting up the areas of memory, organizing the 'system variables', and generally putting the house in order. When the Z80 has finished this 'system initialisation' routine, it jumps to the main operating system loop which waits patiently for someone to press a key. When you do, it knows this to be either a line number or a keyword — on pressing 'P' the operating system looks up how to spell PRINT and places the letters on the screen. It also stores the relevant 'token' value in an area of memory set aside called the 'edit buffer': each BASIC word has its own token value and the one for PRINT is 245 decimal. The operating system then follows the same procedure for the AT.

Now we enter '2,5;' — even more work is involved here, as it stores the relevant numbers as six bytes of data. (This is actually not necessary for small numbers such as 2 and 5, but it is required for larger ones.) The punctuation and string of letters between the quotation marks go into the buffer in the form of their ASCII codes (you can look these up in your manual).

When you finally press ENTER, the operating system places an 'ENTER' code at the end of the buffer, then it checks the 'syntax' of what has been entered. If nothing is wrong (ie there is no illicit combination of codes in the line) it moves on to the next stage; otherwise it queries the line.

All being well it checks if the data in the buffer starts with a line number. If it did, the operating system would transfer the string of data to an area of memory set aside to store BASIC programs for running

later. As the line you typed in is a direct command, the operating system passes control to the BASIC interpreter which looks up the first value — 245. It recognises this as a command to pass control to the output routine at location 10h — this routine handles the ensuing data by putting it on the screen, dealing with the 'AT 2,5;' tokens by moving the printing coordinates accordingly. When the ENTER token is reached, the output routine returns control to the interpreter. This, finding the task complete, prints 'OK 0.1', and causes the Z80 to jump back to the main operating loop. This example is very simple, but it demonstrates the importance of the 'system software' — without the operating system (OS) and BASIC language interpreter, the micro would be useless.

Machine code

Now for the enigmatic machine code. Some readers may already have become familiar with Z80 language, and they will acknowledge that there is great satisfaction in writing a working program. The drawbacks are the number of instructions needed to do anything significant, and the apparently abstract nature of machine code. However, there are mnemonics to help us remember the function of each code. If you refer to the listing in Chapter 4, you will see that 62 decimal means 'load A with the next number', which becomes LD A,N. These mnemonics are often called by the overall title 'assembly language', because there are programs available which will translate the names into the correct codes, and thereby 'assemble' a machine code program automatically.

In order to assist aspiring machine code users, the next section is devoted to expanding the assembler mnemonics. Don't attempt to digest them all in one go. Concentrate on the common ones such as LD, JR, ADD, SUB and CALL to begin with — a study of some of the obscure codes will be easier when you encounter them in the context of a program.

Machine code mnemonics glossary

ADC	This instruction can apply to either the A or the HL registers. Applied to A it means 'add the contents of the specified single byte, and the contents of the carry flag, to A, and leave the result in A'. Bytes held in single byte GP (general purpose) registers, memory pointed to by HL or the index registers, or stored as immediate data in the program, can be added. Applied to HL, the contents of the register pairs (BC, DE, HL) or of the stack pointer, are added to HL, along with an extra 1 if the carry is set, with the result remaining in HL. For example, ADC A,D adds D to A together with the carry bit.

ADD	This is the same as ADC without including the carry bit. In addition to A and HL, it can also be used on the index registers.
AND	This is a logic operation that can be performed on the A register. It takes a byte from memory (pointed to by HL or the index registers), program memory, or a single GP register, and executes an AND logic test on each bit of A and the corresponding bit of the specified byte, leaving the result in A. For example, *AND B* would AND the contents of B with A. If A contained 01010101 and B 00001111 binary, the result in A would be 00000101.
BIT	By using this instruction you can test any bit from the GP registers, and from memory pointed to by either HL or the index registers, and set the zero flag accordingly. For instance, BIT 3,(HL) would test bit 3 of the byte held at address HL, and if it were zero the Z flag would be set; if one, then Z would be reset. Bits are numbered from 0 to 7. This code is at least two bytes long, the first byte always being EDh.
CALL	The equivalent of GOSUB in BASIC, CALL pushes the contents of the PC on to the stack, and then loads the PC with the next two bytes in program memory, causing the Z80 to execute from that address. See RET to discover how to get back! CALL can be conditional — for example, CALL Z only actually occurs if the zero flag is set.
CCF	This means 'complement carry flag'. The value of the carry flag is inverted, zero becomes one and vice versa.
CP	This stands for 'compare'. Bytes from pointed-to memory, the next program location, or the GP registers, are subtracted from the value in the accumulator, and the flags set accordingly, but the original value of A is restored. If A held 20h, then CP 20 would set the zero flag, but A would remain at 20.
CPD	This is a complex instruction. It stands for 'compare and decrement', which is useful for searching a block of memory for a byte to tally with that in the A register. CPD compares the byte pointed to by HL with A, and sets the zero and sign flags accordingly (sign = 1 if bit 7 of the result is high). Then the instruction decrements HL and BC, which act as a counter, and if it reaches zero the P/V (in this case 'overflow') flag is set.

CPDR	'Compare and decrement with repeat.' This will continue to perform CPD until either BC has reached zero or the CP operation has set the Z flag.
CPI	'Compare and increment'. This is the same as CPD except that HL has 1 added to it rather than subtracted.
CPIR	'Compare and increment with repeat'. As CPDR, except that it increments HL. These block-searching instructions are very useful for finding data of a particular value. On completion, HL will either hold the address of the first match, or the end of the block if none was found.
CPL	This stands for 'complement' and applies to the A register only. All the bits are inverted, thus 00110101 would become 11001010.
DAA	This instruction is probably only of use if you are using the 'binary coded decimal' method of storing numbers. An eight-bit register is split into two nibbles of four bits; each nibble should only be in the range of 0 to 9. 'Decimal accumulator adjust' will automatically readjust the values in the A register after an addition or a subtraction in compliance with the rules of binary coded decimal.
DEC	'Decrement' subtracts 1 from the value it is applied to. You may DEC any eight or 16-bit register, or a byte in memory pointed to by HL or the index registers. One important point, which if not known can lead to a frustrating end to first programming attempts, is that decrementing the 16-bit registers (HL, DE, BC) will not affect the flags register.
DI	'Disable interrupt' this causes the CPU to ignore any 'maskable interrupt' it may receive at its INT pin. Interrupts are covered elsewhere in this book.
DJNZ	This useful code means 'decrement and jump if not zero'. It enables us to use the B register as a counter. It reduces B by 1, and IF the result does not equal zero, performs a 'relative jump' (see JR), dictated by the next byte from program memory. If zero has been reached, the operation proceeds to the next instruction.
EI	'Enable interrupt' — this cancels the DI instruction.

EX This enables you to 'exchange' two indicated registers. You may EX DE, HL, which swops their contents; you may EX the two bytes pointed to by the SP with those in HL or the index registers; or, EX AF, AF′ will switch control to the alternative AF pair.

EXX This command brings the alternative registers BC′, DE′ and HL′ into play, after which any operations are directed towards them. In order to revert to the original set, simply EXX again.

HALT This command does precisely what is indicated: the CPU will continue to perform NOP from the time it carries out a HALT until it receives an interrupt signal.

IM0 Refer to the keyboard chapter for a full explanation of interrupts. IM0 sets the Z80 to an interrupt mode not used by the Spectrum, in which the CPU expects an instruction (probably an RST) to be placed on the data bus by the device that has caused interrupt.

IM1 The correct mode for the Spectrum, IMI always causes an RST 38h instruction to be executed when an interrupt is received.

IM2 This is an interrupt mode that reacts to an interrupt by calling a subroutine of which the address has previously been stored in memory. The location looked up in order to find this address is determined as follows. The high order byte is taken to be the contents of the I register, and the low byte is taken from the data bus.

All the above interrupt mode instructions change the method of dealing with a maskable interrupt. The non-maskable interrupt always performs an RST 66h type of operation and, as its name indicates, it is unaffected by the DI instruction.

IN This reads a byte from external circuits that are 'port addressed'. It does so by placing the port address on the eight low address lines, making IORQ and RD active, and then placing the byte collected on the data bus in the specified register. IN takes two forms: IN A,(port), which has the port number contained in the next byte of program memory with A holding the result; and IN reg (C), for which the byte in C is used as the port address, and reg can be any GP register.

INC 'Increment' adds one to the value held in a specified register — the same rules apply as for DEC.

IND
INDR
INI
INIR These are a group of input instructions that serve the same purpose for IN as the CPD group does for comparing, and the LDD group for loading. IND transfers the contents of the port specified by the C register to the memory location whose address is pointed to by the HL register pair. HL is then decremented, and so too is the B register. If the latter reaches zero then the Z flag is set. INDR is the repeating version. It continues to transfer the data at port (C) to the area of memory through which the HL pointer is being decremented, until B reaches zero. INI and INIR function like IND and INDR respectively, except that HL is incremented rather than decremented.

JP 'Jump' allows the program counter to be loaded with a new value, which results in the operation of the program moving elsewhere. There are three alternatives: you may jump to the address indicated in the next two program bytes; perform a conditional JP, which will only occur if a flag test holds true (ie JP NC, 6000h will load the PC with 6000h, but only if the carry flag is not set); or do an indexed jump, which loads the PC with the value held by HL, IX or IY.

JR 'Jump relative' allows you to modify the contents of the PC by adding or subtracting a *seven*-bit value to the low byte of the PC. The required value is stored in the next byte of program memory. If the byte contains *less* than 127, it is added to the value of the PC, but only after the automatic increment following the fetching procedure. Therefore, JR 0 would have no effect, whilst JR 1 would make the CPU omit a byte before fetching its next op code.

If bit 7 of the 'displacement byte' is 1 (in other words, if the number is greater than 127) this value is treated by a convention known as the 'two's complement', which allows us to generate negative numbers. For example, FEh (254 dec) as a two's complement is −2. If you complement (see CPL) the displacement byte and add one, you will arrive at a value which is to be subtracted from the PC. Thus JR FEh (254 dec) forces the CPU to jump back two locations where it will encounter the same instruction, thereby creating an endless loop.

You can also use JR as a conditional instruction, but it can only test a limited number of flags, JR Z, JR NZ, JR C and JR NC. However, JR is often preferable to JP because, as well as being one byte shorter in length, its use absolves the programmer from writing code that must reside in a particular area of memory. JP needs to be accompanied by the actual address, whereas JR only modifies the current value in the PC.

LD This is the most common op .code of all, 'load'. First impressions may suggest that you can load anything with everything, but there are some limitations! In its simplest form LD A, B will transfer the value stored in B and place it in A, destroying the byte previously held in A. Let's systematically go through the load codes that are available.

You may load any eight-bit GP register with 'immediate' data: that is, data stored in the next location in program memory.

All the 16-bit registers (except PC) can be loaded with immediate data stored as the next *two* bytes of program. See, later, the note about how two-byte values are handled by the Z80.

External memory, of which the address is held in the HL or index registers (IX and IY plus a displacement, to be explained later), can also be loaded with immediate data, ie LD (HL), FFh.

It is possible to load the contents of any GP eight-bit register with the contents of any other.

The two special eight-bit registers, I and R, can be loaded with the value in A, and A can be loaded with their contents.

You can only transfer 16-bit values from the double registers on to SP (however, refer to EX). To transfer the contents of, say, BC to DE you can, of course, use LD D, B then LD E, C.

Data pointed to by the addresses held in HL or the index registers can be loaded into any eight-bit GP register.

The contents of any eight-bit GP register can be loaded into external memory if the address of the location is held in HL or the index registers.

The A register can also use BC and DE as memory pointers: so LD A, (BC) will load the contents of the address held in BC into the accumulator, and LD (DE), A will load the address held by DE with the value of A.

The extra flexibility of A is shown by the instructions LD A, (ADDR) and LD (ADDR), A. Here ADDR is a two-byte value stored as immediate data in the next two program locations. This is fetched and used as a memory pointer, so LD A, (0000h) will load A with the bytes of data stored at location zero.

Finally, this technique of using immediate data as a memory pointer can be applied to the 16-bit registers. For example, LD (ADDR), BC will store the contents of BC at two locations indicated by ADDR and ADDR+1.

LD is fairly extensive, but not infinite! For example, it is not possible to do LD (ADDR), (another ADDR). This would require something similar to: LD HL, (another ADDR), then LD (ADDR), HL. Also note that no load instruction alters the flags.

LDD This is a complex instruction similar to CPD. It causes the CPU to transfer the data held at the address stored in HL to the address stored in DE. It then decrements the BC, DE and HL registers. If BC becomes zero then the P/V flag is set.

LDDR As LDD, but with automatic repeat if BC is not zero. You can use this op code to transfer a block of data from one area of memory to another by loading the following: HL with the last address of the block you want to move; DE with the last address of the destination you require to fill; BC with the number of bytes to be transferred. LDDR will then carry out the move, and the data will now be in both areas of memory you have chosen.

LDR As LDD but the HL and DE registers are incremented instead of decremented.

LDIR The auto-repeating version of LDI. By choosing either LDDR or LDIR you can fill your new area from the 'top down' or the 'bottom up', which is important if blocks overlap.

NEG This is an accumulator-only op code. It transforms the value in A into the negative two's complement value of the original contents by complementing and adding one. When not dealing with two's complement arithmetic, a program may use this instruction as a quick way of subtracting a value from zero.

NOP 'No operation'. When the CPU fetches this code it does nothing except increment the PC as always — surprisingly useful, especially during program development.

OR This is a logic op code that can take a byte of immediate data, the contents of a GP register, or a value from memory pointed to by HL, IX or IY, and perform a bit-by-bit ORing test between the byte and the A register. The result is left in A, and the flags are set accordingly. For example, if A contains 00001111 binary, OR 83h (10000011 binary) will have the result 10001111 in A.

OUT The write version of IN. The same rules apply, except that the data is sent from the register to the specified port.

OUTD
OUTI
OTDR
OTIR This group of instructions is the complement of the IND group. The contents of the memory location pointed to by HL are output to port (C). HL is then decremented or incremented, B is decremented, and *if* the instruction is a repeating one, the action is carried out until B reaches zero.

POP This is an op code associated with the stack, which is the area of external memory reserved for the CPU's own use: it stores the return addresses for CALL instructions there. POP takes the previous two bytes stored in this area and loads them into the specified register pair (AF, BC, DE, HL, IX, IY). The stack pointer is then incremented twice, leaving it pointing at the next two bytes to be POPed. Stack operations are dealt with later.

PUSH This reverses the POP procedure: it decrements the SP twice, depositing the contents of the specified register pair at the memory addresses created in the process.

RES 'Reset' to zero. The value of any bit of a byte stored in the GP registers or at a memory location pointed to by either HL or the index registers, can be reset to zero. RES 1,A will make bit 1 of the A register become zero.

RET 'Return' from a CALL: this is used at the end of a subroutine in order to return to the original routine. RET retrieves the address that was pushed on to the stack by the CALL op code, and places it in the PC, thereby forcing a jump to that location. Take care if you tamper with the stack or the SP during the subroutine — you may cause the correct address to become irretrievable. This can sometimes be used on purpose, for example to find the value of the PC.

RETI This is a special RET instruction for use at the end of a subroutine called in response to a maskable interrupt. It acts as a normal RET but, in addition, signals sent via the control bus

indicate to the interrupting device that the interrupt has been 'serviced'.

RETN Return from a non-maskable interrupt.

RL This is the first of a set of 'rotate and shift' codes that are best described graphically: refer to **Table 6.1** for a fuller explanation. However, briefly, RL rotates a byte of data left through the carry flag. The highest bit, bit 7, is placed in the carry flag, and bit 6 shuffles over to replace it. All the bits move along one place, and bit 0 is filled with the old value of the carry bit. Unless otherwise stated, all shift and rotate codes can be applied to any GP register or byte in memory pointed to by HL and the index registers.

RLA An exception to the above rule, RLA can only be applied to the A register. It behaves in the same way as RL except that no flags other than carry are affected. The rotates that end with 'A' are all only one byte long and are quicker to execute than the others.

RLC 'Rotate left circular' is similar to RL. It differs in that bit 7 is copied into bit 0 as well as the carry, the original value of which is lost.

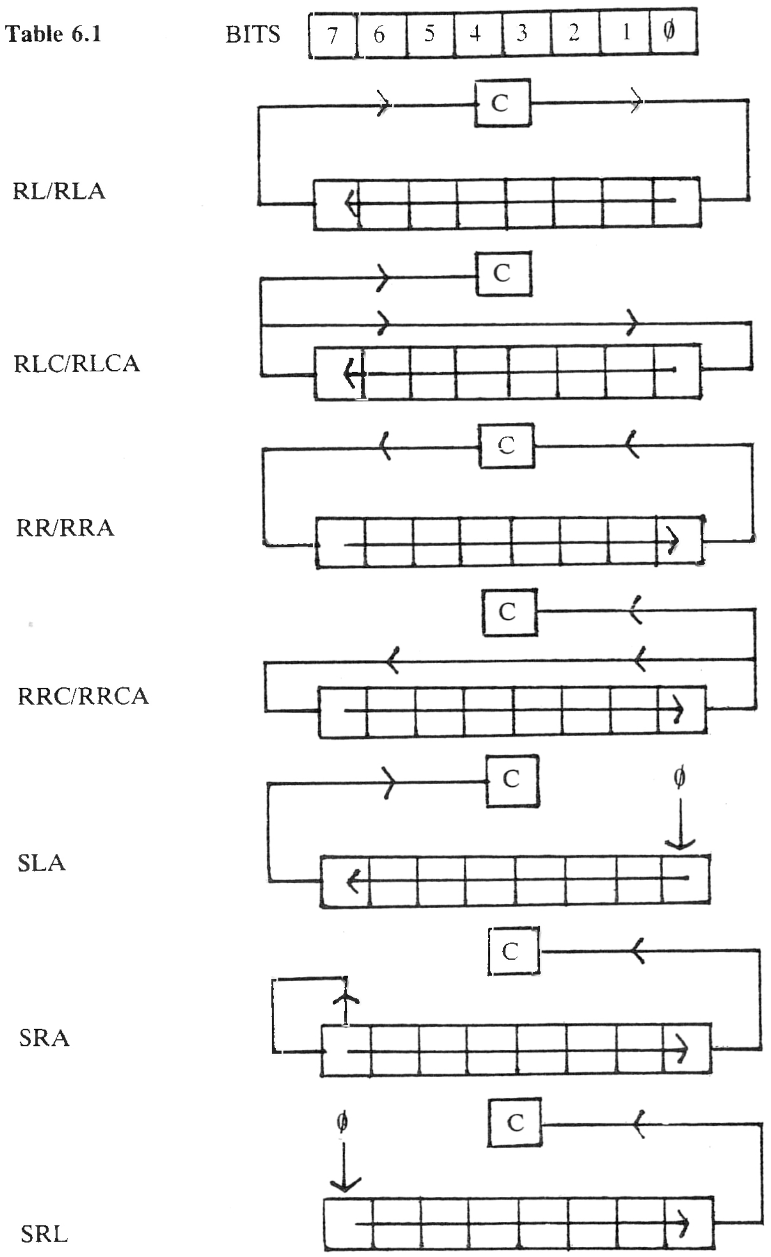
RLCA A circular version of RLA.

RR 'Rotate right'. Palpably, RL the other way!

RRA
RRC
RRCA These are the rotate right versions of RLA, RLC and RLCA.

RRD
RLD These are alphabetically out of order because they are not normal rotate instructions. RLD stands for 'rotate left decimal'. It is used in conjunction with binary coded decimal arithmetic, or in certain cases where you wish to rotate a value by four bits. The HL register must point to the memory location you wish to rotate. The low four bits (or nibble) of the accumulator are placed in the low nibble of the memory location. The contents of the low nibble are placed in the high four bits, the previous contents of which are transferred to the low nibble of the A register. RRD is the same except it rotates in the other direction.

Table 6.1



RST This is not a rotate instruction! It stands for 'restart'. There are eight RST op codes, each of which perform a CALL to one of these specific addresses: 0, 8, 16, 24, 32, 40, 48 and 56 decimal. The RST code is only one byte long, and it is used extensively in the ROM to call frequently-used routines.

SBC 'Subtract with carry'. As ADC, except that a subtraction is performed.

SCF 'Set the carry flag'. Causes the carry flag to become 1, whatever its previous value.

SET See RES. SET makes the specified bit 1.

SLA 'Shift left arithmetic'. This is the first of three shift codes. See RL and Table 7.

SRA 'Shift right arithmetic.'

SRL 'Shift right logical.'

SUB This is an eight-bit only op code that subtracts the contents of either a GP register, or external memory pointed to by HL or the index registers, from the A register. It leaves the result in A and sets the flags. If you wish to perform a 16-bit SUB then you can use SBC having previously reset the carry flag: AND A is the quickest way to achieve this.

XOR This exotic sounding instruction stands for 'exclusive OR', and is in the same mould as AND and OR. It has a logic one result only if the two bits tested are different from each other: thus both two 0s and two 1s will result in a 0. It will crop up often as XOR A because this is the easiest way to zero the accumulator.

The index registers, flags and stack: notes to accompany the glossary

The index registers, IX and IY, are often used as memory pointers. Op codes that use them take the form 'LD A,(IX+displacement byte)', where 'displacement byte' is a two's complement number (see JR) that is added to the index register, before the value is used as a memory pointer to access a location in memory. Hence, if we load an index register with a suitable number at the start of a program, we can have quick and simple use of a block of memory consisting of 255 bytes centred on a selected value.

Spectrum system software uses the IY register to get at the system variables, so it must always be restored to its correct value before there is a return to the operating system. Op code values can be calculated from the equivalent (HL) code, as DDh is the prefix for IX and FDh for IY. The displacement byte is inserted after the (HL) op code value, so, in order to transform OR (HL) into OR (IX+2), for instance, the one-byte code B6h would become the three-byte code DD B6 02 h.

A word about flags. The carry flag is useful for testing to see if one value is greater than another — comparing A with 10 will set the carry to 1 if A contains less than 10. The P/V flag has a number of functions: it is used to test for 'overflows' when performing two's complement arithmetic; it reflects the 'parity' of the result of a logic operation, with an even number of ones in the accumulator setting the flag; it is used by the block op codes as a supplementary zero flag. Also remember that you can change the value of flags accidentally with the POP AF and EX AF AF'.

One quirk of Z80 code that may confuse you is its method of storing two-byte numbers in external memory. It places the low byte in the first location and the high byte in the next. In the case of LD HL,0100, this would be stored in memory as 21,00,01. The second byte represents the eight high bits of the value.

Finally we come to the stack. Think of this as a pile of cards: putting a value on the stack means that it goes on the top of the pile, so that if we were to pick up, or POP, the top card it would always be the last one put down, or the last one exposed when the previous card was taken. The stack pointer holds the address of the next card to be POPed. On the Z80, the SP is decremented before a PUSH, so the stack grows 'top down'.

Monitor program

If you are interested in writing complex programs, an assembler program and reference book would be a good investment. To give you a taste, however, the above glossary, together with the list of op codes in the 'Character Set' chapter of the Spectrum manual will suffice. Program 6.1 is a BASIC program that you can use to enter decimal or hexadecimal data into memory, check what you have done, alter, save, and run the machine code you have created. It contains its own operating instructions, so enter it and save it carefully. After running one of your programs, it prints the value left in the BC register, so load that with any result you wish to examine. Try exploring what the various op codes do, especially the conditional and rotate ones.

Program 6.1: Monitor

```
   1 REM          MONITOR PROGRAM
   2 REM
   5 REM    For pound sign read
   6 REM    read hash symbol
   7 REM
  10 GO TO 9000
  11 REM
  12 REM          Format 1 byte
  13 REM
  20 IF  NOT h THEN  LET a$="  "+ STR$ a
: LET a$=a$( LEN a$-2 TO ): RETURN
  30 LET a$="  "
  40 FOR x= LEN a$ TO 1 STEP -1: LET b=a
- INT (a/16)*16: LET a$(x)= CHR$ (b+48+7
*(b>9)): LET a= INT (a/16): NEXT x: RETU
RN
  50 REM
  51 REM          Format 2 bytes
  52 REM
  60 IF  NOT h THEN  LET a$="    "+ STR$
 a: LET a$=a$( LEN a$-4 TO ): RETURN
  70 LET a$="    ": GO TO 40
 100 REM
 101 REM        Input n
 102 REM
 110 LET a$="Invalid number;try again"
 120 REM
 121 REM        Start here
 122 REM
 130 LET n=0: INPUT (a$+" "); LINE b$
 140 FOR x=1 TO  LEN b$: LET b= CODE b$(
x)-48: IF h THEN  GO TO 170
 150 IF b<0 OR b>9 THEN  GO TO 110
 160 LET n= INT n+b*10^( LEN b$-x): NEXT
 x: RETURN
 170 IF b<0 OR b>9 AND b<17 OR b>22 THEN
  GO TO 110
 180 LET b=b-7*(b>9): LET n= INT n+b*16^
( LEN b$-x): NEXT x: RETURN
 200 REM
 201 REM          Get 1
 202 REM
 210 GO SUB 120: IF n<0 OR n>65535 THEN
 LET a$=y$: GO TO 210
 220 RETURN
1000 REM
1001 REM        EXAMINE
1002 REM
1010 CLS : PRINT  INVERSE 1;"M"; INVERSE
 0;"enu,"; INVERSE 1;"N"; INVERSE 0;"ewa
ddress,"; INVERSE 1;"A"; INVERSE 0;"lter
,"; INVERSE 1;"6&7"; INVERSE 0;" scroll"
: LET a$=z$
1020 GO SUB 200: IF n>65516 THEN  LET a$
="Too high;re-enter": GO TO 1020
1030 LET c=0: LET l=n: PRINT  AT 2,0;: F
OR y=0 TO 19: LET p=y: GO SUB 1210: NEXT
 y: GO SUB 1190
1040 LET a$= INKEY$ : IF a$="A" THEN  GO
 TO 1100
1050 IF a$="6" THEN  GO TO 1130
1060 IF a$="7" THEN  GO TO 1160
1070 IF a$="M" THEN  CLS : RETURN
1080 IF a$="N" THEN  GO SUB 1200: LET a$
=z$: GO TO 1020
1090 GO TO 1040
1100 LET a$="New value"
1110 GO SUB 120: IF n>255 THEN  LET a$=y
$: GO TO 1110
1120 POKE l+c,n: LET p=c: GO SUB 1210
1130 IF c <> 19 THEN  GO SUB 1200: LET c
=c+1: GO SUB 1190: GO TO 1040
1140 IF l>65515 THEN  GO TO 1040
1150 LET l=l+1: RANDOMIZE  USR 23335: GO
 SUB 1190: LET p=c: GO SUB 1210: GO TO 1
040
1160 IF c <> 0 THEN  GO SUB 1200: LET c=
c-1: GO SUB 1190: GO TO 1040
1170 IF  NOT l THEN  GO TO 1040
1180 LET l=l-1: RANDOMIZE  USR 23350: LE
T p=c: GO SUB 1210: GO TO 1040
1190 PRINT  AT c+2,0;"Address>>"; AT c+2
,22;"<<Contents": RETURN
1200 PRINT  AT c+2,0; TAB 9; AT c+2,22,:
 RETURN
1210 LET a=l+p: GO SUB 50: PRINT  AT p+2
,9;a$;"      ";: LET a= PEEK (l+p): GO SU
```

```
B 20: PRINT a$: RETURN
2000 REM
2001 REM        ENTER CODE
2002 REM
2010 CLS : PRINT "Mode 2 allows you to e
nter code"; AT 4,2;"ENTER A VALUE OUT OF
 RANGE TO"' TAB 7;"RETURN TO THE MENU";
AT 10,2;"Current address = "
2020 LET a$=z$: GO SUB 200: LET l=n
2030 LET a=l: GO SUB 50: PRINT  AT 10,20
;a$: LET a$="Code": GO SUB 120: IF n<0 O
R n>255 THEN  RETURN
2040 POKE l,n: LET l=l+1*(l<65535): GO T
O 2030
3000 REM
3001 REM        SAVE CODE
3002 REM
3010 CLS : PRINT "Mode 3 will save code
on tape"
3020 LET a$=z$: GO SUB 200: LET l=n: LET
 a=n: GO SUB 50: PRINT  AT 6,4;z$;" = ";
a$
3030 LET a$="Number of bytes": GO SUB 20
0: LET s=n: LET a=n: GO SUB 50: PRINT  A
T 8,2;"Length of block = ";a$
3040 INPUT "Name of file ";a$: SAVE a$ C
ODE l,s
3050 PRINT  AT 12,3;"REWIND TAPE AND CHE
CK "' TAB 6;a$;'': VERIFY a$ CODE l,s
3060 INPUT "O.K. --- Press enter for men
u";a$: RETURN
4000 REM
4001 REM        RUN CODE
4002 REM
4010 CLS : PRINT "Mode 4 will run machin
e code"
4020 LET a$=z$: GO SUB 200: LET a=n: GO
SUB 50: PRINT  AT 6,3;"The code will be
run from"; AT 8,12;a$; AT 12,3;"Press :-
  R to run code"; AT 14,13;"N to change
address"; AT 16,13;"M for the menu"
4030 LET a$= INKEY$ : IF a$="M" THEN  RE
TURN
4040 IF a$="N" THEN  GO TO 4010
```

```
4050 IF a$ <> "R" THEN  GO TO 4030
4060 CLS : PRINT  USR n: INPUT "Code has
 run;press enter";a$: RETURN
5000 REM
5001 REM        CHANGE BASE
5002 REM
5010 CLS : PRINT  AT 3,2;"The Monitor pr
ogram now works"; AT 5,4;"with numbers t
o the base": IF h THEN  GO TO 5030
5020 LET h=1: PRINT  AT 8,12;"SIXTEEN":
GO TO 5040
5030 LET h=0: PRINT  AT 8,14;"TEN":
5040 INPUT "Press ENTER for the menu";a$
: RETURN
6000 REM
6001 REM       CONVERT NUMBER
6002 REM
6010 LET t=h: CLS : PRINT  AT 4,6;"TO CO
NVERT:-"; AT 8,5;"Hex to decimal press H
"; AT 10,5;"Decimal to hex press D"; AT
15,7;"For the menu press M"
6020 LET a$= INKEY$ : IF a$="" THEN  GO
TO 6020
6030 IF a$="M" THEN  LET h=t: RETURN
6040 IF a$="D" THEN  GO TO 6070
6050 IF a$="H" THEN  GO TO 6100
6060 GO TO 6020
6070 LET a$="Decimal": LET h=0
6080 GO SUB 200
6090 LET a=n: LET h=1: GO SUB 60: PRINT
£1;"Hex = ";a$: GO TO 6020
6100 LET a$="Hex": LET h=1
6110 GO SUB 200
6120 PRINT £1;"Decimal = ";n: GO TO 6020
8000 REM
8001 REM        MENU
8002 REM
8010 CLS : PRINT  AT 1,6;"MACHINE CODE M
ONITOR": PLOT 46,158: DRAW 163,0
8020 PRINT  AT 3,6;"Press the number for
"; AT 4,6;"the desired function"
8030 RESTORE : FOR x=1 TO 6: READ a$: PR
INT  AT x*2+6,5;x;" ------ ";a$: NEXT x
8040 LET a= CODE  INKEY$ -48: IF a<1 OR
```

```
a>6 THEN  GO TO 8040
8050 GO SUB a*1000: GO TO 8000
8060 DATA "Examine Memory","Enter code",
"Save Code","Run Code","Change Base","Co
nvert Number"
9000 REM
9001 REM       Set up
9002 REM
9010 INK 7: PAPER 1: BORDER 1: POKE 2365
8,8
9020 LET h=0: LET z$="Start address": LE
T y$="Out of range;try again"
9030 RESTORE 9200: FOR x=23296 TO 23364:
 READ a: POKE x,a: NEXT x
9100 GO TO 8000
9200 DATA 245,230,24,246,64,103,241,245,
230,7,15,15,15,198,9,111,241,201
9210 DATA 205,0,91,6,8,197,1,13,0,213,22
9,237,176,225,209,36,20,193,16,241,201
9220 DATA 62,2,205,0,91,235,60,205,18,91
,254,21,32,244,201
9230 DATA 62,21,205,0,91,235,61,205,18,9
1,254,2,32,244,201
```

Now for a challenge! Without looking at **Table 6.2,** compile a program that fires an arrow across the top line of the screen. You will need to know the following:

1. RST 10h (code D7) will call the ROM's print routine and place the symbol of the code held in the A register on the screen at the current print position.

2. You can change the current print position (it is automatically updated by the print routine) by sending 16h (the AT code) in A to the print routine (RST 10h again). You should then follow it with the new print coordinates — for example, LD A,16h, RST 10h, LD A,0, RST 10h, LD A,0, RST 10h, will set the print position to the top lefthand corner.

3. The code for the arrow can be 3Eh (the 'greater than' sign).

4. You will have to incorporate a 16-bit delay loop if you are to see the arrow travel!

The 'answer' I have provided in Table 6.2 is not the sole possibility. You should be able to improve on it when you know the ropes. Have fun!

Table 6.2

Address	Hex Code	Assembler
7F80	012000	LD BC,0020h
7F83	110100	LD DE,0001h
7F86	6A	LD L,D
7F87	CDA97F	CALL 7FA9h
7F8A	3E3E	LD A,3Eh
7F8C	D7	RST 10h
7F8D	61	LD H,C
7F8E	10FE	DJNZ −2
7F90	25	DEC H
7F91	20FB	JR NZ,−5
7F93	6A	LD L,D
7F94	CDA97F	CALL 7FA9
7F97	3E20	LD A,20h
7F99	D7	RST 10h
7F9A	6B	LD L,E
7F9B	CDA97F	CALL 7FA9h
7F9E	3E3E	LD A,3Eh
7FA0	D7	RST 10h
7FA1	14	INC D
7FA2	1C	INC E
7FA3	7B	LD A,E
7FA4	FE20	CP 20h
7FA6	20E5	JR NZ,−27dec
7FA8	C9	RET
7FA9	3E16	LD A,16h
7FAB	D7	RST 10h
7FAC	AF	XOR A
7FAD	D7	RST 10h
7FAE	7D	LD A,L
7FAF	D7	RST 10h
7FB0	C9	RET

Part 2

The Sinclair Spectrum

CHAPTER 7
Introducing the Spectrum

In the first section I described the general concepts of microcomputers from first principles. Now it is time to examine the workings of the Sinclair Spectrum computer specifically. There are two models available, but the only difference between them is that one contains 16K of RAM and the other 48K. The additional memory gives the scope for larger programs to be run and useful quantities of extra data to be stored 'on board'.

The Spectrum was one of the first colour computers available at a realistic price to the average consumer in Britain. Much of its design has been developed from its predecessors, the ZX80 and ZX81, both of which utilise the same Z80 microprocessor.

The ZX80 was the first complete computer available for under £100 — it was unable to produce a TV display and compute at the same time, it possessed 4K of ROM containing the operating system and a simple form of BASIC, and included just 1K of RAM, paltry by today's standards and rather modest even in 1980. It was a success even with these limitations, because it sold to people who wanted an off-the-shelf (or rather 'through-the-letter-box') computer aimed at novices and priced accordingly.

The ZX80 was soon replaced by the much improved, million-selling ZX81, with 8K of ROM and a 'slow' mode which maintained the screen display at the expense of very slow operation. These enhancements were combined with a price cut of £20. Soon Sinclair's silent, black and white membrane keyboarded computer was selling like hot cakes and opening up new markets, not only for computers but also for software, books and magazines.

The Spectrum's facilities

May 1982 saw the launch of the Spectrum. It seemed to represent a vast improvement on its predecessor, and in terms of value for money, it was well ahead of the competition — a 16K model was only £5 dearer than a ZX81 with RAM pack. Let's look at some of the noteworthy facilities offered by the Spectrum.

1. A high resolution video display that can be viewed on an ordinary domestic television set. It has a resolution of 256 by 192 dots, and it can print 32 characters in each of its 24 lines of text. Each character space can be displayed as two different colours, termed PAPER and INK and both of these can be one of eight colours — black, blue, red, magenta, green, cyan, yellow and white. In addition, any space can be set to BRIGHT, which increases the luminance, or FLASH, which swops PAPER and INK values at a fixed rate. Software allows drawing to an accuracy of one dot, or 'pixel', on the upper 22 lines of the screen, and the use of 'user-defined' graphic shapes. The screen data is memory mapped, meaning that the display is copied from RAM which is addressed by the address bus, and it occupies over 6K of the memory that would otherwise be available to the programmer.

2. A BEEP facility which allows the creation of simple tones with BASIC, and more complex effects from machine code programs. The sound emerges from a small speaker built into the case. Although quiet, the ear socket can be used to feed an external speaker.

3. An interface makes possible the use of an ordinary audio cassette player as a storage medium. BASIC programs, data and machine code can be saved on normal cassettes, and loaded back into the computer via the MIC and EAR sockets with the supplied lead. The cassette port uses a 'Schmitt trigger' in its circuitry, which enhances reliability so that the data can be transferred at a rate of about 1200 'baud' — fast in comparison with some other machines. The creation of the serial data is handled by the system software, which also supports 'verify', a command that checks that a good recording has been made.

4. A BASIC interpreter accompanies the machine's operating system in ROM. It is a development of the ZX80 and ZX81 BASIC with provision for multi-dimension arrays, a full expression evaluator, floating point arithmetic and syntax checking that occurs on line entry. String slicing is implemented by a TO function — for example, if A$ is 'hello', then A$(2 TO 4) would be 'ell'. BASIC lines are entered from the keyboard by a 'single key' method: each key has up to five different meanings for the operating system, depending on where in the BASIC line the key has been entered, and on the use of the shift keys. As a dialect of BASIC it is excellent for first-time users.

5. A provision to drive a 'port mapped', low cost printer. The printer is controlled by software, thereby simplifying the interface, and was previously sold for use with the ZX81.

6. The keyboard consists of moving rubber keys that are spaced at typewriter pitch. This may not seem much to boast about, but for users of the ZX81 it is a vast improvement.

7. The provision to run a new type of low cost mass storage device called a 'microdrive'. When these finally appeared on the market, it transpired that they consisted of high quality magnetic tape cartridges which travel at high speed. The performance approximates that of 'floppy disk drives', the cost is less than half.

Numerous hardware additions to the Spectrum are available, but there are two from Sinclair Research themselves. The Interface 1 is required to control the microdrives and it also provides an RS232 interface and network capability. The Interface 2 hosts program cartridges and allows two joysticks to be used.

The above specifications combined with low prices have led to the domination of the UK home computer market by Sinclair Spectrum. Before Christmas 1983, the millionth machine rolled off the production line and there were no signs of any waning in the computer's popularity.

Hardware structure

The structure of the hardware of the Spectrum is represented in **Figure 7.1.** If you were to compare this with the actual circuit board inside the case, there would not at first appear to be any resemblance, yet, apart from some small components that cloud the picture, the main features can be identified.

The most obvious is the microprocessor itself. This is a Z80 A — a version of the Z80 which is capable of running with a higher clock frequency than the standard model. An eight-bit wide data bus runs around the board from the microprocessor to the other main components, occasionally being buffered from conflicts by appropriate resistors and diodes. The 16-bit address bus does the same.

The remaining major part of the system is a 40-pin integrated circuit. This 'ULA' chip can be said to perform, with some exceptions, anything that the CPU is unable to do. It is a custom-manufactured workhorse that provides the means of generating a video display — it carries out the more exotic forms of address decoding and all the port address decoding, it provides the clock signal, and more.

The initials represent 'uncommitted logic array' — this is a method of producing, from a standard substance, custom-made integrated circuits for a specific job. The chip manufacturers make a package that contains all the building blocks required for a complex circuit — gates, flip-flops, inverters, buffers, etc. The final task of interconnecting them, using a

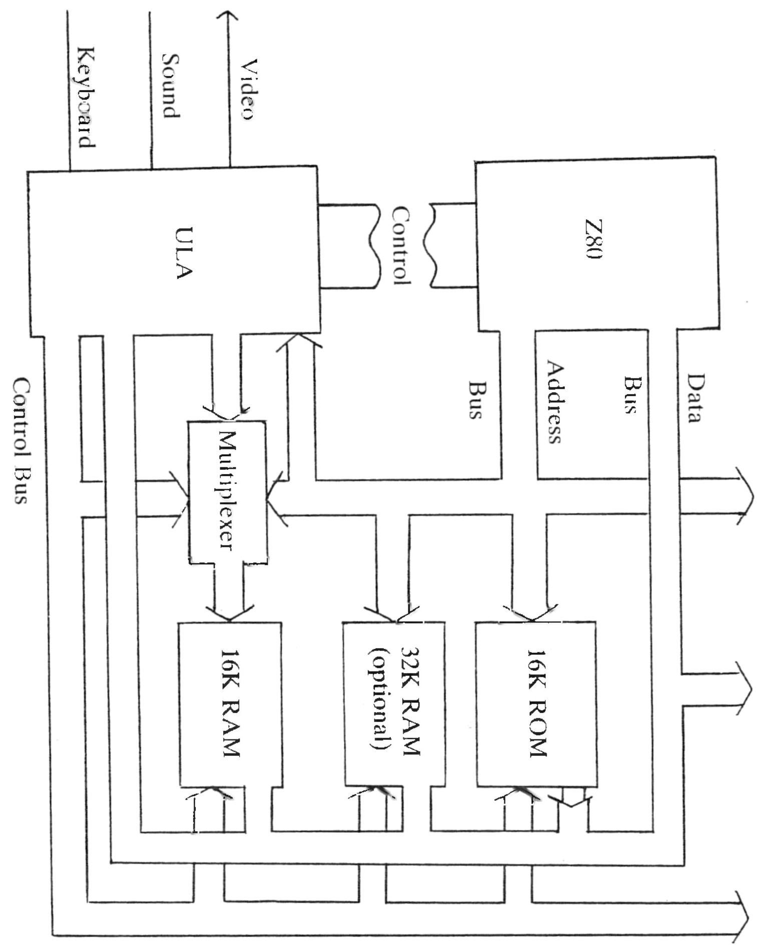


Figure 7.1: Spectrum System Layout

photographic etching method, is left until the required design is finalised with their customers. This makes the job of building a chip to fulfil a specific function much cheaper than using discrete components, especially if large numbers will be required.

There is nothing 'magical' inside the Sinclair ULA. Its role could be undertaken by a collection of logic chips, although the space these would occupy might demand a huge circuit board — the factors of size

and cost work to the advantage of custom-built chips. The ULA is attached to both the data and address buses, and it also receives most of the Z80's control signals. With this information, it generates more specific signals of its own which control the whole computer.

Found in close proximity to it on the board is a 'crystal'. This is the external component of the oscillator inside the ULA: the oscillator generates the clock signal for the CPU and the ULA itself. Other connections to the workhorse include the EAR input socket through which serial data can be fed in from a cassette recorder. Outputs include the video and colour signals, the feed to the MIC socket for recording data, and signals to activate the speaker and keyboard.

A ROM chip is the next most obvious thing on the circuit board. It is capable of storing 128K bits, that is, 16K bytes. This is the home of the system software, permanently etched as connections in silicon. The chip is linked to both address and data buses — its chip enable signal comes from the ULA.

The 48K machine possesses 16 RAM chips, the smaller computer only 8. How the extra eight chips of the larger model are housed depends on the age of your machine. The standard complement of 16K bytes of RAM is made up from eight 4116 16K bit dynamic RAM chips, arranged so that each chip contributes to one line of the data bus: thus a byte of data stored at a particular address will have each individual bit stored in a different RAM chip. This first 16K of RAM is treated to its own address lines because the ULA chip needs to access that area of memory independently of the CPU, in order to collect data to be sent to the screen. To facilitate this, chips known as 'multiplexers' (74 LS 157) are used. They have two inputs for each bit of the address bus, one from the Z80 bus and one fed directly from the ULA. A control signal from the ULA determines which of these is passed on to the RAM chips.

Feeding a domestic television requires a good deal of electronics. Not only must the colour and luminance signals be combined with suitable synchronising pulses and coded into one 'composite video' signal, but also that signal must be 'modulated'. This process imitates the workings of a television transmitter; consequently, the area of the circuit board which does the 'transmitting' is enclosed in a metal screening can, so that the computer will not appear on the airwaves. A weak but identical version of the signals picked up by your TV aerial is then passed down a screened lead to the aerial socket on your set. As you know, different TV stations broadcast on different channels: the Spectrum, in common with most computers, transmits on channel 36.

Most of the remaining components are concerned either with buffering the various signals between the major areas of the circuit board and the connectors, or with providing the correct supply voltages without which the integrated circuits could not function. The voltage which is supplied by the external mains adaptor is 9 volts, but many of the chips need only 5 volts to function, and any more would cause damage: other chips require +5, +12 and −12 voltages, relative to their GRN pins.

A word of advice for the incurably inquisitive amongst you: if you take your machine apart to investigate inside, you will almost certainly invalidate any guarantee that the makers or suppliers offer, but you are unlikely to do any damage if you take care. Wait for the warranty to expire if you can possibly contain yourself.

The computer is held together by five cross-headed self-tapping screws located underneath the case. Having pulled out the power plug and removed the screws, turn the computer the right way up before lifting off the top half. Be gentle! The keyboard is connected to the lower part by two *very* fragile ribbon cables. These can be pulled gently from their sockets to allow the complete removal of the top. Now you can snoop away to your heart's content. However, don't drop anything inside, particularly if it's metal. Another screw holds the printed circuit board to the lower part of the case, but there is little underneath other than copper tracks. When reinserting the tails of the ribbon cables before reassembling the computer, take great care not to kink them as they will bend all too easily and may fracture. And don't leave anything inside!

CHAPTER 8
The Memory Map

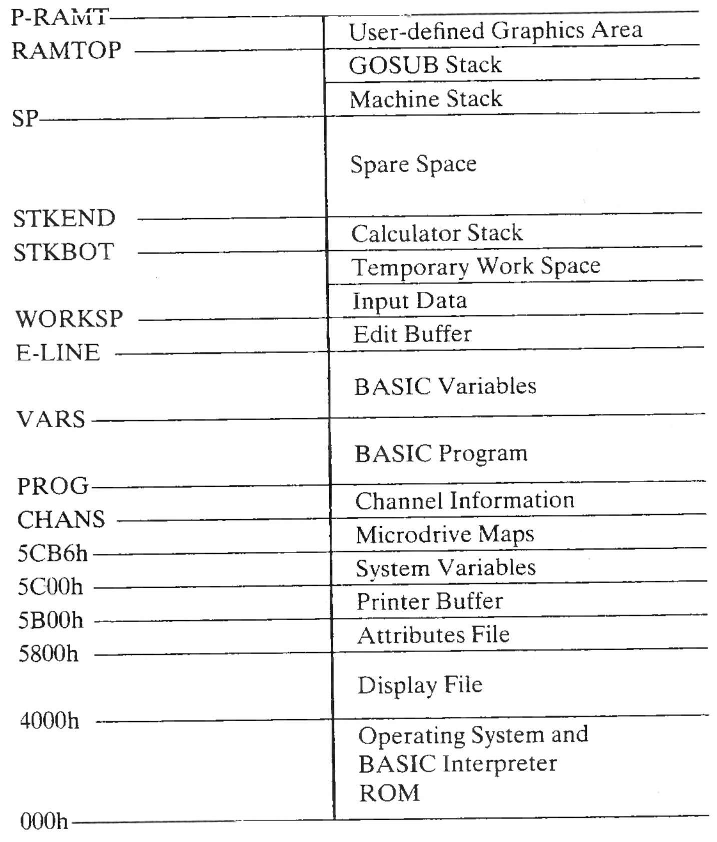
The most prevalent limitation that the majority of microprocessors impose on the design of computers is the size of the address bus. With 64K bytes at their disposal, early microcomputers had room to spare, but, with the falling price of memory chips and the demand for increased storage space, recent products often use all their available addresses. The 48K Spectrum falls into this category. The layout of memory addresses is termed the memory map — on the Sinclair Spectrum this is allocated as follows.

Between addresses 0 and 16383 decimal (0000h to 3FFFh), a 16K ROM holds the system software (including the routines that provide input and output via keyboard, screen and speaker) and the BASIC language interpreter. RAM is found between 16384 and the top of the map. In the 16K machine, this extends only as far as 32767 (7FFFh), and the remaining addresses are vacant. The 48K computer has RAM at all the addresses up to 65535 (FFFFh): on both machines, not all the RAM locations are available to the user. Study the memory map in **Figure 8.1,** and you will see that, from 4000h upwards, a great deal of space is taken up before the location marked PROG, which is the storing place for the beginning of any BASIC program entered. (Please note that this figure is not drawn to scale.)

The pixel data for the screen is stored between 4000h and 57FFh, which is called the 'display file' — this holds the information that remembers whether any dot of the screen is paper or ink. It makes use of a large area, 6K bytes, but that is the penalty for good graphics. Starting at 5800h is another file for display purposes, the attributes file. It is worth repeating that the colour resolution of the Spectrum is low, and 300h bytes are sufficient for the data which instigates the ULA chip to generate the colour part of the picture.

Immediately following the attributes, is a 100h block of memory called the printer buffer: this is used in conjunction with software to drive the low-cost printer available for Sinclair machines. It is in the buffer that a line of text is translated into eight lines of 256 pixels producing the character shapes. This data is then sent to the printer as a serial stream of ONs and OFFs, which controls the stylus of the printer as it travels across the paper.

Figure 8.1:Spectrum Memory Map
System Variables or Hexadecimal Addresses



System variables

The next block of addresses, called the system variables, is concerned with the running of the machine's own system software. Any program which is held totally in ROM is hampered by the fact that it cannot store variables within its own memory area, except for the few bytes it can keep in the registers of the processor, which are unused by the program itself. It may, of course, lodge the bytes on the stack, but retrieving them at random is impossible. Consequently, the area of memory from

5C00h to 5CB6h, is set aside for storing such information as the current colours to be used for printing to the screen; where the BASIC program has stored its variables; or even, where the BASIC program itself starts (this moves about when extra equipment is attached to the computer).

You will find a list of the system variables in the manual — the list indicates briefly what the data is that they store. Some can be altered by the programmer to achieve the desired effect, changing others will cause a peremptory crash of the whole machine. RAMTOP, the variable at 5C82h, for instance, can be POKEd with a lower value to deceive the computer about the existence of memory locations — normally, when the command NEW is carried out, all memory is wiped clean, but with this device the locations above your RAMTOP wil not be affected. On the other hand, if you POKEd a different value into PROG, the address at which the BASIC program starts, the interpreter will fail to find the program when you RUN, causing a crash. (You may just be lucky and POKE in a value that is the start of a line in the program, but still any earlier lines will be lost.)

It is worth noting that the values held by the two-byte system variables are stored in the manner of the Z80 itself — that is, the least significant byte first. Let me give an example that also shows the advantage of using hexadecimal numbers.

PROG stores an address. It is a 16-bit number, so it needs two bytes of memory space, 5C52h and 5C43h. If the BASIC is in its normal place, then PEEKing these two values should provide a standard result. Try entering PRINT PEEK 23635 and PRINT PEEK 23636 — you should get 203 and 92 respectively. In order to find out which addresses these represent, you need to multiply the *second* value by 256 — PRINT 256*PEEK 23636 will give you 23552. Now add the value in the first location by entering PRINT 256*PEEK 23636+PEEK 23635. This should elicit the result 23755, ie (256*92)+203.

If the Spectrum could be persuaded to give its answers in hexadecimal (and programs are available offering this facility) the results of PEEKing the RAMTOP system variable would have been CBh and 5Ch. We can multiply a hex number by 256 through simply adding two zeros to the nd — 5C becomes 5C00h, add CBh, and we have 5CCBh. Guess the decimal version of 5CCBh! Merely by reading the two hex bytes in reverse, we arrive at the true two-byte value stored in RAMTOP. Admittedly, using hex numbers on a machine that cannot understand them is nonsense, but with the use of an appropriate 'monitor' program, many tortuous calculations can be avoided. If you would like proof that 23755 is indeed the starting point of the BASIC program, then try the following procedure.

Enter a single line of BASIC into an otherwise vacant computer — perhaps, 100 REM The first and only line — and then POKE the first two bytes of the program area with zeros, ie POKE 23755,0 and POKE 23756,0. Now list the program, and see the disruption caused. This is because each line is stored with the first two bytes, which represent the line number. If you POKE in other values, you will discover that, to add to potential confusion, these line numbers are stored the 'correct' way round — that is, the most significant byte first.

Returning to the memory map, let us examine what lies between the end of the system variables and the start of any BASIC program. There is an area of RAM designated as microdrive maps — without a microdrive this is non-existent. Finally, before we reach the BASIC program area, there is a small table of data known as the 'channel information'. The Spectrum uses a method of inputting and outputting data that assigns a channel number to each of its available types of communication. The device which is currently the active 'stream', is the one that receives and sends data. For example, if the screen is the active stream, then the data stored for that output channel in the channel information table is used by an all-purpose output routine to discover the whereabouts of the routines that deal with screen output. If you were to open the printer stream, the PRINT command would instigate the sending of what would normally go to the display files for transmission to the TV, to go instead to the printer buffer.

This may seem an unnecessarily complicated way of working, but it comes into its own when extra channels are added to the system. The bulk of the I/O (input and output) routines already exist and, by looking up the information for the current stream, they can be diverted as required to deal with each special case. The table is built up during the initialisation process of the system, by checking which channels are present (including peripherals) and making entries accordingly. Try the following program, which outputs to a stream other than the normal part of the screen. The # symbol, representing 'stream', is an extended and symbol-shifted version of the 3 key.

```
10  PRINT # 1; "Usually this would be at the top"
20  IF INKEY$="" THEN GOTO 20
```

The program illustrates what happens if you output data when stream one is selected — this is the lower part of the screen normally reserved for INPUT prompts and error messages. If you try other numbers, you will normally get an error message, as there is no information for the output routine to find in the table regarding other streams.

BASIC and variables

We have now reached that part of the memory map which holds any BASIC program. The size of this area and the one immediately above, the variable area, will depend on the size of program loaded, and the number of variables it uses. Each time you enter a line of BASIC, the memory contents above the point where it is stored are shifted up to make room, and the relevant pointers that are held as system variables are altered accordingly. The next space is for editing lines of a program. When you select them, they are moved here and a copy sent to the lower part of the screen. As you move the cursor, you can make any changes, until ENTER is pressed. Then the line with the matching number is found, deleted, and replaced by the new, corrected line from the edit buffer. This is also where completely new lines are created as you type them, but before you press ENTER.

Next up the map are spaces that can be expanded to hold temporary data, such as numbers entered in response to an INPUT prompt, or strings that are being manipiulated. Before we reach the spare space, there is a type of stack. This is one used by the system software for handling numbers. As I mentioned earlier, the Spectrum employs a method to represent numbers that requires five bytes to store them: this allows it to manipulate values both small and large to a reasonable degree of accuracy. When handling these numbers, the computer uses the calculator stack.

Free memory

The starting point of the free memory area (spare space) rises and falls as the blocks below expand and contract during program entry, editing and running time. At the top of the free space are two more stacks, the machine stack and the GOSUB stack. The CPU uses the first of these, with the SP register pointing to the location which is currently the 'top' of the stack. In fact the stack 'grows' from the top down — early values passed to the stack will be stored high in memory and, as more data is pushed, the SP will point to lower addresses. The second stack is used by the BASIC interpreter to store RETURN, line and statement values which are used as GOSUB returns. The last value stored is 3E00h, representing an illicit line number, so the BASIC knows that you have tried to RETURN without GOSUB. Finally, at the very top of the memory is a block of bytes — FF58h or FFFFh on the 48K Spectrum, 7F58h to 7FFFh on its smaller sibling. This holds the user-definable graphic dot patterns which are at first filled with the shapes of their corresponding letters, but defining your own shapes will change their contents.

It is worth noting that, at switch-on, none of the RAM addresses hold anything relevant, so any data must be written in during system initialisation. This task and many others are performed in the time between the appearance of the blank screen at the moment of switching on, and the printing of the copyright message: not long in human terms, but for a computer plenty of time for a great deal to be accomplished.
We have come to the end of our guided tour of the memory map. Much of the above will be expanded upon soon, but now it is time to look at how the computer achieves its communication with the outside world.

CHAPTER 9
The Keyboard

A computer can be of little use unless it has some means of receiving information. When a Sinclair Spectrum computer first emerges from its packaging, there are two ways of feeding information to it, by cassette or by keyboard. You can load the memory with data from a cassette player, but the programs would almost certainly need some other data which would come from the keyboard. Even when the screen flashes with the familiar message, 'Press any key to begin', it is expecting data, in its most simple form, from a key.

The way keyboards actually feed information into a microcomputer varies with each machine. Some microcomputers can interrupt whatever the CPU is doing, and call attention to themselves whenever they have something to pass on: what then happens depends on the software that has been built into the machine.

The Spectrum uses a slightly different approach. The keyboard itself is a completely passive device, but the operating system scans it at very short, regular intervals to ascertain whether any key is being pressed. If it is, the operating system stores the value in a particular memory location, which the system's software can read and respond to as required. In order to understand how these regular intervals occur, we shall first need to look at some of the functions of the Z80 processor which I have so far failed to mention.

Using the keyboard from machine code

The two pins that concern us are called NMI and INT. They empower a microcomputer designer to achieve a machine which, although it is still able to do only one thing at a time, can nevertheless share that time between different tasks. For example, some printers, such as the old teletype terminals which you may have seen (and heard!), are very slow: when they are sent a character to print there is a considerable delay, in computer terms, before the next one can be sent. Instead of having your microcomputer standing idle during this time, you can employ it elsewhere, whilst the printer can let it know that it is ready for another character by putting a signal on the INT pin.

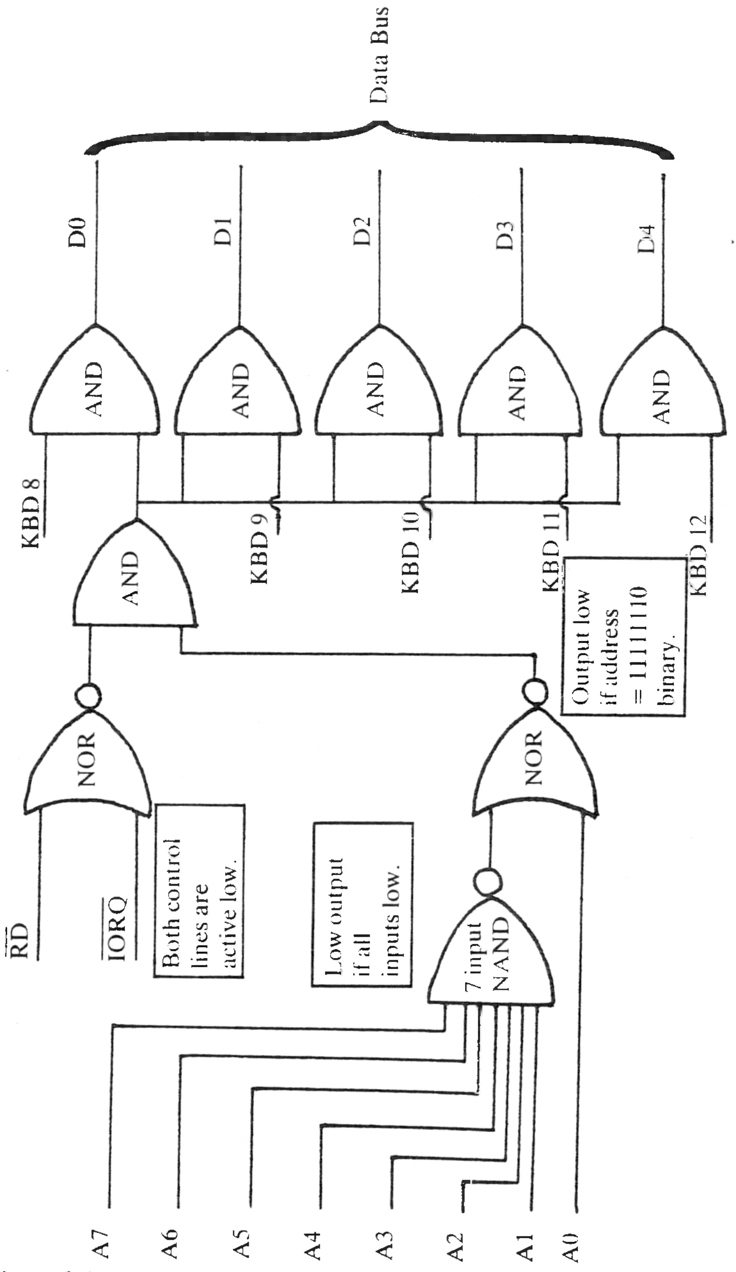
So what's all this nonsense about ancient printers and impatient Z80s? While the interrupt system is available, it can be used to make your Spectrum appear schizophrenic. Every fiftieth of a second, the ULA chip sends out a video signal, starting with a 'frame pulse' with which the TV or monitor synchronises. In the process of generating video, the ULA creates a pulse of zero volts which it sends to the CPU's INT pin every fiftieth of a second. The Spectrum's software has configured the Z80 to respond to this signal in the simplest manner available. When the Z80 has finished performing each instruction, it checks its interrupt input; if it senses a negative pulse, it saves the contents of the program counter (on the stack), and loads it with 0038h (56 decimal).

We have now directed the processor from whatever it was doing and pointed it towards the machine code routine stored at 0038h in the ROM. This routine does three things. Firstly, it increments the system variable called 'frames' — this is a three-byte value held in RAM locations 5C78h to 5C7Ah, which is used to measure time through the BASIC command PAUSE. Secondly, the routine scans the keyboard to see whether any key is depressed. If this is the case, it decodes the key, taking into account shift keys, and unless two or more keys are pressed it stores the code in RAM location 5C58h. Finally, the old value of the program counter is retrieved from the stack and reloaded: thus the Z80 returns to exactly where it left off before the interrupt occurred.

The program ensures that the CPU registers are not changed. Any that are used have their previous values stored and then reinstated once the interrupt has finished, in order that the program that has been interrupted is not at all affected. The operating system can take action on the keypress if it wishes — it may be in command mode and react by interpreting the key as an instruction, or it may be running a program and therefore ignore the key completely.

When you can follow machine code, the main scanning routine will be of interest. It can handle 'rollover', which is easier to demonstrate than to explain. Switch on your machine and clear the copyright message by pressing PRINT. Now hold down the 'p' key, which will auto-repeat and print a stream of 'p's to the screen. Now press another key as well — the repeat printing stops because the machine does not know which key you wish it to send to the screen. Lift your finger off the 'p' key and the second letter will appear. The automatic repeating is also handled by the scanning routine.

Now to explain how the keys are connected. There are 40 keys, each with their own switch. These form a matrix connecting the eight high address lines with five lines numbered KBD 9 to KBD 13. Circuits within the ULA sense when the Z80 is performing an IN (FEh) operation, and respond by placing the voltages from the KBD lines on to the low five bits of the data bus. A study of **Figure 9.1** will show you


Data Bus
D0
D1
D2
D3
D4
AND
KBD 8
KBD 9
KBD 10
KBD 11
KBD 12
Output low if address = 1111110 binary.
NOR
RD
IORQ
Both control lines are active low.
Low output if all inputs low.
7 input NAND
A7
A6
A5
A4
A3
A2
A1
A0
Figure 9.1

how this is achieved: remember that, although you can build the circuit out of discrete components, the Spectrum contains them as part of its ULA. Figure 9.1 is a good example of port decoding. When IORQ and RD are both low, then the Z80 is trying to read the port whose address is held on the eight low bits of the address bus. The function of this circuit is that, when the CPU executes IN (FEh), the five low data lines are fed from the keyboard lines.

In **Figure 9.2,** follow the line marked KBD 9, and study how it can be connected to any of the eight high address lines. It is important to note how the line is pulled high by the voltage at the other side of the resistor — KBD 9 is high until one of the keys is pressed, and contact is made with the appropriate address line. Even when this happens, the data line will only go low if the address line it has been coupled with is also low. The diagram contains some previously unexplained symbols on the address lines, labelled as diodes. You may remember that I earlier described a diode as the simplest type of semiconductor device, as it has the property of passing current in only one direction. The diodes ensure that, if more than one key is pressed, their effect does not extend any further back down the address lines than the diode. This prevents keypressing from interfering with the other operations of the address bus.

The method of placing different levels on the high address lines depends upon the behaviour of the Z80 when performing IN instructions. There are two ways of reading in data from a port. One of these takes the form IN A,(N), where N is the number of the port that is required to supply data to the accumulator register. This eight-bit number is placed on the eight low address lines, and the previous content of the accumulator placed on the eight high lines. When the data is read in, if you load the accumulator with, for instance, 7Fh (127 dec), which has the binary form 01111111, and then give the instruction IN A (FEh), bit 0 of the result read into the accumulator will be high — unless the space key is pressed, in which case it will be low. So you can see from this simple example how you can test for individual keys. The other Z80 IN instruction takes the form IN A (C), and it allows you to load the C register with the number of the desired port. Even more usefully, the value of the B register is placed on the eight high lines, so that the accumulator does not need to be continually reloaded.

Scanning the keyboard

Refer again to Figure 9.2. It shows the layout of the keys, and how they are connected to the address and port lines. Program 9.1 provides a simple illustration of keyboard scanning which checks whether keys are pressed on either side of the computer. It is in machine code because

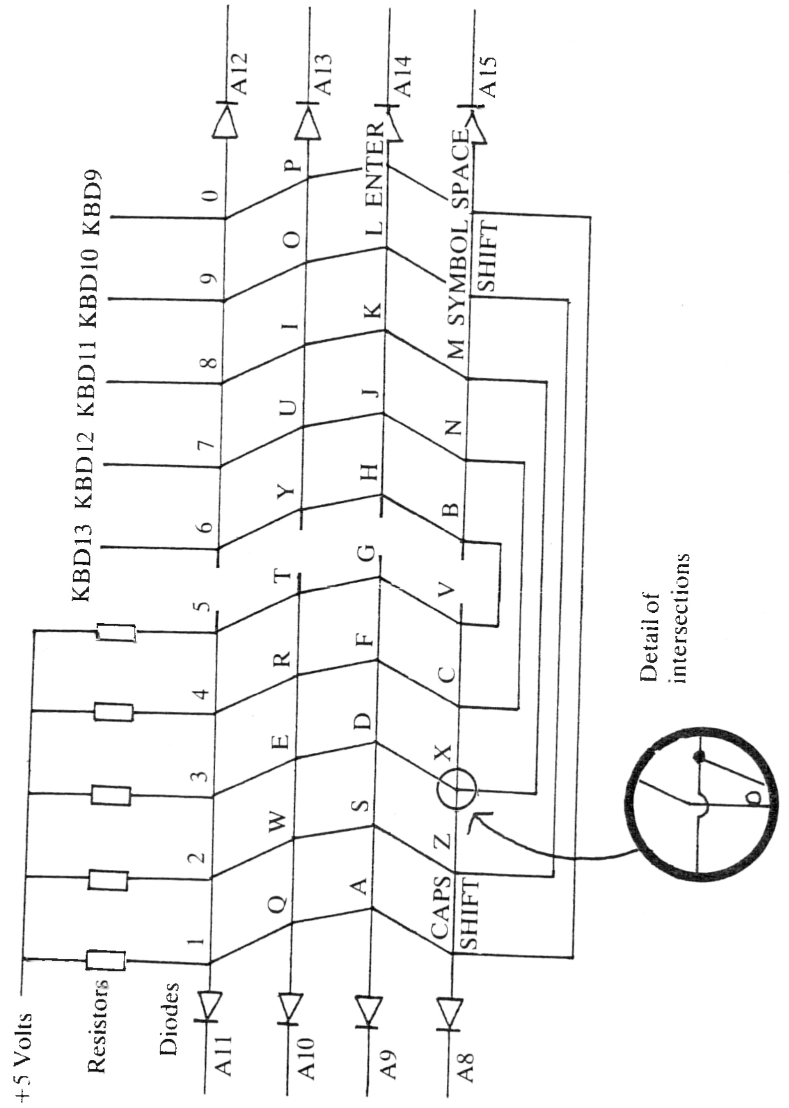


Figure 9.2

this allows the easy manipulation of the individual bits passed between the keyboard and the CPU. Although the Spectrum possesses OUT and IN as part of its BASIC, testing individual bits can be slow and awkward as their arguments are in decimal. You could modify Program 9.1 to test for any half line of keys, or even certain keys within a row, by masking off more than the three high bits read into the A register — ANDing it with, say, 10h, will test the first key of each row only. The program will be of interest to potential games writers; it can be used in a BASIC program, or the code merged into one of machine code. Incidentally, the three high bits that are read in along with the five keyboard bits

Table 9.1: Machine Code for Keyboard Scanning

Address	Hex Code	Assembler
RAMTOP+1	010000	LD BC,0000h
	3EF0	LD A,F0h
	DBFE	IN A,(FEh)
	2F	CPL
	E61F	AND 1Fh
	2802	JR Z,+2
	CBC1	SET 0,C
	3E0F	LD A,0Fh
	DBFE	IN A,(FEh)
	2F	CPL
	E61F	AND 1Fh
	C8	RET Z
	CBC9	SET 1,C
	C9	RET

should always be masked off. On early models of the Spectrum, they always returned zeros, and many professional programs simply ignored them. When the second and third issue machines appeared, much embarrassment ensued, as these bits took on a random nature causing spaceships, monsters and the like to roam about on their own accord!

Program 9.1: Keyboard Scanning

```
  10 REM        Keyboard scanning
  11 REM        function
  12 REM
  20 REM        Lower RAMTOP
  21 REM
  30 RESTORE : LET x=( PEEK 23730+256* P
EEK 23731)-26: CLEAR x
  40 REM
```

```
  41 REM        Poke Machine code
  42 REM        into memory
  43 REM
  50 LET x=( PEEK 23730+256* PEEK 23731)
+1
  60 FOR y=x TO x+24: READ z: POKE y,z:
NEXT y
  70 REM
  71 REM        Print USR address
  72 REM
  80 PRINT  AT 1,2;"The routine is calle
d by the"; TAB 7;"function USR ";x
  90 REM
  91 REM        Example
  92 REM
 100 PRINT  AT 8,10;"Press any keys"
 110 PRINT  AT 12,14;: LET a= USR x
 120 IF a=0 THEN  PRINT "NONE "
 130 IF a=1 THEN  PRINT "LEFT "
 140 IF a=2 THEN  PRINT "RIGHT"
 150 IF a=3 THEN  PRINT "BOTH "
 160 PRINT  AT 16,10;"USR ";x;" = ";a
 170 GO TO 110
 180 REM
 181 REM        Machine code data
 182 REM
 190 DATA 1,0,0,62,240,219,254,47
 200 DATA 230,31,40,2,203,193,62,15
 210 DATA 219,254,47,230,31,200,203,201,
201
```

What if you want to use the built-in scanning routine for your own purposes? Remember that it is interrupt driven — the maskable interrupt can be disabled, in machine code, with the aid of the DI instruction, which causes the CPU to ignore its MI pin. There are a number of ways to elicit a value from the keyboard, the choice of method depending on the kind of program you are writing.

If you leave the interrupts enabled this may upset any critical timing involved, but if you *do* decide to leave them enabled, then proceed as follows. Test bit 5 of flags, the system variable stored at 5C3Bh (23611 dec): if it is set, then a key has been pressed since the last time that flag was reset. You will find the ASCII code for that key in the system variable location, LAST K, 5C08h. If you are only interested in the caps shift value, it is stored in location 5C04h (23556 dec), and it is updated every interrupt, so you will not need to test flags. On the subject of CAPS SHIFT, you can set caps lock from within a BASIC or machine code program, by setting bit 3 of FLAGS 2 (5C69h). To see this happen, enter POKE 23658,8:INPUT A$.

It is possible that an interrupt will occur between testing flags and fetching the value, which may produce the occasional spurious result. In a machine code routine this can be circumvented in two ways. You can ensure that the CPU marks time with the HALT instruction — it will perform NOP until it receives an interrupt. By inserting a HALT in the code immediately prior to reading the system variables, you gain a breathing space of one fiftieth of a second before the next interrupt.

If you do leave the interrupts enabled but wish to protect your routine from the possible tamperings of machine code buffs, make sure that you set interrupt mode 1 at the start of your program. The alternative method is to disable the interrupts altogether and call the scanning routine yourself. As it is located at 38h, the RST 38 op code will do the job in one byte. The system variable will be set accordingly, and the A register will hold the value from LAST K. The scanning routine contains an EI instruction, so you need to use DI again.

It is worth nothing that there is another keyboard scanning routine contained in the ROM, which tests the keyboard to see if BREAK is pressed. When it runs a BASIC program, or performs some commands, such as save or load, the Spectrum makes frequent calls to this routine in order to allow the user to stop the machine and return to the command mode. If you want to use the BREAK testing routine it can be found at 1F54h: it returns the CARRY flag reset to zero if both BREAK and CAPS SHIFT are pressed. This subroutine is quite short — you may wish to copy it for your own purposes, and perhaps modify it to respond to other keys.

The keyboard is not only capable of producing ASCII codes. Each key can also generate the BASIC tokens that represent commands and functions. This is dependent upon what mode the keyboard is in, as shown by the cursor — when it is a flashing K, then the next keypress is translated by the operating system as a command. Pressing 'p' in the command mode generates the value F5h: it only appears on the screen as PRINT because the output routine recognises it as a special case, and it looks up in a table (stored in the ROM from 95h to 204h) what string of characters to send to the screen or, for that matter, what is the currently selected output stream. So the Spectrum saves space and time — it does not need to store five bytes when it represents PRINT, and need only decipher one byte for each command or function. And we need never concern ourselves with the spelling of words such as RANDOMIZE!

The keyboard of the Spectrum is a fairly simple device to scan, and

the operating system does so thoroughly, but if you are writing in machine code you may wish to work out your own scanning routine. A table in the ROM (205h to 22Bh) will help you in decoding the values returned from the scan as this holds the ASCII codes in the order in which the keyboard is laid out. If you are an excellent typist who has fitted a real keyboard to your Spectrum, you may be able to 'beat the scan': how about a routine that can store, say, the last sixteen keypresses? No, I haven't written one — but then I'm a slow typist!

CHAPTER 10
Pictures on the Screen

If a certain Scottish electronics' pioneer had got his way, the television set sitting in the corner of your room would contain a disc with lenses around its edge, spinning at great speed. Fortunately for the video industry, history has left us with a much better system. Both Baird's spinning discs and today's electronic replacement have the same principles of scanning at their roots: the only disadvantage of the modern method is that it is harder to understand.

How television pictures are displayed

The tube which displays the picture we see on the television set is called a 'cathode ray tube'. It is represented in cross section by **Figure 10.1.** The

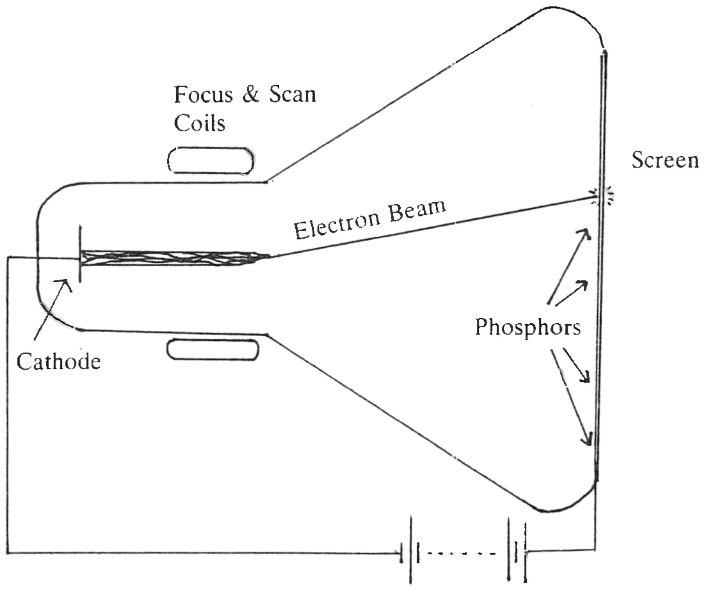


Figure 10.1

air is evacuated from inside the tube and the front face coated with small particles called 'phosphors'. At the back of the tube is an 'electron gun'. Here a piece of metal called a cathode is heated up, and a high enough voltage is generated between it and the phosphors to cause the electrons at the cathode, already excited by the heat, to travel to the highly attractive positive voltage area that exists at the phosphors. As there is nothing to impede their path, a stream of electrons will flow through the vacuum to their goal.

A magnetic field, and other voltages, can have a marked influence on the path taken by these electrons, so it is possible to use electro-magnetic coils, and extra cathodes and 'anodes' (the positive version of cathodes), in order to deflect and focus the 'beam' of electrons travelling to the phosphors. This beam can be made to arrive at any particular spot on the screen. The special properties of the phosphors are significant. If bombarded with lots of electrons, a phosphor will give a glow of light. Within certain limits, the more electrons the greater the glow given off — very useful indeed.

So we have a method of lighting up any particular point on the TV screen with the application of suitable control voltages to the various components of the cathode ray tube. It is now a relatively small step to perceive how we might 'paint' a picture by aiming the beam at various parts of the screen and making them glow, moving on to others and returning before the original glow has subsided. **Figure 10.2** shows how this is done — the screen area is divided up into horizontal lines and a line is scanned. The intensity of the beam varies, resulting in the different brightnesses required. The beam is then reduced in intensity as it 'flies back' to the start of the next line and the process is repeated. When the bottom of the screen has been reached, the beam returns to the top and begins again, but this time it fills in the spaces between the first set of lines. This whole process occurs very quickly, and only a twenty-fifth of a second elapses before the beam returns to the first phosphor it scanned.

In order to make an intelligible picture out of the screen display, its associated circuits must be sent a number of pieces of information. These are pulses which instigate the scanning of a new line or a new frame, and control the brightness generated. The data is contained in an analogue signal called the 'video' signal, which employs negative-going pulses to govern the line and frame circuits, whilst the brightness information is managed by a varying positive voltage coming after the line synchronising pulses. **Figure 10.3** is a graph of one line of video signal. The signals received over the airwaves or from the TV socket of the computer are video signals that have been modulated in the same way as radio signals, so that they can be transmitted and then decoded by the 'tuner' circuits in the television set.

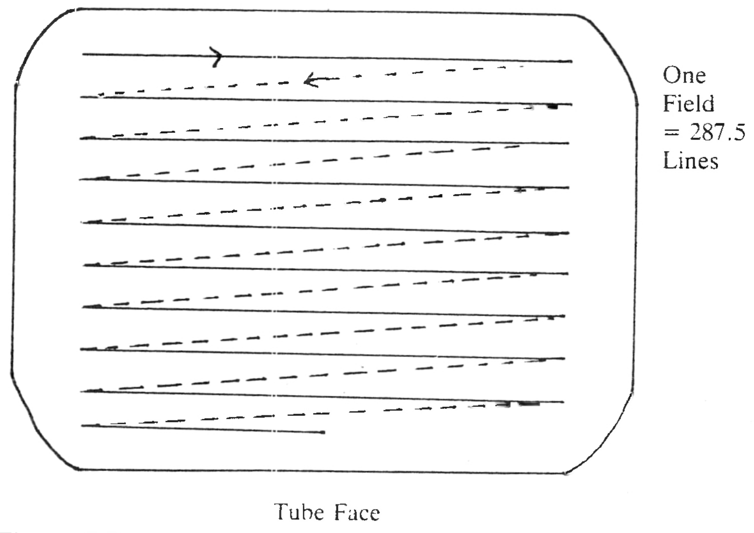


Figure 10.2

As for colour, it makes matters even more complex. Any colour can be made up from a mixture of three very pure sources of 'primary colour' — red, green and blue. A colour television has three kinds of phosphors coated on the screen. The tube contains not one but three electron guns, and masking between the cathodes and the screen ensures that the beam from one gun only lands on one colour of phosphor. The size of the different colour areas is so small that from a distance they merge into one source of light to the human eye. With all three guns set to maximum, the screen will still glow white, but if the relationship of the output of the guns is varied different colours are displayed. Special colour monitors only require the three colour signals to be sent separately and they will achieve a very high quality of display.

Normal televisions expect the colour information to be included with the video signal in what is known as a 'coded composite video' signal, as this is is how the broadcasting system transmits colour. It means that the channel does not take up any more airspace and monochrome receivers can share the same signal. The system used to code the colour information is complex: suffice it to say that colour 'difference' signals are modulated and mixed with the video in much the same way that more than one TV channel can transmit on the airwaves, and a tuner circuit can distinguish between them.

The above sparse description should at least familiarise you with some of the terms associated with video, and give you a clear idea of the way a television set draws its pictures, by scanning across the screen at a very fast rate. Let's now return to your Spectrum and see how it goes about generating the video signal, via its modulator, to feed a television set.

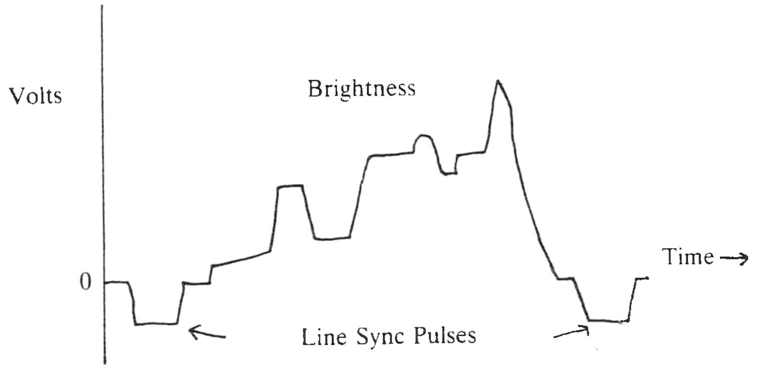


Figure 10.3

The Spectrum video signal

I mentioned earlier that much of the contents of the ULA are concerned with the screen display — the data for the generation of that display is stored in two files in the area 4000h to 5AFFh. Output from the custom chip includes the 'sync' pulses, and the analogue colour and video levels. Timing for the pulses is provided by dividing down the clock frequency. The CPU can signal to the ULA information that controls some of the aspects of how the video is generated. By writing to the three low bits of port FEh, the colour used for the border can be changed — the value corresponds to the colour numbers used by BASIC: 0 is black, 1 is blue, and so on up to 7, which is the equivalent of white. This exemplifies the colour mixing — the three bits each control a primary colour, so if we mix red and green by sending 110 binary to the port, the colour displayed for the border is yellow, the same result as a mixture of red and green light.

When the Spectrum is producing pictures, what is going on in the ULA? Completely independent of the processor, it produces a frame pulse to commence the picture at the top of the television screen, and then it sends out line pulses at the right intervals. For the analogue video information, it uses the border bits as the data to generate the corresponding colour and brightness levels, for insertion between the line pulses of the composite video. After sending out a number of lines containing the border, the ULA starts the first line of the screen proper.

A line synchronising pulse is followed by a bit of border, and then the action starts. Switching over control of the low 16K of RAM address lines to itself via the multiplexing circuit mentioned, the ULA reads in the first byte stored in the display file, and also the first byte in the attributes file. Then it returns control of the RAM to the CPU. A problem that could arise here is the CPU also trying to read or write to the same block of memory when it cannot control the address lines: it is impossible for it to fetch any meaningful data if it attempts to read from that block simultaneously with the ULA. To prevent this, the address bus is monitored by the ULA and, if it senses that a clash is about to occur, it simply freezes the CPU by stopping the clock pulse which it is sending, until it has relinquished control of the address lines.

The information relating to the first eight pixels of the screen is now in the ULA chip, each bit of the byte from the display file representing 1 for ink or 0 for paper. The attributes file controls the colour information as follows — bits 0, 1 and 2 determine the colour of the ink pixels, and bits 3, 4 and 5 the paper. If bit 6 is set, this signifies BRIGHT and the whole of the eight pixels are 'sat up'; that is, the luminance of both colours is increased. Bit 7 is the flash control: if set to 0 then the pixels are unaffected; if 1, then the output of a low frequency oscillator circuit in the ULA determines if the paper and ink values are to be swopped over before coding.

The first eight pixels have been conveyed to the screen, and the process is repeated another 31 times until the entire top row has been sent. Then the border colour is reinstated for the remainder of the line. The ULA proceeds to reiterate this process for the rest of the screen. However, the next line of pixels is not stored immediately after the first. I will show how the memory map of the screen is arranged in a moment. Each line continues to be fetched and transmitted until the bottom of the screen is reached, when more lines of border are produced, until it is time for the next frame pulse, and off we go again. So the ULA happily transfers the contents of the display files to the outside world. A chip (LM 899) helps code the colour information before the signal is sent to the modulator. From there a broadcasting lookalike is sent to the television set. When its access to memory in the region 4000h to 7FFFh would upset the work of the CPU, then it stops the supply of clock pulses until the danger is over.

Now to move on to the actual arrangement of the display files. At first glance, the method used seems remarkably convoluted. However, it is important that the ULA should be able to scan this area with the minimum of decoding work. It is, therefore, understandable that the pattern used makes more sense when we study the addresses in binary.

The screen is divided into three sections, each of which use 2K of memory to store the pixel data. These sections occupy 4000h to 47FFh for the top of the screen, 4800 to 4FFFh for the middle, and 5000h to 57FFh for the bottom. Each section contains eight rows of characters. The top line of the first of these rows of characters is held by the first 32 bytes of each section. The next seven blocks of 32 bytes each hold the same data for the remaining seven text rows of the section. Subsequently, the lines of pixels for the rows of characters are stored in the same manner — that is, all the second lines, then all the third lines, and so on, for the eight lines of pixels that make up a row of character spaces.

This is complicated to describe in words and so it is best illustrated by a simple program. Enter Program 10.1 and run it. It will ask for two values to use for its demonstration. I suggest that you try 255 as the first one, and any eight-bit number for the second. It will then fill the entire display file sequentially with the first byte, and the three sections of the screen will in turn be filled with ink dots. Notice the 'venetian blind' effect as the character rows are gradually built up.

When the screen is full, something slightly more dramatic will occur — the program pokes the second value into the attributes file which sits immediately above D-file 1. This is effected much more quickly due to the smaller size of the file. I have said that things can sometimes be easier in binary, and here is a good example of this. If we write a 16-bit number to represent the screen locations, and use symbols to denote which bits tally with which parameters, it looks like this:

010SSLLLRRRBBBBB

The letters represent the following. S is the section number with 0 at the top and 2 (10 binary) at the botom. L is the number of the line of pixels within the row of characters (a three-bit number can stand for one of eight lines). R is the row number, and B indicates the position of the byte in the row, the number of the column. The address splits neatly into two bytes: you can increment the high byte in order to locate the subsequent seven lines of pixels for each character space.

Program 10.1: Screen Layout

```
  10 REM       Screen layout
  11 REM       examination
  12 REM
  20 REM        Set up
  21 REM
  30 INK 0: PAPER 7: BORDER 7: FLASH 0:
BRIGHT 0: CLS
  40 REM
  41 REM        Get values
```

```
  42 REM
  50 INPUT "Enter byte (0>255) ";byte
  60 IF byte<0 OR byte>255 THEN  GO TO 5
0
  70 INPUT "Enter color (0>255) ";color
  80 IF color<0 OR color>255 THEN  GO TO
 70
  90 REM
  91 REM       Alter Screen
  92 REM
 100 FOR x=16384 TO 22527: POKE x,byte:
NEXT x
 110 REM
 111 REM       Alter Colours
 112 REM
 120 FOR x=22528 TO 23295: POKE x,color:
 NEXT x
 130 REM
 131 REM       Display Values
 132 REM
 140 PRINT  AT 2,4;"Byte = ";byte; AT 6,
4;"Color = ";color; AT 10,4;"Press a key
 to alter"
 150 REM
 151 REM       Wait for keypress
 152 REM
 160 IF  INKEY$ ="" THEN  GO TO 160
 170 GO TO 10
```

The Spectrum uses a quite straightforward method of printing characters on the screen. It has stored in ROM a table of shapes which, when POKEd to the screen, will form the letters. We saw earlier that the routine for printing to the screen can be called at 10h, and that this jack-of-all-trades routine also handles other channels, assuming that the correct one is selected. It has several other assets — it can move the print position when the screen is selected; it prompts you with the query 'SCROLL?' if there is no room left to print in; it sets the colours; it handles the special printing modes such as OVER; and it will even expand the BASIC tokens when you send them to be PRINTed.

However, at the heart of the routine exists code which looks up the current printing position. The code uses the ASCII character code which was passed to it in the A register when the routine was called, to find the corresponding dot pattern in the ROM. It copies the first byte of that pattern on to the screen, increments its pointer to the ROM table, and also increments the high byte of the register that holds the screen address, so that it points to the next line of pixels down the screen. The next byte of dot pattern is then copied into D-file and the process repeated until all eight bytes are in the correct parts of screen memory.

Now that the shape is in the display file, it will automatically be sent to the screen by the ULA. The output routine also sets the attributes' byte for the character space to whatever the current colours are, and updates the print position. This is then stored in two ways — as line and column numbers as used by the PRINT AT command, and as an absolute address for the first byte of the character space. So the 'real' screen address need be calculated less frequently than it would be if only the coordinates were stored.

Another system variable that is used by this routine is CHARS, which holds an address which is 256 less than the beginning of the character shape table. For most characters you need only multiply their ASCII code by eight and add it to CHARS. This enables you to find the address of the block of eight bytes which holds the shape of a particular character. The normal value of CHARS is 3C00h as the table is located from 3D00h to 3FFFh in the main ROM, but as CHARS is held in RAM you can make the operating system use your own character set which is located elsewhere. If you want to create confusion, try POKE 23606,8: for further nonsense, POKE 23607,0.

Much the same method is used to store another source of shapes, the user-definable graphics, which are kept at the top of the memory. When it is asked to print a graphic character, the output routine fetches the address of the table from UDGs and obtains the shape from there. Before any new shapes have been defined, the RAM holds a copy of part of the ROM table, so we obtain normal capital letters. In addition, there is a set of 'predefined graphics', in the form of quarter-character-sized blocks, which are obtained by calculation. If you were to study their shapes in conjunction with their character codes, translated into *binary*, you would probably guess how this is done. (See your Manual for the codes.) The four quarters can be seen as the bits of a four-bit binary number, with the prefix of 1000 binary. The shape is therefore 'made up' by a clever piece of machine code.

Manipulating the screen from machine code

In order to manipulate the screen from machine code, you can use one of two approaches. You can either write your own purpose-designed printing routines, which is not as difficult as it sounds, or use the Spectrum's own software. If you are writing a program that does not require outstandingly fast and smooth graphics, then the available ROM routines are more than adequate. You can send all the BASIC control codes to the output routine and even print to the bottom two lines of the

screen if you change the value of DF SZ, which holds the number of lines reserved for error messages. (Don't neglect to restore it before returning to BASIC or there will not be anywhere for the computer to print OK!) For those of you intent on bettering the 'Buck Rogers and the Planet of Zoom' arcade game with your own routines, then good luck! — and here are two small tips.

The first relates to the border. Many commercial programs for the Spectrum are marred by black flashes in the border every time the colour is changed, yet preventing this is simple. Before changing the three low bits of port FEh to the new colour, use a HALT command so that the CPU waits for an interrupt before continuing. When it recommences, this will coincide with a frame pulse, so the change of border colour will not occur during active picture time.

The second tip is Program 10.2, a machine code listing which calculates screen addresses. I have only provided the op codes and not a BASIC program to load it, because it will need to be incorporated into a machine code program as a subroutine. It can be located anywhere in free memory. To use it, load the E register with the horizontal column number as in the AT command, and load the D register with the screen pixel number, which starts at the top with line zero and ends on 192 dec. Then call the routine. When it returns, the HL register will contain the address of the required byte in the display file, and the BC register will hold the corresponding byte of the attributes file. You may wish to shorten the program to deal only with the first byte of each character; and there is no need to use the same registers either, if these changes suit your purposes.

I have kept the program simple so as to demonstrate the method more effectively: you should find the principle useful.

Program 10.2: Machine code routine to calculate screen addresses

```
Address        Hex code       Assembler

SCALC          F5             PUSH AF
               7A             LD A,D
               E6F8           AND F8h
               0F             RRCA
               0F             RRCA
               0F             RRCA
               6F             LD L,A
               E618           AND 18h
               F640           OR 40h
               67             LD H,A
               7A             LD A,D
               E607           AND 07h
```

```
84      ADD A,H
67      LD A,L
E607    AND 07h
0F      RRCA
0F      RRCA
0F      RRCA
83      ADD A,E
6F      LD L,A
7A      LD A,D
E6C0    AND C0h
07      RLCA
07      RLCA
F658    OR 58h
47      LD B,A
4D      LD C,L
F1      POP AF
C9      RET
```

On entry	D holds the pixel line number
	E holds the column number
Routine returns	HL holding the display file address
	BC holding the attributes file address

Writing high quality graphics routines is no mean task, but you are starting off with some pretty good raw materials. You will be pleasantly surprised by some of the effects that can be achieved with just a small dose of machine code, particularly if applied to the attributes file.

CHAPTER 11 Sound

'If only it could talk', you mutter as you stare, grasping for comprehension, at an error report at the bottom of the screen. In fact you may already have heard a talking Spectrum, as they can be persuaded to speak, albeit somewhat crudely, by software. The hardware for making sound consists of a simple BEEP facility, the clicking that occurs when the keyboard is in use comes from these beep circuits, and the signals sent to the MIC sockets are from the same source.

How the beep port works

The simple beep circuit is contained mainly in the ULA. The gates responsible for decoding the CPU's attempt to write to or read from port FEh, connect to a pin of the ULA which has some external components attached. If a write operation to port FEh sets bit 4, then a high enough voltage comes out of the ULA pin to activate the small built-in speaker. If bit 3 is set, then the voltage generated will be insufficient to drive the speaker, but it will still be present at the MIC socket for recording on cassette. The sharp-edged square wave that occurs when a voltage is switched between high and low, is somewhat modified by the use of a capacitor which rounds off the edges.

The same ULA pin is also connected to the EAR socket — by reading bit 6 of port FEh, either a digital 1 or a 0 is returned, depending on the incoming voltage. Inside the ULA, a special kind of gate called a Schmitt trigger is used to approximate the signal into a 1 or 0. If you slowly raise the voltage on the input of this gate, its output will remain low until the input is around 2.5 volts, when it 'triggers' and the output goes high.

The Spectrum's system software creates a beep in the following manner. Bit 4 of port FEh is set, and a looping routine makes the CPU wait for a length of time which relates to the frequency of the beep required. When this time has elapsed, bit 4 of the port is reset, and the loop delay is repeated. One cycle of sound has now been sent to the speaker and the whole process recurs until cycles sufficient to achieve the required sound duration have been generated.

There are two points to note about his routine. If we double the frequency, that is, the number of cycles per second, by halving the delay period, then the duration of the beep will halve, unless we double the number of cycles that are generated. The beep routine calculates how many cycles of any particular frequency are needed, in order to make the sound last for the required duration. In doing this, it makes extensive use of the floating point number routines. The other factor is the three low bits of port FEh which control the border colour — if you write to the port indiscriminately, without setting the bits to the right value, then the border will change. The beep routine looks up the current border colour in a system variable called BORDCR: bits 3, 4 and 5 indicate which colour is active. These bits are moved to the low three of the accumulator, so that the border remains constant when an OUT (FE),A instruction is executed.

Table 11.1: Machine Code for Sound Effect

Address	Hex Code	Assembler
RAMTOP+1	3A485C	LD A,(5C48h)
	1F	RRA
	1F	RRA
	1F	RRA
	E607	AND 07h
	0EFF	LD C,FFh
	2600	LD H,00h
	44	LD B,H
	CBE7	SET 4,A
	D3FE	OUT (FEh),A
	10FE	DJNZ −2
	44	LD B,H
	CBA7	RES 4,A
	D3FE	OUT (FEh),A
	10FE	DJNZ −2
	CBE7	SET 4,A
	D3FE	OUT (FEh),A
	10FE	DJNZ −2
	CBA7	RES 4,A
	D3FE	OUT (FEh),A
	10FE	DJNZ −2
	24	INC H
	0D	DEC C
	20E2	JR NZ,−30
	C9	RET

Using the beep port from machine code

Using the beep port from machine code threatens all the above pitfalls for the user, with one extra thrown in for good measure, but a valuable advantage is the speed with which the parameters can be changed. So, for example, alternate cycles of different duration can be produced, or the shape of the cycles altered, with the low part lasting for less time than the high part. The extra pitfall I mentioned is caused by the video generating circuits that will stop the CPU when both it and the ULA are trying to use the low 16K of RAM. Any machine code that is located here will run in a 'lumpy' manner, as it gets interrupted quite often. Although it is still wise to turn off the interrupts, as the ROM routine does, these cause much less 'warbling' than does the constant halting of the program, if it is in the low 16K of RAM. If you want to create pure tones with the beep port, then you must stick with the ROM code, readily available at 3B5h in the ROM, or use the expanded machine.

An example of what can be attained in a few bytes is given in Program 11.1. The BASIC program can simply be typed in, saved and run. The machine code listing explains what is happening and allows you to incorporate the effect in your programs. Note how the individual tones blend into one another, an effect that you are unlikely to achieve using BASIC alone.

Program 11.1: Beep Port Routine

```
  10 REM        Beep port routine
  11 REM
  20 REM        Lower RAMTOP
  21 REM
  30 RESTORE : LET x=( PEEK 23730+256* P
EEK 23731)-43: CLEAR x
  40 REM
  41 REM        Poke Machine code
  42 REM        into memory
  43 REM
  50 LET x=( PEEK 23730+256* PEEK 23731)
+1
  60 FOR y=x TO x+42: READ z: POKE y,z:
NEXT y
  70 REM
  71 REM        Print USR address
  72 REM
  80 PRINT  AT 1,2;"The routine is calle
d by the"; TAB 7;"function USR ";x
```

```
  90 REM
  91 REM        Example
  92 REM
 100 PRINT  AT 12,10;"Press any key"
 110 IF  INKEY$ ="" THEN  GO TO 110
 120 LET a= USR x
 130 GO TO 110
 200 REM
 201 REM        Machine code data
 202 REM
 210 DATA 58,72,92,31,31,31,230,7,14,255
 220 DATA 38,0,68,203,231,211,254,16,254
,68
 230 DATA 203,167,211,254,16,254,203,231
,211,254
 240 DATA 16,254,203,167,211,254,16,254,
36,13
 250 DATA 32,226,201
```

Tape routines

Let's now take a brief look at the way that cassette save and load functions are acomplished by the system software, using routines in the ROM from 4C2h to 9F3h. If you want to devise your own routines, perhaps to achieve a faster method of storage, then the ULA's policy of halting the CPU means that the best place for them would be the extra RAM area of an expanded machine.

The method of saving goes along the following lines. The file name (and the information as to which type of file it is) is assembled in memory as a string of bytes, to be used later as a header. Then the 'start tape' message is displayed. The machine then tests the keyboard for a press and, when one occurs, first the header, then the data itself, is formatted and sent to the MIC socket for recording. This format consists of, firstly, a leader of tone with a frequency just over 800 cycles, then one cycle of 2040 hertz. Now single bits are transmitted as a cycle of either 2040 hertz to represent a zero, or 1020 hertz for a one.

The first eight bits come from the A register, which holds a flag that is set if the data to follow is header information. Next, each byte to be stored is sent, one bit at a time, as tones to the EAR socket. After each byte, the keyboard is tested to ascertain whether BREAK is being pressed, and the routine aborts if it is. At the end of the data, a final byte is written to tape (the length of the block was held in the DE register). This byte, called the parity byte, is created by XORing each byte of data together as a checksum.

Loading information back from tape consists of the above process in reserve. The CPU monitors bit 6 of port FEh and, when it detects the leader tone, it waits for the first cycle of 2040 hertz which signals the start of data. It then reads in the bits from tape, and forms them back into bytes. The first byte informs the CPU whether it is reading in a header or data, and the software reacts accordingly. When a suitable header is found, the information as to where the data is to be stored (PROG if it is a BASIC program) and how long it is, is read from the header information. The file name is printed to the screen and the next data file on tape is loaded into memory.

At the end of the data, the parity byte from tape is compared with another parity byte which the loading routine has been building up from the incoming data. If there is a discrepancy, the routine stops with an error report. During the time that bits are read in successfully, they are also reflected in the behaviour of the border, which switches between yellow and cyan as the EAR voltage alters. You may like to set up two adjacent areas of RAM as blocks of 0s and FFhs (255), and save them with a code save. You will then see the width of the stripes double while they roll through the border as the FFhs are transmitted.

For the musically inclined, here are a few observations. The beep facility allows you to produce notes that relate to the musical scale — using SOUND and machine code will necessitate the calculation of your own values. Middle C is a frequency of 440 hertz; if you double the frequency of a note, the resultant sound is one octave higher. The 11 semitones in between can be calculated by knowing that the frequency of C multiplied by the 12th root of 2 will produce the frequency of C sharp, C sharp times the 12th root of 2 gives D, and so on. The 12th root of 2 is roughly 1.059463. Happy composing!

CHAPTER 12
More Letters to the Line

The number of characters that will fit on to one line of the Spectrum's screen display can be something of a disability. If you wish to use your computer for more serious applications such as word processing, then 32 columns is quite inadequate — even just a few extra characters on each line would improve the usefulness of the computer. To squeeze 42 characters on to a line is not particularly difficult if you resort to machine code, but then how do you interface the machine code with BASIC? The following program is designed to be incorporated into your own BASIC application. It demonstrates some simple techniques for screen manipulation and also provides an example of how you can search the BASIC variables area for a particular variable.

Firstly, let's look at how to use the program. The machine code is stored at the top of the memory. It is protected from being over-written by the use of a CLEAR statement which lowers RAMTOP in order to leave room. The code is 953 bytes in length, of which only 283 bytes are the program proper. The remainder constitutes a look-up table of the character shapes so you need not worry about entering them accurately at first. In order to print a string to the screen, you use the statement LET a$="Whatever you want to print" followed by RAND USR 64421 if you have a 48K machine, or RAND USR 31653 for 16K Spectrum owners. There is no need to make sure that a$ is the last variable declared. For example, you may use the routine as part of a FOR . . . NEXT loop in the form:

```
LET a$="Small print":FOR X=0 TO 23:RAND USR 64421:NEXT X
```

The routine will remember its last print position in the same manner as BASIC, and you can move the print position with an AT type of command. To do this you embed within the string CHR$(22) followed by CHR$(line number) and CHR$(column number). For example

```
LET a$=CHR$(22)+CHR$(5)+CHR$(10)+"Please wait": RAND
USR 64421
```

will print the message at line 5, column 10, regardless of the previous print position. Any other CHR$ below 32 (dec) will force the text to start a new line. A CHR$ above 127 will cause the premature end of printing. The routine makes no attempt to alter the attributes file — the letters overlap the attribute boundaries.

If you have a 16K Spectrum there will not be enough room for Program 12.1. Also, the op codes are slightly different as the routine is not position independent — in other words, it contains absolute addresses which must be changed if the program is to be located elsewhere. Therefore Program 12.2 should be used, which only loads the first section of code. When successfully loaded and saved, RUN the program and then load the monitor, Program 6.1, or any other routine capble of poking decimal data into memory. You must now enter the character shape data into addresses 31928 upwards, using the data from the 48K version (Program 12.1) from line 500 onwards. You can now test the machine code and, when you are satisfied that the characters are the correct shape, save the routine on tape.

The installation of the routine is much easier if you have a 48K Spectrum. Enter Program 12.1 — the large number of data statements may appear daunting, but, once the first section (up to line 450) is correct, then the program will run without crashing, if you delete the STOP statement at the end of line 30. The errors in the character table will then be easier to trace by printing the entire character set to the screen. When you have a correct program, save it carefully on tape. You can now save the code separately to load back into your own programs after the necessary CLEAR command.

Program 12.1: 48K version

```
   5 REM      42 COLUMN SCREEN
   6 REM
   7 REM          48K VERSION
   8 REM
  10 CLEAR 64412: CLS : PRINT "Checking
data; please wait."
  20 RESTORE : LET total=0: FOR x=64413
TO 64695: READ byte: LET total=total+byt
e: POKE x,byte: NEXT x: REM  IF total <>
 32570 THEN  PRINT "Error in program dat
a.": STOP
  30 LET total=0: FOR x=64696 TO 65367:
READ byte: LET total=total+byte: POKE x,
byte: NEXT x: REM  IF total <> 61796 THE
N  PRINT "Error in character data.": STO
P
  40 CLS : LET a$="42 column screen rout
```

```
ine now installed    above RAMTOP.To sav
e  to tape use:-"+ CHR$ 22+ CHR$ 3+ CHR$
 6+"SAVE "42col" CODE 64413,953"
  50 RANDOMIZE  USR 64421
  60 LET a$= CHR$ 0+ CHR$ 0+"To use LET
a$= string you wish to print"+ CHR$ 0+"t
hen RAND USR 64421"
  70 RANDOMIZE  USR 64421
  80 LET a$= CHR$ 0+ CHR$ 0+"CHR$ 22+ CH
R$ 1+ CHR$ c will set the printposition
to line 1 and column c; CHR$ 0   will fo
rce a newline"
  90 RANDOMIZE  USR 64421: STOP
  97 REM
  98 REM      MAIN CODE
  99 REM
 100 DATA 0,0,0,0,0,0,0,0
 110 DATA 42,75,92,126,71,254,128,200
 120 DATA 35,230,224,254,224,32,6,17
 130 DATA 18,0,25,24,238,254,160,32
 140 DATA 11,203,126,35,40,251,17,5
 150 DATA 0,25,24,223,254,96,40,246
 160 DATA 94,35,86,35,120,254,65,40
 170 DATA 3,25,24,207,34,159,251,25
 180 DATA 34,161,251,42,159,251,84,93
 190 DATA 237,75,161,251,167,237,66,208
 200 DATA 235,126,35,34,159,251,254,128
 210 DATA 208,254,22,32,24,126,254,23
 220 DATA 48,3,50,158,251,35,126,35
 230 DATA 34,159,251,203,39,71,128,128
 240 DATA 50,157,251,24,206,254,32,218
 250 DATA 165,252,111,38,0,203,37,203
 260 DATA 20,203,37,203,20,203,37,203
 270 DATA 20,22,0,95,167,237,82,17
 280 DATA 216,251,25,229,221,225,33,163
 290 DATA 251,58,158,251,95,230,24,246
 300 DATA 64,87,123,230,7,15,15,15
 310 DATA 95,58,157,251,203,63,203,63
 320 DATA 203,63,131,95,14,255,58,157
 330 DATA 251,230,7,71,62,3,40,10
 340 DATA 55,31,203,25,31,203,25,5
 350 DATA 16,247,119,121,50,164,251,58
 360 DATA 157,251,230,7,14,0,71,221
 370 DATA 126,0,40,9,31,203,25,31
```

```
380 DATA 203,25,5,16,247,71,26,166
390 DATA 176,18,19,35,26,166,177,18
400 DATA 27,43,221,35,20,122,47,230
410 DATA 7,32,212,58,157,251,198,6
420 DATA 50,157,251,254,250,218,224,251
430 DATA 175,50,157,251,58,158,251,60
440 DATA 254,24,202,224,251,50,158,251
450 DATA 195,224,251
497 REM
498 REM   CHARACTER SHAPE TABLE
499 REM
500 DATA 0,0,0,0,0,0,0
510 DATA 64,64,64,64,0,64,0
520 DATA 80,80,0,0,0,0,0
530 DATA 0,80,248,80,248,80,0
540 DATA 32,120,160,112,40,240,32
550 DATA 64,168,80,32,80,168,16
560 DATA 64,160,72,176,144,104,0
570 DATA 64,64,0,0,0,0,0
580 DATA 64,128,128,128,128,64,0
590 DATA 128,64,64,64,64,128,0
600 DATA 0,168,112,248,112,248,0
610 DATA 0,32,32,248,32,32,0
620 DATA 0,0,0,0,0,192,64
630 DATA 0,0,0,240,0,0,0
640 DATA 0,0,0,0,0,192,0
650 DATA 32,32,64,64,128,128,0
660 DATA 96,144,176,208,144,96,0
670 DATA 64,192,64,64,64,64,0
680 DATA 96,144,16,32,64,240,0
690 DATA 96,144,32,16,144,96,0
700 DATA 32,96,160,240,32,32,0
710 DATA 224,128,192,32,32,192,0
720 DATA 112,128,224,144,144,96,0
730 DATA 240,16,32,32,64,64,0
740 DATA 96,144,96,144,144,96,0
750 DATA 112,144,144,112,16,16,0
760 DATA 0,64,0,0,64,0,0
770 DATA 0,64,0,0,64,128,0
780 DATA 0,32,64,128,64,32,0
790 DATA 0,0,224,0,224,0,0
800 DATA 0,128,64,32,64,128,0
810 DATA 64,160,32,64,0,64,0
820 DATA 112,136,8,104,168,112,0
```

```
 830 DATA 96,144,144,240,144,144,0
 840 DATA 224,144,224,144,144,224,0
 850 DATA 96,144,128,128,144,96,0
 860 DATA 224,144,144,144,144,224,0
 870 DATA 240,128,224,128,128,240,0
 880 DATA 240,128,224,128,128,128,0
 890 DATA 96,144,128,176,144,112,0
 900 DATA 144,144,240,144,144,144,0
 910 DATA 224,64,64,64,64,224,0
 920 DATA 240,32,32,32,160,64,0
 930 DATA 144,160,192,192,160,144,0
 940 DATA 128,128,128,128,128,224,0
 950 DATA 136,216,184,136,136,136,0
 960 DATA 144,144,208,176,144,144,0
 970 DATA 96,144,144,144,144,96,0
 980 DATA 224,144,144,224,128,128,0
 990 DATA 112,136,136,168,152,120,0
1000 DATA 224,144,144,224,160,144,0
1010 DATA 112,128,96,16,16,224,0
1020 DATA 240,64,64,64,64,64,0
1030 DATA 144,144,144,144,144,96,0
1040 DATA 136,136,136,136,80,32,0
1050 DATA 136,136,136,168,168,80,0
1060 DATA 136,80,32,32,80,136,0
1070 DATA 136,136,80,32,32,32,0
1080 DATA 240,16,32,64,128,240,0
1090 DATA 224,128,128,128,128,224,0
1100 DATA 128,64,64,32,32,16,0
1110 DATA 224,32,32,32,32,224,0
1120 DATA 32,112,168,32,32,32,0
1130 DATA 0,0,0,0,0,0,252
1140 DATA 96,144,128,192,128,240,0
1150 DATA 0,96,16,112,144,112,0
1160 DATA 128,128,224,144,144,224,0
1170 DATA 0,96,128,128,128,96,0
1180 DATA 16,16,112,144,144,96,0
1190 DATA 0,96,144,224,128,112,0
1200 DATA 96,128,192,128,128,128,0
1210 DATA 0,96,144,144,112,16,224
1220 DATA 128,128,192,160,160,160,0
1230 DATA 128,0,128,128,128,192,0
1240 DATA 32,0,32,32,32,160,64
1250 DATA 128,128,160,192,160,144,0
1260 DATA 128,128,128,128,128,64,0
```

```
1270 DATA 0,80,168,168,136,136,0
1280 DATA 0,160,208,144,144,144,0
1290 DATA 0,96,144,144,144,96,0
1300 DATA 0,224,144,144,224,128,128
1310 DATA 0,96,160,160,96,32,48
1320 DATA 0,160,192,128,128,128,0
1330 DATA 0,96,128,64,32,192,0
1340 DATA 128,192,128,128,128,96,0
1360 DATA 0,144,144,144,144,240,0
1370 DATA 0,136,136,136,80,32,0
1380 DATA 0,136,136,136,168,112,0
1390 DATA 0,144,144,96,144,144,0
1400 DATA 0,144,144,144,112,16,224
1410 DATA 0,240,32,64,128,240,0
1420 DATA 32,64,64,128,64,64,32
1430 DATA 64,64,64,64,64,64,64
1440 DATA 128,64,64,32,64,64,128
1450 DATA 80,80,0,0,0,0,0
1460 DATA 112,136,168,200,168,136,112
```

Program 12.2: 16K version

```
   5 REM      42 COLUMN SCREEN
   6 REM
   7 REM         16K VERSION
   8 REM
  10 CLEAR 31644: CLS : PRINT "Checking
data; please wait."
  20 RESTORE : LET total=0: FOR x=31645
TO 31927: READ byte: LET total=total+byt
e: POKE x,byte: NEXT x: IF total <> 2982
4 THEN  PRINT "Error in program data.":
STOP
  30 PRINT "The program code is now loca
ted at 31645, length 283"'"When you have
 loaded the"'"character data the routine
 runs"'"from 31653": STOP
  97 REM
  98 REM      MAIN CODE
  99 REM
 100 DATA 0,0,0,0,0,0,0,0
 110 DATA 42,75,92,126,71,254,128,200
 120 DATA 35,230,224,254,224,32,6,17
 130 DATA 18,0,25,24,238,254,160,32
```

```
140 DATA 11,203,126,35,40,251,17,5
150 DATA 0,25,24,223,254,96,40,246
160 DATA 94,35,86,35,120,254,65,40
170 DATA 3,25,24,207,34,159,123,25
180 DATA 34,161,123,42,159,123,84,93
190 DATA 237,75,161,123,167,237,66,208
200 DATA 235,126,35,34,159,123,254,128
210 DATA 208,254,22,32,24,126,254,23
220 DATA 48,3,50,158,123,35,126,35
230 DATA 34,159,123,203,39,71,128,128
240 DATA 50,157,123,24,206,254,32,218
250 DATA 165,124,111,38,0,203,37,203
260 DATA 20,203,37,203,20,203,37,203
270 DATA 20,22,0,95,167,237,82,17
280 DATA 216,123,25,229,221,225,33,163
290 DATA 123,58,158,123,95,230,24,246
300 DATA 64,87,123,230,7,15,15,15
310 DATA 95,58,157,123,203,63,203,63
320 DATA 203,63,131,95,14,255,58,157
330 DATA 123,230,7,71,62,3,40,10
340 DATA 55,31,203,25,31,203,25,5
350 DATA 16,247,119,121,50,164,123,58
360 DATA 157,123,230,7,14,0,71,221
370 DATA 126,0,40,9,31,203,25,31
380 DATA 203,25,5,16,247,71,26,166
390 DATA 176,18,19,35,26,166,177,18
400 DATA 27,43,221,35,20,122,47,230
410 DATA 7,32,212,58,157,123,198,6
420 DATA 50,157,123,254,250,218,224,123
430 DATA 175,50,157,123,58,158,123,60
440 DATA 254,24,202,224,123,50,158,123
450 DATA 195,224,123
```

Now let's look at how the program works. I have provided explanatory comments with the assembler listing (at the end of this chapter) which may prove helpful. The routine can be divided into three sections, the last one being the character shape table mentioned above. There are five variables used by the machine code for its own purposes: XPOS and YPOS are single bytes that store the current column and line positions on the screen. XPOS holds the pixel location so it may have a value up to 248 (the characters have a width of six pixels each). YPOS holds a line number in the range 0 to 23: DATA, LENTH and BUFFR are all two bytes in size. Please forgive my spelling of LENTH — the convention for labels only allows five letters to be used on some assembler programs.

The first task for the routine is to discover the size and position in memory of a$. We can find both the beginning and the end of the BASIC variables area with the system variables VARS and ELINE. Each time we define a variable in BASIC the interpreter looks through the variables, starting from VARS, to check if that variable is already in existence. If it is, the interpreter deletes it and closes the gap left behind before placing the new variable at the end of the VARS area.

But, I hear you protest, what happens if I use a statement such as LET A=A+1? To allow for this, the interpreter will always work out what a variable is supposed to be before trying to store it. We can see from the above that we are going to find the most recently defined variables at the end of the VARS area, but unfortunately we have no way of searching the area backwards to save time. If you study Chapter 24 of your Spectrum Manual, you will see that each type of variable can be identified by its first byte, which holds the first letter of its name in the low five bits and an identifier in the high three bits. It is therefore possible to sort through for the type we are looking for.

First of all, the machine code routine finds the start of the VARS area from the system variable, and then it fetches what is stored there. If it finds an 80 hex it returns to BASIC, as this indicates the end of the variables. Throughout this section of the program the routine uses the HL register to keep track of where it has reached in the area: this technique is known as 'pointing' HL at the data. If the byte fetched has the pattern 111 in the high three bits, then the variable is a FOR . . . NEXT control variable which is 18 bytes long and the pointer is added to, so that it points to the next variable.

If the pattern is 101, then the number of the variable has a name or more than one letter. In this case, the program searches through the following bytes of the name. The BASIC interpreter sets bit 7 of the last letter of the name to one, so we can use this to find the end of the name and then add the length of a numerical variable (five bytes) to the pointer. A number variable with a name of only one letter has the pattern 011, so when the program encounters this it simply adds five to the pointer.

The remaining types of variable all have their size stored as two bytes, Z80 fashion (that is, least significant byte first) as bytes 2 and 3, so the program fetches this information. It then tests the name byte to see if it is 41 hex, which is the code for a single dimension a$. If it is, then we have located the required variable. Its starting address is stored in DATA, the end is calculated by adding its length to the pointer, and then it is stored in LENTH. If a$ was not found, then the next variable is located in the same manner as above with the whole of the procedure repeated until either a match is found or the 80h market is encountered.

So we have finally found a$ (unless we forgot to declare it)! The above process is very similar to that which BASIC has to go through each time it deals with a variable, so it's just as well machine code is so fast.

Now we can get on with printing the string. The second part of the program fetches each byte of the variable in turn, using its own variable DATA to keep track of its progress. Each time, before it fetches a character for printing, it checks DATA against LENTH to see if it has reached the end of the string. If it encounters any of the control codes it reacts accordingly, 16h prompting it to fetch the next two bytes of the string and place them in XPOS and YPOS. For each character, the program must find the shape data for that letter in the character table — this is quite simple because each shape takes up seven bytes, so we multiply the character code by seven and then add a base address (the start of the table minus 32*7 for the unused ASCII codes). We must then calculate the screen address from YPOS and the high five bits of XPOS, using a very similar technique to that demonstrated earlier in the book in the SCALC program (Program 10.2).

Now for the tricky part. We have the shape byte, the screen position, and, from the low three bits of XPOS, we know how much offset needs to be applied to the shape before it is placed on the screen. What we must first do is make up a 'mask' that can be used to ensure that we do not wipe any pixels that should remain undisturbed. We achieve this by making up a 16-bit register containing 0000001111111111 binary and then rotating it right circular for as many bits as the offset demands. The shape is also placed in a 16-bit register and rotated in the same manner. It is perhaps best to use an example to show what happens next. Assume that we wish to place the six bits, 001110, fetched from the high six bits of the shape byte in the table, with a six pixel offset on the screen. The mask would be:

1111110000001111 binary

and the shape would be rotated to:

0000000011100000 binary

Obviously we must deal with two bytes of the screen. Say that they contained:

1111111111111111 binary

If we AND the screen bytes with the mask byte, the result would be a six-bit 'hole':

1111110000001111 binary

Now by ORing the shape byte we get:

1111110011101111 binary

which is the desired result! The program has to work one byte at a time, but the end result is the same. So each line of the shape is POKEd to the screen until all seven lines of the character have been printed. When the letter is complete, the program advances the print positions and then fetches the next character of the string for printing, unless it has reached the end of the string.

I hope that the above description combined with a study of the assembler listing will give you a good understanding of how the program operates. You may wish to develop the sytem — it is quite possible to reduce the width of the letters to five bytes at the expense of legibility. A better proposition (and one which can be made to work!) is to use proportional spacing: a letter I need only take up three pixels so, if you store the width of each character in the table along with each shape, your print routine can act accordingly. The program also indicates how smooth one-pixel-at-a-time movement can be achieved for graphic shapes.

Assembler listing for 42 column screen machine code program

Note: The op codes are in hexadecimal and are for the 48K version.

Code	Label	Instruction		Comments
—		ORG	FB9D	Start program from this address
00	XPOS	DEFB	00	Set up eight bytes for use by
00	YPOS	DEFB	00	the program as variables
0000	DATA	DEFW	0000	
0000	LENTH	DEFW	0000	
0000	BUFFR	DEFW	0000	
2A4B5C	FIND$	LD	HL,(5C4B)	Set HL to the start of the BASIC variables area
7E	GETBY	LD	A,(HL)	Load A with first byte of BASIC variable
47		LD	B,A	Save it in B
FE80		CP	80	If it is the marker at the end
C8		RET	Z	of the VARS area return to BASIC
23		INC	HL	Point HL to second byte

Code	Label	Instruction		Comments
E6E0		AND	E0	Mask of letter data from A
FEE0		CP	E0	If not a control variable then
2006		JR	NZ,NUM?	jump to NUM?
111200		LD	DE,0012	Add length of control variable
19		ADD	HL,DE	to pointer in HL
18EE		JR	GETBY	Loop back to GETBY
FEA0	NUM?	CP	A0	If not a variable with a long
200B		JR	NZ,SNUM?	name jump to SNUM?
CB7E	TEST	BIT	7,(HL)	Find the last letter of the name,
23		INC	HL	set HL to point to next byte
28FB		JR	Z,TEST	
110500	ADD5	LD	DE,0005	Add the length of a numerical
19		ADD	HL,DE	variable to HL
18DF		JR	GETBY	Jump to GETBY
FE60	SNUM?	CP	60	If a simple number variable
28F6		JR	Z,ADD5	then jump to ADD5
5E		LD	E,(HL)	Load DE with the length of
23		INC	HL	the BASIC variable and point
56		LD	D,(HL)	HL past length data
23		INC	HL	
78		LD	A,B	Restore first byte of variable to A
FE41		CP	41	If BASIC variable is a single
2803		JR	Z,STORE	dimension a$ jump to STORE
19		ADD	HL,DE	Add the length of variable to HL
18CF		JR	GETBY	Loop back to GETBY
229FFB	STORE	LD	(DATA),HL	Store the location of a$ in DATA
19		ADD	HL,DE	Load HL with address after a$
22A1FB		LD	(LENTH),HL	and store it in LENTH
2A9FFB	PRNTA	LD	HL,(DATA)	Load HL with address of character to be printed
54		LD	D,H	Store HL in DE for quick access
5D		LD	E,L	
ED4BA1FB		LD	BC,(LENTH)	Load BC with address after a$
A7		AND	A	Clear the carry flag
ED42		SBC	HL,BC	Compare current address with
D0		RET	NC	BC and return to BASIC if it is greater than or equal to it
EB		EX	DE,HL	Restore current address to HL
7E		LD	A,(HL)	Load A with character
23		INC	HL	Point HL to next character
229FFB		LD	(DATA),HL	and store it
FE80		CP	80	If the character is equal or
D0		RET	NC	greater than 80 return to BASIC

```
FE16              CP     16           If the character is not 'AT'
2018              JR     NZ,NL?       jump to NL?
7E                LD     A,(HL)       Get line position
FE17              CP     17           If it is out of range (over 23)
3003              JR     NC,TOOHI     jump to TOOHI
329EFB            LD     (YPOS),A     Set new line position
23       TOOHI    INC    HL           Point HL to column position
7E                LD     A,(HL)       Load A with new column
                                      position
23                INC    HL           Point HL to the next character
229FFB            LD     (DATA),HL    and store it
CB27              SLA    A            Multiply the line position in A
47                LD     B,A          by 6
80                ADD    A,B
80                ADD    A,B
329DFB            LD     (XPOS),A     Set column position to new
                                      value
18CE              JR     PRNTA        Loop back to PRNTA
FE20     NL?      CP     20           If the character is less than
DAA5FC            JR     NEWLN        20 jump to NEWLN
6F                LD     L,A          Place A into HL
2600              LD     H,0
CB25              SLA    L            Multiply HL by 7 so that it
CB14              RL     H            can point to the character
                                      table
CB25              SLA    L
CB14              RL     H
CB25              SLA    L
CB14              RL     H
1600              LD     D,0
5F                LD     E,A
A7                AND    A
ED52              SBC    HL,DE
11D8FB            LD     DE,FBD8      Load DE with a base address
19                ADD    HL,DE        for the character table and add
                                      it to HL
E5                PUSH   HL           Transfer HL into IX which
DDE1              POP    IX           now points to the shape table
21A3FB            LD     HL,(BUFFER)  Point HL to the buffer area
3A9EFB            LD     A,(YPOS)     Load A with the screen line
5F                LD     E,A          Save A in E
E618              AND    18           Mask off section number
F640              OR     40           Add 0100000 binary
57                LD     D,A          Store A in D
7B                LD     A,E          Restore line number in A
E607              AND    7            Isolate last 3 bits
0F                RRCA                Multiply A by 8
0F                RRCA
```

```
0F                RRCA
5F                LD     E,A          Put A in E
3A9DFB            LD     A,(XPOS)     Load A with column number
CB3F              SRL    A            Divide A by 8
CB3F              SRL    A
CB3F              SRL    A
83                ADD    A,E          Add E to A and replace in E
5F                LD     E,A          DE now points to first screen
                                      location
0EFF              LD     C,FF         Place 11111111 binary in C
3A9DFB            LD     A,(XPOS)     Load A with the number of
E607              AND    7            pixels displacement required
47                LD     B,A          Store displacement in B
3E03              LD     A,03         Load A with 00000011 binary
280A              JR     Z,TRNFR      If displacement is zero then
                                      jump to TRNFR
37                SCF                 Make carry equal 1
1F       LOOP1    RRA                 Rotate the binary value
CB19              RR     C            0000001111111111 right
1F                RRA                 circular the amount of
CB19              RR     C            the displacement
05                DEC    B
10F7              DJNZ   LOOP1
77       TRNFR    LD     (HL),A       Place the mask pattern in
79                LD     A,C          BUFFR and BUFFR+1
32A4FB            LD     (BUFFR+),A
3A9DFB            LD     A,(XPOS)     Load A with pixel
                                      displacement
E607              AND    7
0E00              LD     C,0          Load C with 00000000 binary
47                LD     B,A          Put displacement in B
DD7E00            LD     A,(IX)       Load A with shape byte
2809              JR     Z,JMP2       If displacement is zero jump to
                                      JMP2
1F       LOOP3    RRA                 Rotate the shape right circular
CB19              RR     C            by the amount of the
1F                RRA                 displacement
CB19              RR     C
05                DEC    B
10F7              DJNZ   LOOP3
47       JMP2     LD     B,A          BC now contains the shape
1A                LD     A,(DE)       Load A with what is already
                                      on the screen
A6                AND    (HL)         Mask out a space and
B0                OR     B            add in the shape
12                LD     (DE),A       Poke new shape to screen
13                INC    DE           Point DE to next screen byte
23                INC    HL           Point HL to BUFFR+1
```

1A		LD	A,(DE)	Repeat printing for second byte
A6		AND	(HL)	
B1		OR	C	
12		LD	(DE),A	
1B		DEC	DE	Restore pointers in HL and DE
2B		DEC	HL	
DD23		INC	IX	Point IX at next shape
14		INC	D	Point DE to next pixel line
7A		LD	A,D	Test D to find out if it has
2F		CPL		its last three bits set
E607		AND	7	
20D4		JR	NZ,LOOP2	If they are not jump to LOOP2
3A9DFB		LD	A,(XPOS)	Load A with column position
C606		ADD	A,06	and add 6
329DFB		LD	(XPOS),A	Store new position
FEFA		CP	250 dec	Test to check if A has reached
DAE0FB		JR	C,PRNTA	end of line; if not jump to PRNTA
AF	NEWLN	XOR	A	Set A to zero
329DFB		LD	(XPOS),A	Set column position to zero
3A9EFB		LD	A,(YPOS)	Load A with line position and
3C		INC	A	add 1
FE18		CP	18	If print position has reached
CAE0FB		JP	Z,PRNTA	the bottom jump to PRNTA
32A0FB		LD	(YPOS),A	Store new line position
C3E0FB		JP	PRNTA	Jump back to PRNTA
00---	TABLE	DEFB	---	Look-up table containing the shapes of the characters

CHAPTER 13
Spectrum Speaks

You may remember that in the chapter on sound I mentioned the possibility of a speaking Spectrum. Special speech-generating accessories may now be purchased that allow the production of words from either the speaker or a loud extension speaker. The problem with incorporating these effects into programs is that they will only run on other Spectrums that have identical accessories fitted. At the cost of reduced quality, the following program will allow you to include words and messages that are intelligible if not exactly hi-fi. It will work on any 48K Spectrum and the results may even be placed back on a 16K machine if you are using a few short words.

The first thing to explain is how to get the program into your machine. The listing, Program 13.1, is all that is required. As you can see, it contains machine code in the form of DATA statements which are POKEd into RAM above an altered RAMTOP. When you have entered the program (and made a safety copy), run it to see if there are any errors in your data statements. A message will inform you if the program detects an error. However, it is possible that, as the 'checksum' method is very simple, two errors may have the effect of cancelling each other out. If the program does not work then recheck the data statements, or cross-check the RAM contents with the monitor program, both against the decimal data and the hex op codes in the assembler listing.

Program 13.1: The Spectrum Speaks

```
   1 REM     The Spectrum Speaks
   2 REM
   3 REM     Install code
   4 REM
  10 CLEAR 32767: RESTORE : LET sum=0: F
OR x=32768 TO 32867: READ y: LET sum=sum
+y: POKE x,y: NEXT x: GO SUB 50: IF sum
<> 11185 THEN  PRINT "There is an error
in the DATA.": STOP
  17 REM
  18 REM     Menu
  19 REM
  20 CLS : PRINT  AT 2,8;"SPECTRUM SPEEC
H"; AT 6,7;"MENU:-"''"1-","To load speec
h"''"2-","To edit speech"''"3-","To save
 message"; AT 18,2;"Press the number you
 require"
  30 LET a$= INKEY$ : IF a$<"1" OR a$>"3
" THEN  GO TO 30
  40 GO SUB 100* VAL a$: GO TO 20
  47 REM
  48 REM      Reset start,length
  49 REM
  50 LET start=0: LET length=28572
  60 LET h= INT (start/256): LET l=start
-(256*h): POKE 32808,l: POKE 32809,h: LE
T h= INT (length/256): LET l=length-(256
*h): POKE 32819,l: POKE 32820,h: RETURN
  67 REM
  68 REM      Get keypress
  69 REM
  70 IF  INKEY$  <> "" THEN  GO TO 70
  80 IF  INKEY$ ="" THEN  GO TO 80
  90 RETURN
  97 REM
  98 REM     Load message
  99 REM
 100 CLS : PRINT  AT 3,1;"Start tape the
n press any key"''
 110 GO SUB 70: PRINT "Loading..."'': RA
NDOMIZE  USR 32768
 120 PRINT "Message saved:-"''"Press any
 key to replay"''
 130 GO SUB 50: GO SUB 70: PRINT "Replay
ing...": RANDOMIZE  USR 32813
 140 RETURN
 197 REM
 198 REM     Edit message
 199 REM
 200 CLS : PRINT  AT 3,3;"The editing op
tions are:-"''"1-"; TAB 10;"Change start
"''"2-"; TAB 10;"Change length"''"3-"; T
AB 10;"Restore old values"''"4-"; TAB 10
;"Play message"''"5-"; TAB 10;"Return to
 menu"
```

```
 210 PRINT  AT 17,4;"Press number requir
ed"; AT 20,0;"Start=";start; TAB 5; AT 2
0,15;"Length=";length; TAB 5
 220 LET a$= INKEY$ : IF a$<"1" OR a$>"5
" THEN  GO TO 220
 230 PRINT  AT 17,4; TAB 25: GO TO 10* V
AL a$+230
 240 INPUT "New start?";start: LET start
= ABS start: GO TO 210
 250 INPUT "New length?";length: LET len
gth= ABS length: GO TO 210
 260 GO SUB 50: GO TO 210
 270 LET start=start+8: GO SUB 60: RANDO
MIZE  USR 32807: LET start=start-8: GO T
O 210
 280 GO SUB 60: RETURN
 297 REM
 298 REM     Save message code
 299 REM
 300 CLS : FOR x=32867 TO 32813 STEP -1:
: POKE x+start, PEEK x: NEXT x
 310 INPUT "Name?";a$: SAVE a$ CODE 3281
3+start,length: CLS : PRINT  AT 10,5;"Re
wind tape and verify": VERIFY a$ CODE
 320 PRINT  AT 10,0;"Tape O.K. Size of r
outine=";length''"Press a key..........":
 GO SUB 70: IF start<55 THEN  GO TO 10
 330 GO TO 20
 397 REM
 398 REM     Machine code
 399 REM
 400 DATA 62,15,211,254,243,33,100,128
 410 DATA 17,0,240,14,0,6,0,4
 420 DATA 40,7,219,254,230,64,185,40
 430 DATA 246,112,79,35,229,167,237,82
 440 DATA 124,181,225,32,232,251,201,33
 450 DATA 0,0,9,68,77,33,55,0
 460 DATA 9,229,17,156,111,25,235,225
 470 DATA 58,72,92,203,63,203,63,203
 480 DATA 63,230,7,79,243,70,4,5
 490 DATA 40,4,47,230,16,177,211,254
 500 DATA 0,0,0,5,32,248,35,229
 510 DATA 167,237,82,71,124,181,225,120
 520 DATA 32,227,251,201
```

Assuming that you have correctly loaded the program, this is how you use it. The menu offers you three options. The first of these is for entering the message that you wish your computer to utter. Record the spech on cassette using the highest quality system you have available. If you can use a cassette machine without an automatic level control, so much the better as, when machines with this facility are used to record from a microphone, they tend to accentuate the background noise between words.

Now place the cassette, cued up to the start of the message, into the cassette player that you use to save and load Spectrum programs with, plugged up for loading. Initially, set the volume control to a little below whatever you would normally use to load a program. Select option one on the menu and the program will prompt you to start the tape and press a key. Do this and, after a period of time, a message will tell you that the loading has finished. Again press a key, and with any luck you should hear the speech played back.

If the time taken to load the speech was more than about ten seconds and the playback is either just silence or silence plus very distorted words, then you need to increase the volume setting on the cassette player. If recording took less than three seconds but also playback was distorted, then decrease the volume. Keep experimenting until you find the optimum level and then mark the volume setting. You now have a message inside the machine.

The second menu option allows you to 'edit' the message so as to play only the word or words you want and not the noise or silence either side. Firstly, keep increasing the 'start' parameter until the word you want is said immediately you press the play key. Secondly, reduce the 'length' value so that there is nothing extraneous on the end. When you are happy with the result you can move on to option 3 which will save and verify the machine code on tape.

Now you have recorded on tape a machine code program that will speak a word or phrase. To incorporate it into another program, make a note of the 'size' given by option 3. This is the amount of space that must be allowed for the code — you can load the code into any RAM location that is free, provided there is sufficient room above. For example, let's assume that 'size' is given as 4000, that is to say that the machine code routine requires 4000 bytes of space. On the 48K machine RAMTOP is normally 65367, so CLEAR 61367 will allow us to load the code using LOAD "" CODE 61368. In order to get the Spectrum to speak the message, use RAND USR the address that the code has been loaded to, in this case 61368.

The principle behind the machine code section of the speech program is quite simple — refer to the assembler listing for a detailed analysis. The record routine monitors bit 6 of port FEh, and counts time with the

B register. If the input from the tape changes state (ie switches from high to low or vice versa), or if the B register becomes 'full' and cycles back to zero, the contents of B are stored in a vast table of RAM, the pointer is incremented, and the process repeats until the table is full. Thus recording silence is much more efficient on memory than sound. The replay routine is more or less the record routine in reverse. It fetches data from the table and toggles the speaker through bit 4 of port FEh at the pace dictated by the daa. The second routine is written so that it can be position independent — when the 'turnkey' code is saved, it is relocated at the start of the required section of data.

You can have a lot of fun with this program, despite its somewhat low fidelity. It is quite possible to identify the speaker of the message, so perhaps you could get Kenneth Kendall to record one for you!

Assembler listing for speech program machine code

Note: The op codes are given in hexadecimal.

Code	Label	Instruction		Comments
3E0F	LOAD	LD	A,0F	Clear port
D3FE		OUT	(FE),A	
F3		DI		Disable interrupts
216480		LD	HL,8064	Point HL to the beginning of the storage table
1100F0		LD	DE,F000	Point DE at end of table
0E00		LD	C,00	Clear C
0600	NBYTE	LD	B,00	Set counter to zero
04	LISTN	INC	B	Increase count by one
2807		JR	Z,STORE	If count has cycled back to zero then jump to STORE
DBFE		IN	A,(FE)	Read port FE into A
E640		AND	40	Isolate bit 4
B9		CP	C	If A matches C (holding last
28F6		JR	Z,LISTN	value read in) jump to LISTN
70		LD	(HL),B	Store the count in the table
4F		LD	C,A	Load C with new port state
23		INC	HL	Point to next space in table
E5		PUSH	HL	Save HL on the stack
A7		AND	A	Clear the carry flag
ED52		SBC	HL,DE	Compare HL with DE and if
7C		LD	A,H	equal (ie the pointer has
B5		OR	L	reached the end of the table) then set the Z flag
E1		POP	HL	Retrieve old value of HL

```
20E8              JR     NZ,NBYTE      If Z flag not set jump to
                                       NBYTE
FB                EI                   Re-enable the interrupts
C9                RET                  Return to BASIC
210000   TEST     LD     HL,0000       Load HL with start offset
09                ADD    HL,BC         Add HL to the 'RAND'
44                LD     B,H           address held in BC and transfer
4D                LD     C,L           result back into BC
213700   REPLY    LD     HL,0037       Load HL with length of
                                       REPLY
09                ADD    HL,BC         Point HL to start of table
E5                PUSH   HL            Store HL on the stack
119C6F            LD     DE,6F9C       Load DE with size of table
19                ADD    HL,DE         Point HL to end of table and
EB                EX     DE,HL         transfer it to DE
E1                POP    HL            Restore HL
3A485C            LD     A,(5C48)      Load A with contents of
                                       BORDCR
CB3F              SRL    A             Divide A by 8
CB3F              SRL    A
CB3F              SRL    A
E607              AND    07            Mask off high five bits
4F                LD     C,A           Put border colour in C
F3                DI                   Disable the interrupts
46       OUTBY    LD     B,(HL)        Load B with byte from table
04                INC    B             Test B to see if it is zero
05                DEC    B
2804              JR     Z,LOOP        If it is jump to LOOP
2F                CPL                  Complement bit 4 of A
E610              AND    10
B1                OR     C             Add in the border colour
D3FE              OUT    (FE),A        Output bit 4 of A to speaker
00                NOP                  Delay so that speed of replay
00                NOP                  matches record routine
00                NOP
05                DEC    B             Decrement counter and if not
20F8              JR     NZ,LOOP       zero jump to LOOP
23                INC    HL            Point HL to next byte
E5                PUSH   HL            Store HL
A7                AND    A             Reset carry flag
ED52              SBC    HL,DE
47                LD     B,A           Store A
7C                LD     A,H           If HL has reached end of table
B5                OR     L             then set the zero flag
E1                POP    HL            Restore HL
78                LD     A,B           Restore A
```

```
20E3              JR     NZ,OUTBY      If the Z flag is not set jump to
                                       OUTBY
FB                EI                   Enables the interrupts and
                                       then
C9                RET                  return to BASIC
00---    TABLE           A large free area of memory
```

Endword

We have reached the point where the hardware and firmware aspects of using the Sinclair Spectrum have been covered. How you use the information will depend on how you use your computer.

Some people drive a car with no desire to open the bonnet, others prefer to have some idea of how it works even if they don't ever intend getting their hands dirty. If you see yourself as a potential computer 'mechanic', then you will need to acquire machine code skills as well as a great deal of patience. There are many books on the finer points of hardware and Z80 machine code. As for books about patience . . .