AN EXPERT GUIDE TO THE SPECTRUM

An Expert Guide to the Spectrum

An Expert Guide to the Spectrum

Mike James

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing

British Library Cataloguing in Publication Data
James, M.
An expert guide to the Spectrum.
1. Sinclair ZX Spectrum (Computer)
I. Title
001.64'04 QA76

ISBN 0 246 12278 1

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface

The Sinclair Spectrum is a phenomenally successful microcomputer, and deservedly so. It is always surprising to discover how much it can achieve with so little programming effort. It can be considered a revolutionary machine because it introduces new ways of doing things. For example, ZX BASIC is a new and excellent dialect of BASIC, and its video display uses parallel attributes for colour control. The advent of Interface 1 and the Microdrives has resulted in even greater versatility and power.

Lots of microcomputer users must have been wondering what exactly makes the Spectrum such a success, and this book sets out to explore a variety of reasons and enable Spectrum users to put all its remarkable features to good use. The book is therefore about both the Spectrum's hardware and its software, and the vital interaction between them.

After an introductory chapter which discusses general aspects of computer technology, the next three chapters examine the standard 16K or 48K Spectrum, exploring it both in terms of the chips that make it up and ZX BASIC. Chapter 5 describes the sophisticated I/O system hidden within the standard Spectrum, a system based on streams and channels. The video display is obviously an important part of any application, and two chapters are devoted to describing how it works, and giving examples of how this knowledge can be put to good use. The Spectrum's standard peripherals – the tape system, the sound generator and the ZX printer – form the subject matter of Chapter 8 while Chapters 9 and 10 are devoted to Interface 1 and the Microdrives. Chapter 11 introduces the RS232 interface and the Sinclair Network, showing the Spectrum's communication potential. The final chapter is a collection of applications examples to indicate the sort of advanced projects you can tackle for yourself.

This book assumes a working knowledge of BASIC at an introductory level, and builds on this foundation. Although it is

outside the scope of the book to teach assembly language, it includes many examples of applications where assembly language is a great advantage, and in these cases appropriate machine code routines are presented and incorporated into BASIC programs. If you've learnt assembly language programming and are wondering what to do with it, these examples will give you plenty of ideas. If, on the other hand, you've not yet picked up this knowledge, you can use the routines in any case – though they may well persuade you of the advantages assembler can offer, and provide a stimulating introduction.

You'll find that this book contains a lot of material – probably more than can be absorbed in one go. Don't worry if there are parts of it you don't understand first time round. Like lots of technical subjects, computing cannot be grasped simply by reading about it. You have to experiment and try things out for yourself before you really come to terms with it. Don't be frightened to explore ideas of your own – this book aims to give you some leads and pointers, but they can only be the tip of the iceberg.

Lots of the ideas in the book are interdependent – you will find that as you are introduced to new ideas in later chapters, you will gain a deeper understanding of material presented earlier. Because of this, you can't expect to start reading this book at page one and carry on reading through to the end of the final chapter, having assimilated every word. Instead, I hope you will find yourself turning back to re-read sections as they begin to make more sense in the light of new information. Equally, I hope you find that this is a book that will last, in the sense that it contains enough interesting material to keep you busy in lots of areas for a long time. Above all, I hope I manage to indicate why the Spectrum is such an exciting micro, and how to go about making the most of its enormous potential.

My grateful thanks are due to Richard Miles and Sue Moore of Granada Publishing for all their hard work and help in the preparation of this book.

Mike James

Chapter One

Becoming an Expert

To be an expert on any computer it is necessary to know something about its software and something about its hardware. In fact the division between software and hardware is not clear cut. You cannot specialise in one without the other getting in the way! When using personal computers like the Spectrum most of the really exciting things happen when software takes advantage of the hardware in new ways. Fortunately this doesn't mean that every programmer has to become a hardware engineer! Electronics can be a difficult and time-consuming subject; but in computer programming what really matters is an understanding of how the hardware affects what can be achieved through software. Much of this book is concerned with explaining the Spectrum's hardware from the point of view of a creative programmer. This chapter presents an overview of computer hardware in general. Chapter 2 describes how this applies to the Spectrum in particular. Much of the material in these first two chapters is used and developed further in the later chapters dealing with more specific topics.

If all this talk of hardware is making you afraid that the software side of things is going to be neglected, then Chapters 3, 4 and 5, which look at the inner workings of ZX BASIC, will reassure you that software really is as important as hardware. Later chapters describe the workings of some of the standard Sinclair peripherals for the Spectrum – the cassette system, the ZX Printer, Interfaces 1 and 2 and the Microdrive – and explain their use in such applications as program storage and networking.

All you need to understand the material in this book is a knowledge of ZX BASIC. If you are a BASIC beginner then I recommend you look first at an indroductory book on the subject such as *The Spectrum Programmer* by S. M. Gee, published by Granada. Although BASIC will normally be used to illustrate the ideas described, it is not always possible to achieve the speed needed

using nothing but BASIC. When this is the case there is really no choice but to use Z80 assembler. While this book doesn't deal in any depth with Z80 assembler, avoiding its use altogether would prohibit too many interesting subjects. The solution adopted is, where necessary, to give Z80 assembly language programs that can be used within BASIC programs as USR functions. What such USR functions do, and the general way that they do it, will be described, but the actual detailed code will not. If you understand Z80 assembly language then the program listings, complete with their comments, will be enough for you to know how the programs work. If you do not understand it then you will have only a general appreciation of what the programs are doing, but you will still be able to use them from BASIC. In other words, while you will be able to follow the algorithms involved, you won't necessarily understand the details of the code.

It is helpful to realise that although many computer books do contain a logical progression of ideas from Chapter 1 to the end, this doesn't mean that you have to read and fully understand each chapter before moving on to the next. There is an old saying that the best way to read a computer manual is to read it once forwards, then once backwards, and then make your first attempt to understand it! There is more than a little wisdom in this suggestion, and the backwards and forwards approach often pays dividends: information given later will improve your understanding of what has already been described. It is worth keeping this idea in mind while you read this book. If you find that you are not sure that you understand something, resist the temptation to backtrack. Read on to the end of the section. It is surprising how often small details fit into place when you have managed to get an overview of the situation. Don't expect to understand all of *An Expert Guide to the Spectrum* at first reading. Some of the material will only be useful to you when you actually put it into practice, and only then will it really make sense to you. In this respect the *Expert Guide* is also a reference book for the future.

The parts of a computer

All computers have certain features in common. In particular they all have a CPU (Central Processing Unit) that is responsible for carrying out the instructions within your program. They all have some kind of memory to hold your program and data, and they all

have some kind of input/output (I/O) device to allow you to communicate with your program (see Fig. 1.1). This simple picture is complicated by the fact that there are a number of different types of computer memory and a very wide range of possible I/O devices.

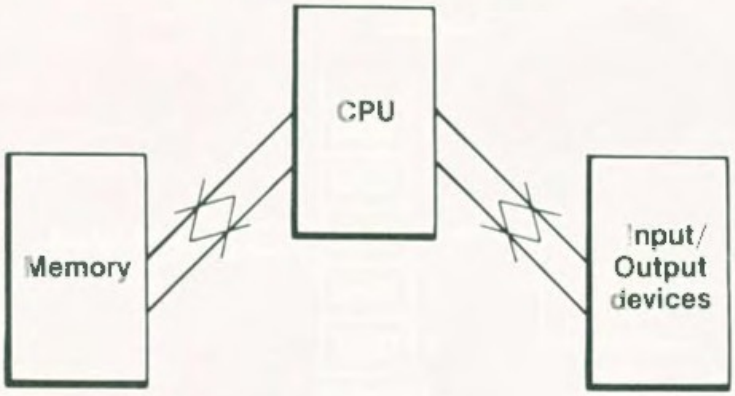


Fig. 1.1. The parts of a computer.

Memory can be divided into two types, *primary* and *secondary* or *backing store*. Primary memory is used to store programs that the CPU is carrying out and data that it is actually processing. Secondary memory, such as tape storage, is used to store the machine's 'library' of programs and data. Primary memory is further sub-divided into *Random Access Memory* (RAM) and *Read Only Memory* (ROM). The difference between RAM and ROM is that the information stored in RAM can be changed, but the contents of ROM are fixed at the time of manufacture. RAM is badly named: 'Random Access Memory' conveys little of the essential difference between RAM and ROM. As RAM can be both read and written to, it might be better to call it 'Read And Write Memory' in contrast to 'Read Only Memory'.

Any computer will contain a certain amount of RAM, used to store user programs, and a certain amount of ROM, containing any fixed information that the machine needs to run your programs. In the case of the Spectrum, and most other microcomputers, the ROM is used to hold the rules for the BASIC language – in other words the *BASIC interpreter*. Before going on to examine the Spectrum's hardware it is worth looking briefly at the way information is stored in memory.

Addresses, data and bit patterns

From the point of view of the CPU, memory looks like a collection

of numbered locations, each one capable of holding some data. The number that identifies each memory location is called its *address* (see Fig. 1.2). The data that can be stored in a memory location takes the form of a bit pattern. As a bit is either a one or a zero, a bit

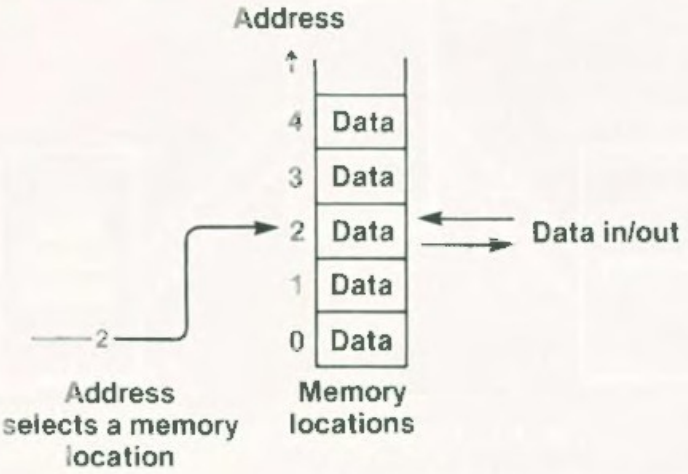


Fig. 1.2. Use of an address to find data in a memory location.

pattern is exactly what its name suggests – a pattern of ones and zeros. For example, 01010 is a bit pattern. Most micros, the Spectrum included, have memories that can store a pattern composed of eight bits in each memory location. This size of bit pattern is so common that it is given a special (and well-known) name – a *byte*. A bit pattern can be used to represent the more familiar forms of data that are encountered in BASIC, but it is important to realise that a bit pattern is all that a memory location can store.

Binary numbers are the best-known use of bit patterns to represent data, so it is useful to go over the details of this representation. The eight bits stored in a memory location are usually labelled b0 (bit zero) to b7 (bit seven) as shown below:

```
b7 b6 b5 b4 b3 b2 b1 b0
```

Each bit in the bit pattern that represents a binary number is associated with a value:

```
b7 b6 b5 b4 b3 b2 b1 b0
128 64 32 16  8  4  2  1
```

If you look at these values carefully you will see that starting from the 1 associated with b0 each value increases by a factor of two for each position that you move to the left. Another way of looking at this is that each value is equal to 2↑n where ↑ means 'raise to the

power' and n is the bit number. To convert a binary number to the more familiar decimal notation all you have to do is add up the values associated with each 1 in the bit pattern. For example:

```
b7 b6 b5 b4 b3 b2 b1 b0
0  0  1  0  1  0  1  0
```

is 32+8+2 or 42 in decimal.

If you want to convert a binary number to decimal the easiest way is to use the Spectrum's BIN function. Type:

```
PRINT BIN x
```

where x is the binary number for which you want to know the decimal equivalent. The Spectrum will obligingly convert and print it for you. Remember you can always use the computer to avoid the difficult arithmetic that so often puts people off simple subjects such as binary numbers! It is more important that you understand the idea of using a bit pattern to represent a number rather than be able to perform miracles of mental arithmetic in converting from binary to decimal! Unfortunately, the Spectrum doesn't have a function that will convert a number in decimal form into binary, but such a conversion isn't often needed. For the few occasions when it is, the following subroutine will accept a decimal number in D and return the bit pattern that represents it as a binary number in the string B$.

```
1000 LET B$=""
1010 LET B=D-INT(D/2)*2
1020 IF B=0 THEN LET B$="0"+B$
1030 IF B=1 THEN LET B$="1"+B$
1040 LET D=INT(D/2)
1050 IF D=0 THEN RETURN
1060 GOTO 1010
```

High level languages such as BASIC go to a lot of trouble to hide the fact that memory can only store bit patterns from the user. However, the data types you find in BASIC – numbers, strings and arrays – are created out of this more fundamental data type. Once you know something about bit patterns and binary numbers it becomes much easier to understand how and why computers work. For example, each memory location can only hold eight bits. This means the smallest binary number that can be stored is 00000000, or zero, and the largest is 11111111 or, if you convert this to decimal, 255.

As already mentioned, the memory location when data is stored

or retrieved is specified by a number called an address. This, too, has a close connection with bit patterns and binary numbers. From the point of view of computer hardware, the most important feature of a bit pattern is that each bit needs only two states to represent it. Normally these two states are written as zero and one, but there is nothing to stop us from renaming them 'off' and 'on' without altering anything that matters. Computer hardware uses two voltage states, low and high, to represent the bits that make up a bit pattern. For example, the eight bits in a single memory location are stored as a pattern of low and high voltage states. In the same way, the number that is used to select a single memory location – i.e. the address – can also be represented by a bit pattern of low and high voltage states. Most micros, including the Spectrum, use 16 bits to specify the address of the memory location in use. The lowest 16-bit binary number is 0 and the largest is 65535. This determines the maximum amount of memory that can be handled, which doubles each time a bit is added to the address:

1 bit address can handle 2 memory locations
2 bit address can handle 4 memory locations
3 bit address can handle 8 memory locations

and so on. It is therefore easier to measure memory sizes in a way that takes this into account. Instead of using 1000 memory locations as the basic unit of memory size, it is more convenient to use 1024 or 1Kbyte. Using a 10-bit address you can handle a maximum of exactly 1024 memory locations or 1K of memory. Thus using an 11-bit address you can handle a maximum of 2K of memory, using a 12-bit address you can handle a maximum of 4K, and so on up to a 16-bit address which will handle 64K of memory. If the basic unit of memory was 1000 memory locations then the numbers associated with each address size would be very messy.

Bit patterns in hardware – the bus

Bit patterns and binary numbers are very much part of the software side of a machine. However, they do correspond to something that is very much part of a machine's hardware – the *bus*. In the last section we saw how a bit is represented in hardware by two different voltage states – low and high. Clearly a group of bits – a bit pattern – will need a pattern of voltages to represent it in hardware. A bus is just a group of wires used to convey a bit pattern, in the form of voltage

levels, from one part of the computer to another. Each wire in the bus carries the state of one bit. For example, the CPU generates addresses which are conveyed to the memory by the *address bus*. If the CPU uses 16-bit addresses then the address bus is composed of 16 wires, each carrying the state of one bit in the address. In a real computer system the address bus leaves the CPU and is connected to all of the parts, memory and I/O devices, that need to be informed of the current address that the CPU is using.

In the same way, data is passed around the computer by way of a *data bus* that connects all data-receiving and data-transmitting parts of the computer. If the CPU and the memory work with eight-bit data, then the data bus will consist of eight wires. Notice that the data bus is different from the address bus in that it can carry bit patterns to *and* from the CPU. The address bus and the data bus connect up all the parts of a computer to make it a single machine.

As well as these two fundamental hardware buses there is usually a small group of wires that connect the CPU to the rest of the machine – the *control bus*. The control bus carries a bit pattern that synchronises the workings of the whole machine and passes information about what different parts of the machine are doing. For example, the control bus usually includes a wire that signals whether or not the CPU is reading data in. From the software point of view, very few of the signals carried by the control bus are likely to be of any use.

After this discussion of computers in general it is time to turn our attention to the Sinclair Spectrum, and to discover what makes it special.

Chapter Two

Inside the Spectrum

The Spectrum is a very special computer. Most of its hardware is incorporated in a single purpose-built chip called a ULA, standing for *Uncommitted Logic Array*. This single fact is responsible for the Spectrum's high performance and low price. However, the way that the ULA is designed makes it difficult to alter the way that the machine works, and in this sense the Spectrum is a 'single-mode' machine. For this reason there is little point in examining the Spectrum's hardware in detail, for instance with a complete circuit diagram. A circuit diagram isn't even very useful if you are trying to repair a Spectrum, because the number of components is very low, and one of the largest – the ULA – is available only from Sinclair! However, it is worth gaining a general idea of the overall functioning of the Spectrum, and a detailed knowledge of one or two important 'external' connections such as the loudspeaker and tape circuits. After all, detailed hardware knowledge is only of use if it helps you to alter the way that software behaves, or if it can be used to change or add to the workings of the Spectrum.

The CPU

The CPU used by the Spectrum is the very popular Z80A microprocessor. The only difference between the standard Z80 chip and the Z80A is that the Z80A can work twice as fast as the Z80. The working speed of a microprocessor is governed by the maximum clock frequency it can accept. The clock is simply a regular pulse that the microprocessor uses to synchronise all the different operations necessary to obey an instruction. The number of clock pulses needed to carry out each instruction depends on the complexity of the instruction. In theory, the Z80A can work with a clock up to 4MHz, giving a singie clock pulse time of ¼ of a millionth of a second! In

practice the Spectrum uses a clock of 3.5MHz which is not quite as fast as it could be.

The Z80 is a fairly ordinary microprocessor. As it processes data eight bits at a time it is called an *eight-bit processor*. It has 16 address lines which give it a maximum addressing range of 64K, all of which is used in the case of the Spectrum. One important feature of the Z80 is that it has an additional 64K of address space that is dedicated to I/O devices. This is achieved by adding what amounts to an extra address bit called IORQ (Input Output ReQuest) which will select between 64K of memory and 64K of I/O devices. This sounds like a powerful facility, and indeed it is, but there is a limitation. All the instructions that the Z80 can obey will work on any memory location, but the 64K of I/O devices have their own special and very restricted set of instructions. These essentially amount to reading data in and writing data out to whatever I/O devices are present. (See **IN and OUT** later in this chapter.)

You may find it puzzling to talk of I/O devices in the same way as memory locations, but this is exactly what I/O devices look like as far as the computer is concerned. An I/O device sends data to and receives data from the computer in exactly the same way as a memory location. The main difference is that any given I/O device might correspond to a number of I/O locations or 'ports'. For example, the ZX printer is an I/O device. It uses a single port at address 251 to receive the data that determines what it prints, and to send data back to the Spectrum to indicate what 'state' it is in. The Microdrives and Interface 1 use three I/O ports at 254, 247 and 239 to communicate with the Spectrum. The use of I/O ports and I/O instructions will be discussed in more detail, with practical examples, in later chapters.

The most important feature of the Z80 as far as the programmer is concerned is that it determines the machine code and assembly language that the Spectrum uses. It is not the purpose of this book to teach Z80 machine code but, as already mentioned, it will be used where there is no other way to achieve the processing speed necessary, for a demonstration. If you would like to learn Z80 machine code or assembly language then there are suggestions for further reading at the end of this book.

The memory

The Spectrum's memory addressing space is divided up into two

parts, as can be seen in Fig. 2.1. The 16K ROM is used to hold all the machine code necessary to implement the rules of ZX BASIC, and subroutines to handle the Spectrum's standard hardware. For instance, there is a subroutine that will read the keyboard, and

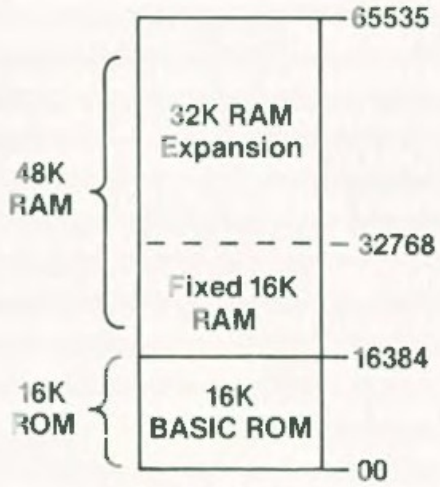


Fig. 2.1. The structure of the Spectrum's memory addressing space.

another that will make a sound using the loudspeaker. This ROM is implemented as a single 16Kbyte chip. If you are feeling adventurous, and have the necessary programming hardware, you can replace this ROM with a 2718 EPROM containing your own machine code program. EPROM stands for Erasable Programmable Read Only Memory, and is simply a type of ROM in which it is possible to store a program using fairly cheap equipment. An EPROM can be wiped clean by exposing it (for some minutes) to ultraviolet light, and is therefore reusable. However, you would have to copy many of the subroutines to handle the hardware – such as the keyboard and the video display – into your new EPROM; and writing 16K of machine code is not something to be tackled lightly.

The Spectrum's RAM is split into two sections. The first 16K is always present, and is used to store the information that generates the video display as well as a lot of system information and user programs. The final 32K is optional, and is added to the basic 16K Spectrum to bring it up to the maximum 48K of RAM. Both sections are implemented using dynamic RAM chips. The 16K section uses eight standard 4116 16K bit chips and the 32K section uses eight 4532 32K bit chips. The 4532s are rather special. They are only available from Texas instruments, and are 'failed' 64K chips. It is difficult to make memory chips that can store as much as 64K bits, and to save throwing away large quantities of chips with only a few

faults Texas Instruments designed their 64K bit chip to work as separate 32K halves. If the faults all lie in one half of the chip then it certainly cannot be used as a 64K bit chip, but there is no reason why it cannot be used as a 32K bit chip – and this is what a 4532 is. If you have a 16K Spectrum and want to upgrade it to 48K then my advice is to buy a complete upgrade kit from one of the many suppliers: the Texas chips are fairly difficult to get hold of. Early Spectrums (Issue 1) cannot be upgraded simply by adding missing chips, because an extra printed circuit board has to be used. You can tell an Issue 1 Spectrum by removing the bottom of the case and looking at the printed circuit board to the right of the two jack sockets (EAR and MIC). There you will see the words 'ISSUE ONE'. An Issue 2 board is marked 'ISSUE TWO' on the front edge of the printed circuit board, just right of centre. (Later issue numbers will also be shown just right of the centre of the circuit board.) Upgrading an Issue 2 Spectrum is just a matter of soldering in twelve chips the right way round and making one wire link.

You may be wondering what the word 'dynamic' means when applied to RAMs. The answer is that there are two different ways of implementing RAM – static and dynamic. Static RAM will hold the data that is stored in it until it is changed or until the power is switched off. In this sense it is simple to use, reliable, and easy to test. The trouble is that manufacturers haven't been able to make static RAM chips with very large capacities. Dynamic RAM, on the other hand, is available in sizes up to 64K bits per chip, for very reasonable prices. Its disadvantage is that information stored in it fades away unless it is read and rewritten every now and again. This reading and re-writing of information is known as 'refreshing' dynamic RAM, and in practice special circuitry is supplied to carry it out in a way that the user will not notice. In the Spectrum's case, refreshing is carried out by the Z80 and the ULA working together, and the whole of the 48K is refreshed without loss of performance or any trouble on the user's part.

BASIC access to memory - PEEK and POKE

ZX BASIC provides two direct ways of examining and altering memory locations. The command

PEEK (address)

will return the contents of the memory located at 'address'. As

'address' must be a 16-bit binary number (see Chapter 1) it must lie in the range 0 to 65535. Similarly, as the data stored in the memory is a bit pattern consisting of eight bits, the value returned (as a binary number) by PEEK has to lie in the range 0 to 255. The command

```
POKE address, data
```

will store the bit pattern corresponding to the binary representation of 'data' in the memory located at 'address'. Once again, 'address' should be in the range 0 to 65535, and 'data' should be in the range 0 to 255.

Notice that although both PEEK and POKE work in terms of decimal numbers it is very often the underlying bit pattern that is of interest. For example, when defining new characters (see Chapter 6) each pixel is represented by a single bit which is 1 for an ink pixel and 0 for a paper pixel. To POKE a bit pattern representing ink/paper pixels it would be necessary to treat the bit pattern as a binary number, then convert this binary number to decimal. Fortunately, ZX BASIC includes the BIN command which makes the conversion to decimal unnecessary. If you want to POKE a bit pattern into a memory location then you can use:

```
POKE address,BIN x
```

where x is the bit pattern. However, this method fails if x is a variable (BIN will not work with variables) and PEEK always returns a decimal value. For this reason, later chapters will introduce methods of using BASIC to manipulate decimal values as if they were bit patterns.

The video display

The Spectrum's video display uses a very ingenious system of parallel attributes to obtain an eight-colour display (with some restrictions) in not much more memory than would be used for a black and white display. Nearly all the work involved in generating the display is the responsibility of the ULA chip. It is something of a disappointment that the ULA chip is not programmable to produce different display modes. From the moment that the Spectrum is switched on, the ULA displays the information stored in a fixed area of memory – the video RAM – to produce a fixed format (256 dots by 192 dots) colour display. This single display mode of operation

offers little scope for experiment. However, 256 by 200 dots in colour is a more than adequate display resolution.

The only really useful aspect of the video display's hardware is the way that the video RAM determines what is displayed on the screen: this is the subject of Chapter 6. However, it helps to have a complete picture of the way things work, so the general principles behind the generation of the video display will be explained here.

The video RAM is always the first 6912 bytes in the lower 16K of RAM. While a TV picture is being displayed this area of RAM is accessed by the ULA, and the information it contains is used to determine the colour of each dot or *pixel* (picture element) on the screen. A TV picture (in the UK at least) is composed of 625 scan lines displayed every fiftieth of a second. To produce a stable picture, the ULA must not only generate the synchronising signals that mark the beginning of every line and every frame, it must also retrieve data from the video RAM fast enough to determine the colour of each pixel in the scan. It must also retrieve each item of data just before the pixels that are controlled by it are displayed in the scan. In other words, to produce a stable picture the ULA must be able to access the video RAM at any time that it needs to. The only major difficulty with this is that the CPU also has to have access to the video RAM occasionally. Otherwise how would the data that controls the display ever be changed? This means that the video RAM's data and address bus have to be shared by the ULA, which generates the display, and the CPU, which manipulates it (see Fig. 2.2). Of course, only one of the two can actually be using the video memory at any one moment. If both want to use the memory then some sort of priority has to be established to decide which one has to wait. As the ULA is generating the video display, making it wait for the CPU to use the video RAM would result in gaps (white speckles) in the display. (This is what happens in some other machines.) It is better to make the CPU wait until the ULA is finished with the video RAM, and this is what the Spectrum does. However, there is a hidden problem with this scheme. The ULA and the CPU have to share the data and address buses to access the video RAM. If the CPU is not allowed to use the video RAM while the ULA is using it, it is equally not allowed to use any other RAM or ROM in the system, because the address and data buses are also in use by the ULA. If this limitation was accepted, the resulting machine would run very slowly indeed. Every memory access made by the CPU would have to wait until the ULA wasn't using the memory. The solution adopted for the Spectrum is to provide separate CPU and ULA data and

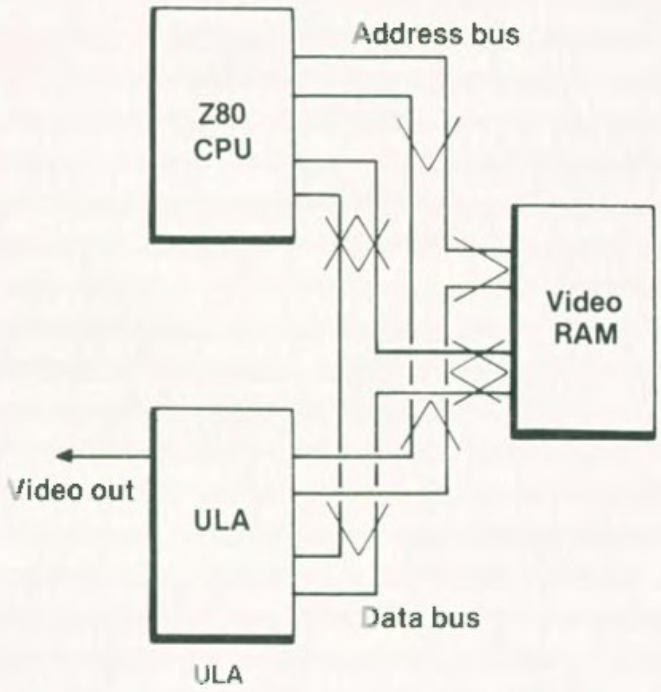


Fig. 2.2. Shared connections between the video RAM, the ULA and the CPU.

address buses. This means the ULA can use the 16K of RAM that contains the video RAM while the CPU can simultaneously use any other memory apart from this 16K. This can be seen in Fig. 2.3, where the ULA can be seen to have a direct connection to the 16K of RAM that contains the video RAM, while the CPU has a direct connection to the rest of the memory. If the CPU wants to use the ULA's 16K this is detected by the ULA, which stops the CPU's clock until it is ready to allow the CPU access to its address and data bus. This causes a slight delay in the CPU's operation when using the lower 16K of memory. It is not usually noticeable when you are running a slow language like BASIC, but it can cause machine code programs stored in the lower 16K to run at different rates. This is only a real problem if timing is critical, or if timing loops are included in the program.

Although it is of little practical use, it is interesting to notice that the Spectrum does not use expensive multiplexing chips to control access to the video RAM's bus. Instead it uses the simplest of all electronic components – the resistor. When the CPU is not trying to use the low 16K of RAM, the two buses work independently, with the signals on one bus appearing at a much reduced level on the other because of the voltage 'dropping' action of the resistors. The voltage reduction is such that on each bus the signals of the other appear as 'noise' and do not influence what happens. However,

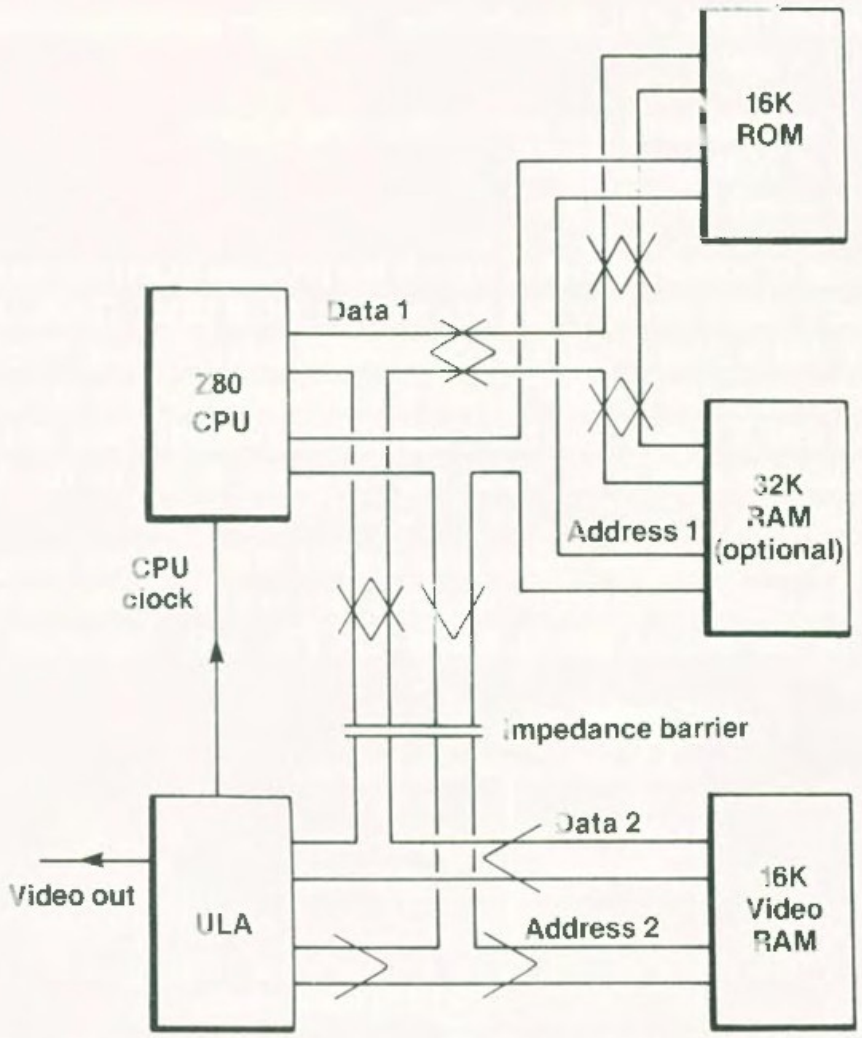


Fig. 2.3. Connections between the main memory areas, the CPU, and the ULA, showing how access to the video RAM is shared.

When the ULA allows the CPU to access its part of the address and data buses it stops 'driving' the buses. The voltage reduction produced by the resistors is now much less, so the CPU's signals gain control. This is a remarkably clever, simple and cheap solution to a very common problem in hardware design, and is typical of Sinclair's ingenious engineering.

The video output circuit

The ULA is responsible for taking the data from the video RAM and using it to construct the colour information as three video signals. However, the task of taking these three colour signals and producing a single PAL (UK standard) colour video signal is the responsibility

of an LM1889N PAL encoder chip. The three signals produced by the ULA are:

Y = luminance and synchronisation signals
U = blue–green signal
V = red–yellow signal

This use of colour difference signals is a problem to anyone wanting to use a video monitor that has only an RGB (Red, Green, Blue) input, but it does simplify the Spectrum's video circuits. The PAL encoder takes the U and V signals and generates a colour or *chroma* signal that is mixed with the Y signal by a two-transistor mixer to produce the final PAL video signal, which is fed to the UHF modulator.

You can adjust the colour and quality of the display using the variable capacitor and resistors positioned in a line on the left-hand side of the printed circuit board (see Fig. 2.4). Adjusting TC1

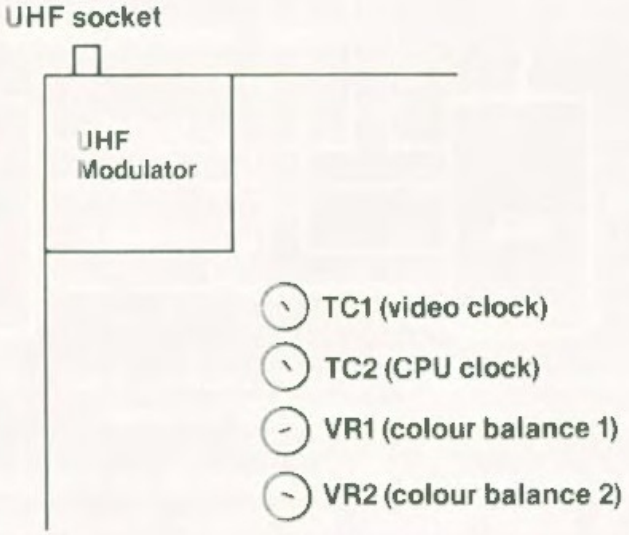


Fig. 2.4. Video output adjustments.

carefully might improve the sharpness of the image by removing any interference patterns. VR1 and VR2 adjust the relative colour balance of the display. VR1 alters the red-yellow balance and VR2 alters the blue-yellow balance. In practice it is better to adjust the colour balance of the TV set that the Spectrum is driving, rather than 'fiddling' with VR 1 and VR2. TC2 adjusts the frequency of the clock pulses to the CPU and should not be altered. The U, V, Y and the composite colour video signals are all available at the rear edge connector which is described later. With a little effort these signals can be used to drive a standard RGB colour monitor, or a black and

white or colour monitor that accepts a composite video signal. This is discussed further after the section on the signals available at the edge connector.

BASIC I/O - IN and OUT

The Z80 provides an additional 64K of address space for I/O devices. However, both data and I/O addresses are carried by the standard data and address buses that connect the CPU to the rest of the computer. As already described, memory and I/O addresses are distinguished by the state of a line in the control bus called IORQ (I/O ReQuest). ZX BASIC provides two additonal commands to access I/O devices, in the same way that it provides PEEK and POKE to allow direct access to memory. The command

IN address

returns a data value from the device located at I/O 'address'. The command

OUT address, data

will send the value 'data' to the device located at I/O 'address'. The main difference between PEEK/POKE and IN/OUT is that all memory locations behave in roughly the same way, but the device located at 'address' can behave in a wide variety of ways depending on its type. Notice also that no storage of data is implied by an OUT command. For example, if a special printer interface was constructed to connect a non-standard printer to the Spectrum, then it might be configured to accept data from, say, I/O address 56. To do this it would have to monitor the address lines and IORQ for the occurrence of the bit pattern corresponding to I/O address 56. When this was detected, it would read in the data currently on the data bus and pass this on to the printer to interpret as a character code. So in this case OUT 56, CODE ("A") would send the ASCII code for A to the printer, but IN 56 would be totally ignored by the printer interface and so wouldn't return any useful data. Some I/O addresses correspond to I/O ports that only accept data such as printer interfaces. In this case it only makes sense to use OUT. Some I/O addresses correspond to I/O ports that will only *supply* data, and in this case it only makes sense to use IN. However, some I/O addresses correspond to I/O ports that can both accept and supply data. A cassette interface, for example, can both read and write data.

In short, to use IN and OUT properly you have to know not only the address that a device occupies but quite a lot about how it functions.

The Spectrum's built-in I/O devices

The Spectrum's built-in I/O devices are the loudspeaker, the cassette interface and the keyboard. All of these are controlled by the ULA. Indeed, the loudspeaker and cassette interface are both handled by a single ULA connection, and in this sense they are a single I/O device!

As already mentioned the Z80 has a separate 64K of addresses that can be used to select I/O devices. However, instead of assigning each I/O device its own address (or group of addresses) the Spectrum assigns each device to a particular bit in the address. For example, the first bit, b0, selects the internal I/O devices connected to the ULA. The action of this bit is such that when it is zero the ULA is selected. Thus any I/O address that has b0 set to zero will select the ULA. In the same way b2 selects the ZX printer when it is zero. If you carry on assigning address bits to devices you should be able to see that the maximum number of devices that can be handled is 16. In fact the Spectrum only uses b0 to b4 of the address to select one of six devices according to the following table.

b0	ULA keyboard/loudspeaker/cassette interface
b1	not used
b2	ZX Printer
b3	Microdrives and Interface 1
b4	Microdrives and Interface 1

However, this only leaves b5, b6 and b7 for special uses: bits b8 to b15 are used to select which column of the keys that make up the keyboard is being read (see later). It is clear that things would be very confused if more than one I/O device were selected at a time, so valid I/O addresses can only have one of b0 to b7 set to zero.

As the ULA is selected by a single bit in the address it might seem impossible for it to handle so many different I/O devices. In fact the ULA behaves differently depending on whether it is being read (using IN) or written (using OUT).

The ULA as an output device

When the ULA is sent data as an output device it controls the loudspeaker and the cassette output connection, MIC. Although it isn't strictly anything to do with I/O, the colour of the TV display's border is also controlled by the ULA, acting as an output device. Each of these internal output devices is controlled by the bit pattern of the data sent to the ULA according to the following plan:

```
b7 b6 b5 b4   b3    b2 b1 b0
 *  *  *  L/S  MIC (  colour  )
```

where * means that the bit isn't used. So the loudspeaker is controlled by b4, the MIC by b3, and the colour of the border by the binary number represented by the three bits b2 to b0. To be able to use this information, all we have to know is the I/O address to use for sending data to the ULA. As the ULA is selected when b0 is zero and b1 to b7 are one, we only have to determine the values of b8 to b15. As noted earlier, address lines b8 to b15 are used to scan the keyboard; so for output they might as well be set to zero. This gives the following bit pattern for the ULA's output address:

```
b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
 0   0   0   0   0   0   0  0  1  1  1  1  1  1  1  0
```

or 254 in decimal.

As an example of using the ULA as an output device, try the following program:

```
10 INPUT B
20 OUT 254,B
30 GOTO 10
```

If you type in numbers in the range 0 to 7, you will see the colour of the border change. Although you can affect the loudspeaker and the MIC output using the same technique, BASIC is so slow that the best you can achieve is a low-pitched buzz. For example:

```
10 OUT 254,16
20 OUT 254,0
30 GOTO 10
```

Line 10 sends the bit pattern 00010000 to the ULA and line 20 sends 00000000. You should be able to see that as this program is in the form of a loop, the result is that b4, which controls the loudspeaker, is continually changing between 0 and 1. This produces a low-

pitched buzz from the loudspeaker. The use of I/O port 254 to control the loudspeaker is described in more detail in Chapter 8. As the loudspeaker and the tape recorder MIC socket are both driven by the same pin on the ULA, the signal to the loudspeaker is also present on the MIC socket. This means that if you record while the rather quiet loudspeaker is making sounds you can replay the tape later and reproduce the sounds at a rather louder volume. Similarly, if you connect an amplifier with a speaker to the MIC socket you can boost the Spectrum's sound to any level that you require. In other words, MIC is not only a 'tape out' connection; it is a 'sound out' connection as well.

The ULA as an input device

When the ULA is used as an input device it sends data to the CPU concerning the state of the EAR input and the keyboard. The keyboard is the most complicated input device, so it will be described first.

Figure 2.5 is a schematic diagram of the Spectrum's keyboard. You can see it takes the form of a rectangular matrix of connections. Each key on the keyboard is arranged so that pressing it connects one of the horizontal wires to one of the vertical wires. Obviously, to identify which key, if any, has been pressed you have to find out which horizontal and vertical wires have been connected. The eight horizontal wires are connected to b8 to b15 of the address bus, so they can be set to different voltage levels according to the bit pattern of the address in use. The five vertical wires are connected to five input pins on the ULA, and when the ULA is used as an input device it is their state that is sent to the CPU as b0 to b4 of the data. In other words, IN 254 'reads the state' of the five vertical keyboard lines, and returns the decimal equivalent of b0 to b4. As the vertical input lines are connected to +5 volts (high) they return a value of 1 when no key is pressed. At the instant when the input lines are read by IN 254 the address on the address bus (i.e. 254) is such that all of b8 to b15 are 0, i.e. low voltage. If a single key is pressed then the vertical line that it connects to the address line will be connected to a low voltage and so will return a 0 in the bit pattern. That is, IN 254 will return a bit pattern that has a zero corresponding to any vertical line that is connected to an address line. This works well for detecting whether or not a key is pressed, but how can you tell which of the eight keys connected to the vertical line it is? The answer is that if all of the

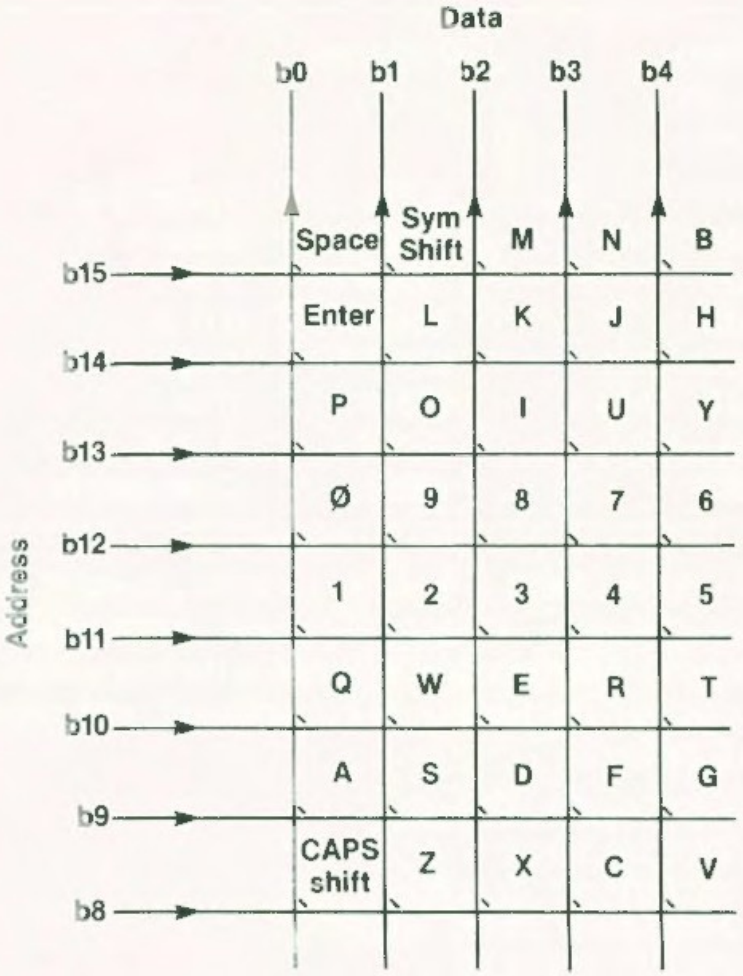


Fig. 2.5. Schematic diagram of the Spectrum keyboard.

address lines b8 to b15 are low, you cannot. The solution is to make only one of the address lines low at a time; then only one row of keys can connect the input lines to low voltage. Thus instead of using the I/O address 254 to read the keyboard, you have to set all but one of b8 to b15 to 1. For example, to read the row of keys connected to the address line b15 you would have to use the following bit pattern for the I/O address:

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

which is 32766 in decimal. That is, IN 32766 will return a value that, when expressed in binary, has b0 to b4 set according to the state of the first row of keys – caps shift to v – with 0 representing a depressed key.

Carrying on in this way gives the following decimal values for the I/O addresses to read each row of the keyboard matrix:

address line set to zero	I/O address	keys
b15	32766	Space to B
b14	49150	Enter to H
b13	57324	P to Y
b12	61438	0 to 6
b11	63486	1 to 5
b10	64510	Q to T
b9	65022	A to G
b8	65278	Caps shift to V

Using this information it is possible to write programs that will detect when a number of keys are pressed simultaneously. For example, the following program will read in the two groups of five keys that make up the top line of the keyboard, and display the resulting bit pattern:

```
10 LET D=IN 63486
20 GOSUB 1000
30 PRINT AT 5,10;
40 FOR I=8 TO 4 STEP -1
50 PRINT B$(I);
60 NEXT I
70 LET D=IN 61438
80 GOSUB 1000
90 PRINT B$(4 TO 8)
100 GOTO 10
```

Notice that subroutine 1000, given in Chapter 1, which converts decimal to binary, has to be included to make this program work. As the top row of keys includes the four arrow keys this could obviously be used in games and other programs that need movement control. However, notice that the two half rows of keys 'interact', so that pressing more than one key in each half at the same time can give false readings.

Now that the keyboard hardware and the principles behind its operation have been explained, you should be able to see that all the sophisticated keyboard features are produced by the Spectrum's software. Machine code routines in the BASIC ROM read the state of the keyboard. Taking into account any shift keys that have been pressed, they convert knowledge about which key is pressed into the

code that represents one of the five possible legends on or around the key. The software is also responsible for producing the auto-repeat facility and checking (once every fiftieth of a second) for the BREAK key.

When it is used as an input device the ULA also returns the state of the EAR cassette socket as b6 of the bit pattern. If the input from the cassette recorder is a high voltage state then b6 is a 1. Otherwise it is zero. The ULA will return the state of the EAR socket as b6, no matter what b8 to b15 are set to. So if you want to read the keyboard and the EAR socket then use one of the addresses given above. If you want to read the EAR socket independently of the keyboard then all the bits b8 to b15 should be set to 1, so the I/O port address that should be used is 65534. In normal use the state of b6 is used to decode the audio tones from the cassette recorder. However, it is possible to use it for other simple input tasks. For example, the following program will detect the start of a recording on the tape:

```
10 PRINT "PLAY TAPE"
20 IF IN 65534=255 THEN PRINT AT 2,0;
   "Silence":GOTO 20
30 PRINT "Sound started"
```

If you play a tape then this program will print 'Silence' until the first noise on the tape is detected. Notice, however, that as the EAR input is connected to the ULA via a low value capacitor, it cannot be used to monitor slowly changing voltages.

The expansion connector

Most of the signals used within the Spectrum are available from the edge connector at the back. This is usually used to connect the ZX printer, Microdrives, and Interfaces 1 and 2. However, it can be used to connect home-built peripherals, and the video signals can be used to drive a monitor. As the Spectrum manual gives very little information on the nature of the signals it is worth giving a list along with brief comments. A-side connections are on the component side of the board and B-side connections are on its reverse side.

The video signals available from 15B, 16B, 17B and 18B can be used to drive a monitor, and so improve the quality of the display that the Spectrum produces. The composite video signal on 15B is a direct connection to the output of the two transistors (emitter follower) that drive the UHF modulator, so this has enough power

1A	b15 of address bus	
2A	b13 of address bus	
3A	b7 of data bus	
4A	not connected	
5A	SLOT	
6A	b0 of data bus	
7A	b1 of data bus	
8A	b2 of data bus	
9A	b6 of data bus	
10A	b5 of data bus	
11A	b3 of data bus	
12A	b4 of data bus	
13A	INT	Z80 interrupt line; connecting this to +5 will stop the interrupts generated by the ULA
14A	NMI	Z80 non-maskable interrupt line; this interrupt isn't used by the Spectrum. A low pulse will cause BASIC to do a reset
15A	HALT	Z80 halt line which signals that a machine code halt instruction has been executed
16A	MREQ	standard Z80 control bus line
17A	IORQ	standard Z80 control bus line
18A	RD	standard Z80 control bus line
19A	WR	standard Z80 control bus line
20A	−5V	low current −5V supply
21A	WAIT	Z80 wait line which when held low will temporarily halt the Z80. A wait must not last for longer than about 1ms otherwise the dynamic memory will forget!
22A	+12V	smoothed 12V supply
23A	+12V	unsmoothed 12V supply
24A	M1	standard Z80 control bus line
25A	RFSH	Z80 memory refresh signal
26A	b8 of address bus	
27A	b10 of address bus	
28A	not connected	

1B	b14 of the address bus
2B	b12 of the address bus
3B	5V supply
4B	9V supply
5B	SLOT

6B 0 volts
7B 0 volts
8B CK Z80 system clock 3.5MHz
9B b0 of the address bus
10B b1 of the address bus
11B b2 of the address bus
12B b3 of the address bus
13B IORQGE holding this line high (i.e. +5V) will stop the ULA responding to I/O requests. With suitable circuitry it could be used to expand the number of I/O devices that the Spectrum can select
14B OV video ground
15B composite colour video signal
16B video Y signal
17B video V signal
18B video U signal
19B BUSRQ standard Z80 control bus line
20B RESET momentarily connecting this line to 0V will reset the machine just as if the power had been switched off and on
21B b7 of address bus
22B b6 of address bus
23B b5 of address bus
24B b4 of address bus
25B ROMCS connecting this to +5V will remove the BASIC ROM from the Spectrum's memory map.
26B BUSACK standard Z80 control bus line
27B b9 of address bus
28B b11 of address bus

More information on the connections described as 'standard Z80 control bus line' can be found in any Z80 manual.

to drive a monitor directly. The only problem is this video output is not the standard 75 ohm impedance, and most monitors will not work very well with it. The three colour signals on 16B, 17B and 18B are all unbuffered outputs from the ULA, and do not have enough power to drive a monitor directly. This makes a buffer amplifier essential, and to derive a standard RGB signal needs quite a complicated subtractor circuit. All in all the composite video signal on 15B is much easier to use! On many Spectrums some of these

video signals are not connected, and this simple fact explains why many attempts at driving monitors have failed! The solution is simple – inside the Spectrum, near the video circuits to the far left of the printed circuit board, are four links marked U, V, Y and VID. If the 'pads' are connected by a wire link, then the video signals will appear at the edge connector. However, if the pads are connected by nothing but a white line you will have to solder a wire link to make the signals appear.

The system diagram

Now that each section of the spectrum has been described it is time to give a complete block diagram of the system. You should be able

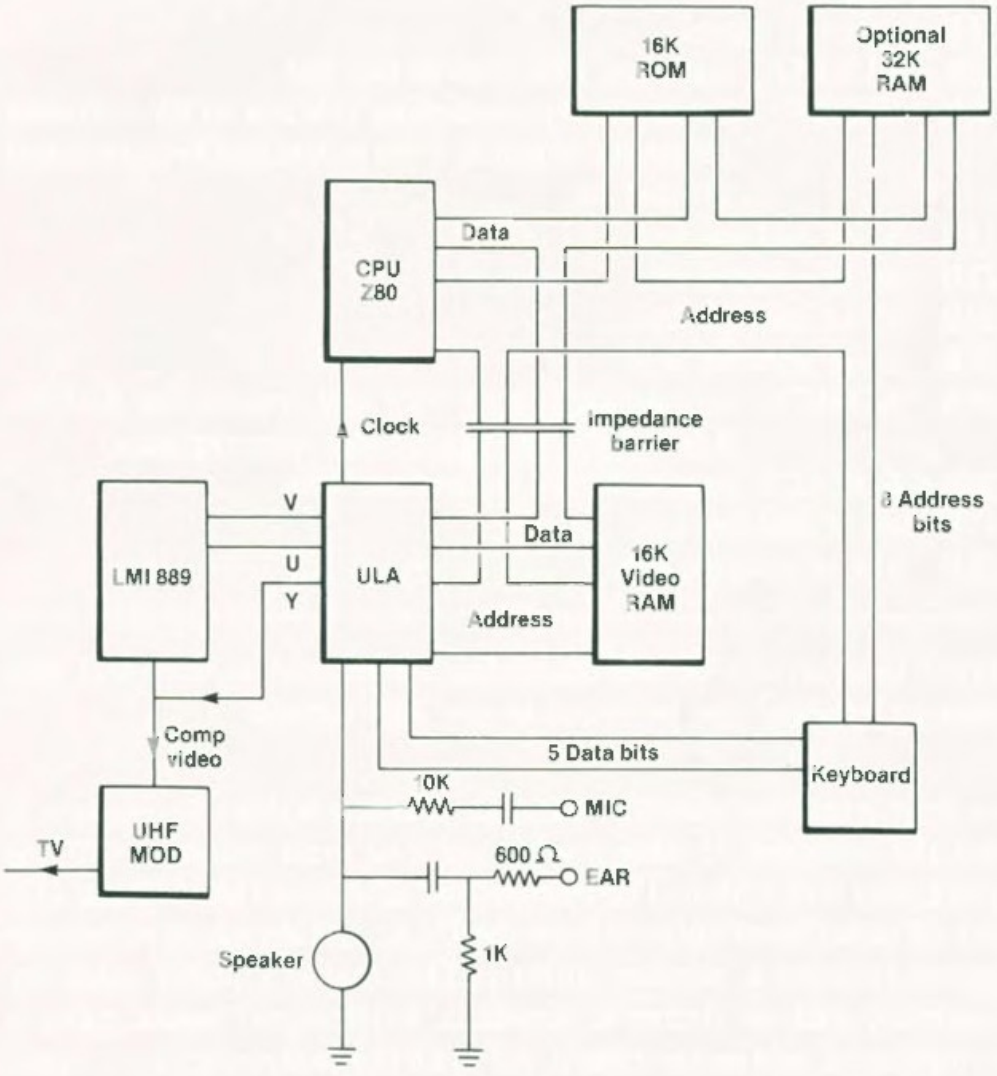


Fig. 2.6. Block diagram of the Spectrum.

to see all of the details that have been discussed in the previous sections in Fig. 2.6. Although the principles that lie behind the Spectrum's operation are interesting, the most important hardware features from the programmer's point of view are the I/O devices. The information on how the keyboard, loudspeaker and tape interface work will be used in later chapters to increase the range of things you can do with an unmodified Spectrum.

Chapter Three

Inside ZX BASIC

The subject of this chapter and the next two is the internal workings of ZX BASIC. There was little point in explaining the detailed workings of the Spectrum's hardware; similarly, there is little point in giving a complete listing of the Spectrum's BASIC ROM. Such a listing does indeed contain all the information you could ever want to know about ZX BASIC, but much of it will be irrelevant. If you are writing machine code then it is helpful to know something about the subroutines that are present in the BASIC ROM so you can make use of them; but if you are writing BASIC then it is more important to know how BASIC organises the memory that it uses. Knowledge of BASIC's general methods of obeying your commands can also suggest ways of using BASIC more economically and creatively.

The first part of this chapter describes the way that ZX BASIC divides the RAM into different areas, each used for a particular purpose. It gives an overview of the various sections, most of which are subsequently dealt with in greater detail. Then the many uses of the system variables area of memory are described. Chapter 4 considers how BASIC organises program lines and variables within memory, and Chapter 5 the method that ZX BASIC uses to extend the PRINT and INPUT commands to I/O devices other than the screen and the keyboard.

The memory map

When the Spectrum is first switched on, it goes through an initialisation sequence that determines the amount of memory available (normally 16K or 48K) and divides it up into a number of areas. These can be seen in the 'memory map' given in Fig. 3.1. Notice that some of the boundaries between areas of memory are

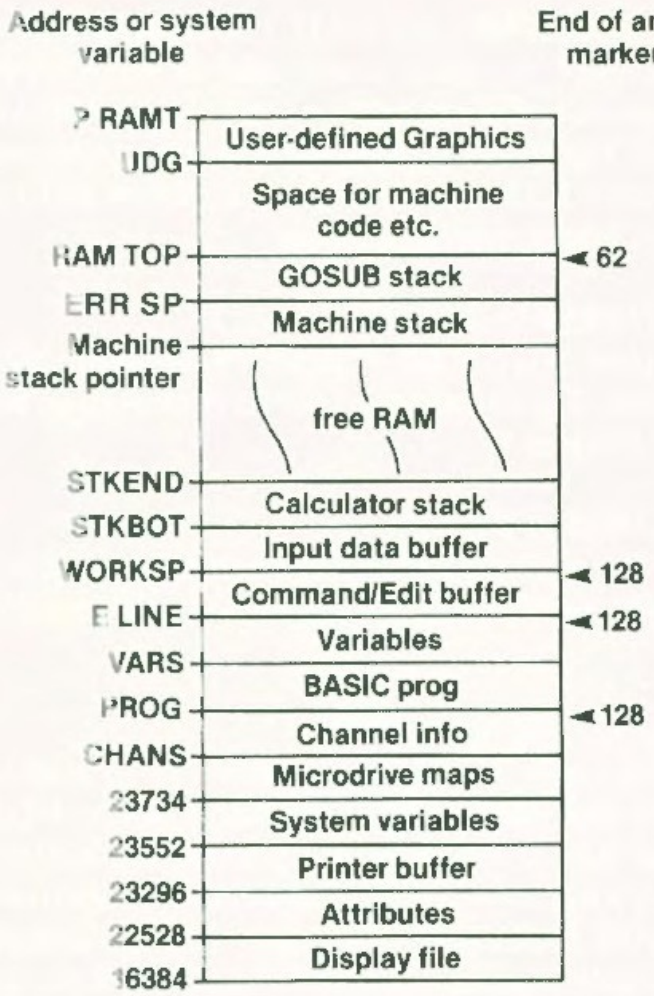


Fig. 3.1. Memory map for the ZX Spectrum.

fixed and others are variable. For example, the display file always starts at 16384 and ends at 22528, but where a BASIC program is stored in memory depends on how much space the Microdrive maps and the channel information have taken. Equally, the place where the area used to store variables starts depends on the size of the program area. The addresses where these movable areas of memory start (and occasionally where they stop) are stored in the system variables area of memory along with other data about the current state of the Spectrum. The idea of storing an address in memory is not difficult once you get used to it. For example, CHANS is a system variable that holds the address of the start of the channel information area. The CHANS system variable consists of two memory locations (remember a single memory location can only hold eight bits, and an address is 16 bits long) and their locations are 23631 and 23632. So if you want to know the start address of the channel information area you have to look at (PEEK) the contents

of memory locations 23631 and 23632. It is important to realise that ZX BASIC doesn't recognise names such as CHANS etc. If you want to gain access to the information stored in CHANS you have to use its address. The system variables area is such an interesting part of the Spectrum's RAM that it is given a section all to itself. The other areas are briefly described below.

Display File (16384 to 22527)
Used to store pixel data (i.e. ink or paper) for the entire 24-line by 32-character screen. The format used for its data storage is discussed more fully in Chapter 6.

Attributes (22528 to 23295)
Used to store the attributes of each character location in the entire 24-line by 32-character screen. This topic is also discussed more fully in Chapter 6.

Printer Buffer (23296 to 23551)
Used to hold a single line of 32 characters to be sent to the ZX Printer. Notice that the characters are stored as 8 by 8 dot patterns rather than as ASCII codes. Hence printing the contents of this buffer is simply a matter of transferring each complete row of dots to the printer. If the ZX Printer isn't in use, then this area of memory can be used to hold machine code USR functions. Refer to Chapter 7 for more details.

System Variables (23552 to 23733)
Used to hold a wide range of different values that reflect the current state of the Spectrum. This area is discussed at length later in this chapter.

Microdrive Maps (23734 to CHANS)
The Microdrive maps are used to store information concerning which sectors are free and which are used on the current tape cartridge. Of course, if the Microdrives are not in use this area doesn't exist, and CHANS is set to 23734.

Channel Information (CHANS to PROG−2)
This area is used to store data concerning which stream is associated with which channel. Streams and channels are discussed in Chapter 5.

BASIC Program (PROG to VARS−1)
Used to store the lines of text that make up a BASIC program. The program is not stored in the same form that it appears on the screen. Some parts of it are coded to save either memory space or time when run. Information on the storage of program lines is given in Chapter 4.

Variables (VARS to E LINE−2)
Used to store the variables created while a program is running. Notice that the variables area is only cleared just after the RUN command is given, so any variables created by a program exist until it or another program is run, or until the NEW or CLEAR commands are given. This is also dealt with more fully in Chapter 4.

Edit Buffer (E LINE to WORKSP−1)
Used to store a command or program line while it is being edited.

INPUT Data Buffer (WORKSP to STKBOT−1)
Used to store data typed in response to INPUT commands and for other miscellaneous data storage applications.

Calculator Stack (STKBOT to STKEND−1)
Used during the calculation of any string or arithmetic expression to store intermediate results. The workings of a stack are explained in the next chapter.

Machine Stack (stack pointer to ERR SP)
The machine stack is used by the Z80 to store temporary data etc. It is not possible to find the lowest address used by the machine stack from BASIC. The reason for this is that the end-of-stack address is permanently stored in a Z80 register called the stack pointer.

GOSUB Stack (ERR SP+1 to RAMTOP)
This is used to store the line numbers used by RETURN instructions. The operation of this stack is mixed up with the operation of the machine stack, so it is not easy to alter return addresses by POKEs. The detailed workings of this stack are explained in Chapter 4.

User Defined Graphics (UDG to P RAMT)
This area is used to store the dot patterns associated with the user-defined characters. We return to this topic in Chapter 6.

Notice that the Spectrum uses memory 'from both ends'. The program and variable storage both start from the low address end of the RAM, and expand upwards as the need occurs. The machine stack and GOSUB stack both start at the high address end of the RAM and work their way down. This means that the current free RAM is to be found between STKEND and the address in the Z80's internal stack pointer.

System variables

The use of the system variables to hold the addresses of the boundaries between the different areas of memory has already been introduced. In fact the system variables area of memory is used to hold many different pieces of information that can be very useful to the BASIC programmer. A full list of system variables in order of address can be found in Chapter 25 of the Spectrum Manual. However, they are better classified according to what they do rather than by their location in memory. There are five groups of system variables:

(1) the RAM boundary variables
(2) the keyboard state variables
(3) the system state variables
(4) other I/O variables
(5) the video display variables

To avoid repetition, the other I/O variables are described in Chapter 4 and the video display variables in Chapter 5. The use of the other three groups is described below.

Before moving on to the uses of these system variables we must first look at a problem common to all the groups: how a 16-bit address is stored in a pair of eight-bit memory locations. At one level, the answer is obvious. If you list the bits of the 16-bit number as b0 to b15 then the storage problem can be solved by using one memory location to hold b0 to b7, and the other to hold b8 to b15. The memory location that holds b0 to b7 is called the 'least significant' byte and the memory location that holds b8 to b15 is called the 'most significant byte'. In the Spectrum, with one or two exceptions, the least significant byte is stored in the memory location with the lower address. So if memory location N holds bits b0 to b7 of the memory location, N+1 holds b8 to b15. To 'reconstruct' the decimal equivalent of a 16-bit number from its two

eight-bit halves is quite easy. If you PEEK the least significant byte, the decimal value returned is correct, but PEEKing the most significant byte returns a value that is too small by a factor of 256. The reason for this is not difficult to see if you consider the weights given to each bit in the binary-to-decimal conversion carried out by PEEK. For the least significant byte the weights used are 128, 64, 32, 16, 8, 4, 2, 1 and these are correct for b7 to 0 of a 16-bit binary number as well as an eight-bit number. However, the same weights are used for the most significant byte, i.e. bits b15 to b8. These should in fact be 32768, 16384, 8192, 4096, 2048, 1024, 512, 256; and these are bigger than the first set by a factor of 256. Thus if a 16-bit number is stored in the two memory locations N and N+1 the following user-defined function will return its decimal equivalent:

```
DEF FND (N)=PEEK(N)+256* PEEK (N+1)
```

Similarly, if you want to POKE the 16-bit number that corresponds to the decimal number V into the two memory locations N and N+1 then use:

```
POKE N, V-256*INT(V/256)
POKE N+1,INT(V/256)
```

The expressions V−256*INT(V/256) and INT(V/256) occur so often in this type of application that it is worth defining two user-defined functions for them. The expression V-256*INT(V/256) finds the remainder after dividing V by 256, and INT(V/256) is simply the whole number of times that 256 will divide V. The function

```
DEF FNH(V)=INT(V/256)
```

will return the decimal equivalent of the most significant byte of V and

```
DEF FNL(V)=V-INT(V/256)*256
```

will return the decimal equivalent of the least significant byte of V.

Using the RAM boundary variables

All of the RAM boundary variables have been described in connection with the memory map given earlier. In this section some of their possible uses to the ZX BASIC programmer are described.

The most obvious use for the RAM boundary variable is to find out how much memory is being used and for what. For example, the

difference between the address stored in PROG and the address stored in VARS will tell you how much memory is being used by a BASIC program. The following subroutine will PRINT the amount of memory allocated to the program and variables, and the amount of free memory:

```
9000 DEF FNp(N)=PEEK(N)+256*PEEK(N+1)
9010 PRINT "Program: ";FNp(23627)-FNp(23635)
9020 PRINT "Variables: ";FNp(23641)-FNp(23627)
9030 PRINT "Free: ";FNp(23613)-FNp(23653)
9040 RETURN
```

The system variables used in the subroutine are:

23627 VARS
23635 PROG
23641 E LINE
23613 ERR SP
23653 STKEND

The estimate of the amount of free memory produced by this program does not include the memory used by the machine stack, so it is too large by around 10 memory locations. The Spectrum's ROM contains a machine code routine that will return the exact amount of free space available, but there is no guarantee that its location will remain fixed in future issues of the ROM. However, you might like to try replacing line 9030 with:

```
9030 PRINT "Free: ";65536-USR 7962
```

This should return a very similar result.

Another use of the boundary variables is to find the start of the program or variables area so that they can be examined. An example of this is given in the next chapter.

The keyboard state variables

The keyboard state variables can be used to control the way that the keyboard behaves. This can be very useful in applications programs that need to tailor the keyboard's response for user data entry. The system variables concerned are:

KSTATE – 8 memory locations from 23552 to 23559
This is used to record which keys have been pressed for the purpose

of controlling the auto-repeat facility, and is not really of any use to the BASIC programmer.

LAST K - 1 memory location at 23560

This memory location holds the code of the last key to be pressed. It is updated every 1/50th of a second unless the Spectrum is loading or saving to tape, or making a sound. The value in LAST K is used by the INPUT routine to make sure that no key-presses are lost. In this sense it acts as a single character type-ahead buffer! To see LAST K working try

```
10 PRINT CHR$(PEEK(23560))
20 GOTO 10
```

which prints the character corresponding to the code stored in LAST K. Notice that INKEY$ reads the keyboard directly and so bypasses LAST K.

REPDEL – 1 memory location at 23561 and REPPER – 1 memory location at 23562

These two keyboard variables need to be considered together because they both control aspects of the auto-repeat. REPDEL sets the time that a key has to be held down before it starts to repeat, and REPPER is the rate at which the key auto-repeats. You can POKE values into these variables to alter the way the keyboard behaves. For example, to produce a keyboard with an almost instantaneous repeat rate use:

```
POKE 23561,1:POKE 23562,1
```

High repeat rates are useful when keyboard input is used to control the movement of screen graphics during games etc. Very low repeat rates produced by POKEing both variables with zero are useful for novice computer users.

RASP – 1 memory location at 23608 and PIP – 1 memory location at 23609

These two memory locations control the length of characteristic sounds associated with the keyboard. The value in RASP alters the duration of the warning tone that accompanies errors, such as typing in lines that are too long. The value in PIP alters the duration of the keypress tone. Normally this is so short that the tone is reduced to a click. By experimenting with PIP you can achieve a variety of sounds.

The system state variables

The system state variables are used by ZX BASIC to keep track of the current state of the machine. Most of these variables are of little use to the BASIC programmer and cannot be altered. Included in this group, however, is the familiar three-location timer starting at 23672. It counts the number of TV frames that have been displayed since the Spectrum was switched on. The function

```
DEF FNt()=(PEEK(23672)+256*PEEK(23673)+
65536*PEEK(23674))/50
```

returns the time in seconds since the Spectrum was switched on. To zero the timer use

```
POKE 23674,0:POKE 23675,0:POKE 23676,0
```

There are two other variables included in this group which may be of use to Z80 assembly language programmers.

ERR NR – 1 memory location at 23610
Holds one less than the error report code. This could be used as part of an implementation of an ON ERROR GOTO type of statement to extend ZX BASIC, but this is not an easy project.

ERR SP – 2 memory locations at 23613
This contains the address of a pair of memory locations on the machine stack. These locations contain the address of the machine code routine within the ZX BASIC ROM that is jumped to when an error occurs. If you decrease the contents of this pair of locations by two you will find that the BREAK key is disabled, but any subsequent errors that occur will cause the machine to crash. It is possible for the Z80 assembly language programmer to alter the error return address to replace the standard error handling by a new error routine. However, this is not as easy as it first seems, since the Spectrum changes the error return address as it runs to allow for different types of error handling. For example, during an INPUT command a data entry error doesn't crash the machine, it simply causes the input editor to ask for the input over again. This is a difficult, challenging but possible project!

The shifting memory

As already described, many of the areas of memory change their size as a program is entered or run. For example, each time a line of BASIC is entered the program area increases in size. What is less obvious is that each time a memory area changes its size all the memory areas above it have to be moved, and all the system variables that mark the boundaries have to be changed. For example, if space is made within the input data area then the calculator stack has to be moved up. All this shuffling of memory is taken care of automatically by ZX BASIC, but it is worth knowing a little of how it is done.

Whenever space of x bytes is to be made within a memory area, all the memory above the area and below STKEND is moved up. Then the fifteen system variables starting at VARS (23627) and ending at STKEND (23653) are examined one by one. If the system variable contains an address that is above the area of memory that is being extended, then the address is increased by x. The opposite process is carried out when an area of memory is being reduced by x bytes. In other words, all the memory above the area is moved down by x bytes, and the fifteen system variables that contain addresses above the area are reduced by x.

This shifting and adjustment of system variables has to be taken into account by any Z80 assembly language programs that alter the standard position of any memory area, or the value of any system variable. For example, in Chapter 5 the shifting of memory causes trouble if the area of memory 'pointed at' by CURCHL is positioned above the input editing area and above STKEND. Although the area of memory isn't moved, because it is above STKEND, its system variable contains an address that is above the input editing area, so it is adjusted as if the area *had* been moved. The result is an irrecoverable system crash! Another consequence of the memory shifting is that you cannot be sure that anything stored above 23734 will stay at a fixed location in memory. Maybe you knew where it was at the start of the program, but that doesn't mean it will stay there for the entire course of the program. The moral is to find every item, variable, program line etc. each time you need it, unless you know it couldn't possibly have moved. Some examples of finding objects in memory are given in the next chapter.

Conclusion

This chapter has described the overall layout of the Spectrum's memory in some detail. However, discussion of and examples involving many of the items introduced have been postponed to later chapters, where their relation to other topics can be explored. If, in later chapters, you lose a sense of where everything is, then use the memory map in this chapter as a guide.

Chapter Four

The Structure of ZX BASIC

ZX BASIC is a completely new implementation of BASIC, and it has many new features. In particular its string handling is a complete break from the methods used by the older and clumsier Microsoft standard. The topic considered in this chapter is not the outer appearance of ZX BASIC, but how it organises and uses memory to implement some of the more important facilities it provides. The most complete statement of the way ZX BASIC works is, of course, contained in a listing of the ZX BASIC ROM. However, to a great extent this is overprovision. Most of the ROM is concerned with the detailed implementation of arithmetic, functions etc. These sections might be of interest, but are generally of little practical use.

The best way to understand the workings of ZX BASIC is to study the way that it organises and uses memory, and the principles that lie behind its implementations of GOTOs, GOSUBs, FOR loops etc. This knowledge makes it easier to understand the overall layout of a ROM listing; in most cases it is also sufficient to make consulting a ROM listing unnecessary. To demonstrate this, a number of practical examples of manipulating the program and variables area, and altering the way that ZX BASIC works, are given. These examples are all written in ZX BASIC to make them as accessible as possible; but if you already know or are learning Z80 assembler, then they would all benefit from the extra speed that would result from being rewritten in Z80 assembler.

The format of variables - a variable dump program

Chapter 24 of the Spectrum Manual gives a great deal of information about the format of the different types of variable created by ZX BASIC in the variables area of memory. This said, it

is still worth summarising the information presented to show the system that lies behind all the formats.

The six different types of variable in ZX BASIC are all stored in the variables area of memory, starting with a single byte which serves both to identify the type of variable and to store the first (and possibly only) character of its name. The way these two pieces of information are packed into a single byte is not difficult to understand. ZX BASIC regards upper and lower case characters as identical when naming variables, so the first letter of a variable name can always be stored as a lower case character. Using the full ASCII code for the 26 lower case letters is perfectly possible, but it uses up a whole memory location (8 bits) when only 5 bits are necessary. It is much more efficient to store a number in the range 0 to 25 to indicate which of the 26 letters a variable name starts with. It also frees three of the bits in a memory location to store a *variable type code*. Thus the first byte of each variable has the following format:

b7 b6 b5 b4 b3 b2 b1 b0

type code	letter code

The type codes used are:

2 string variable
3 numeric variable with a single letter name
4 numeric array
5 numeric variable with a multiple letter name
6 character array (i.e. a dimensioned string)
7 index variable (i.e. a variable used in a FOR loop)

Notice that the type codes are converted to a three bit binary number, and the resulting bit pattern is used to set b7 to b5 respectively. For example, if the type code is 5, the three-bit binary equivalent is 101, and thus b7=1, b6=0 and b5=1. The ASCII code of the first letter of the variable name can be constructed from the value of b4 to b0 by simply adding 96. Thus if A holds the address of the first memory location used to store a variable, the following function

```
DEF FNt(A)=INT(PEEK(A)/32)
```

will return the value of the variable type code as given in the above table and

```
DEF FNc$(A)=CHR$(PEEK(A)-FNt(A)*32+95)
```

will return the first letter of the variable name. (Both functions make use of techniques of BASIC bit manipulation described in the last chapter.)

What follows the first byte of each variable depends on what type of variable it is. These data formats are given in detail in Chapter 25 of the Spectrum Manual, and are reproduced with additional comments in Fig. 4.1. Although a knowledge of these formats is

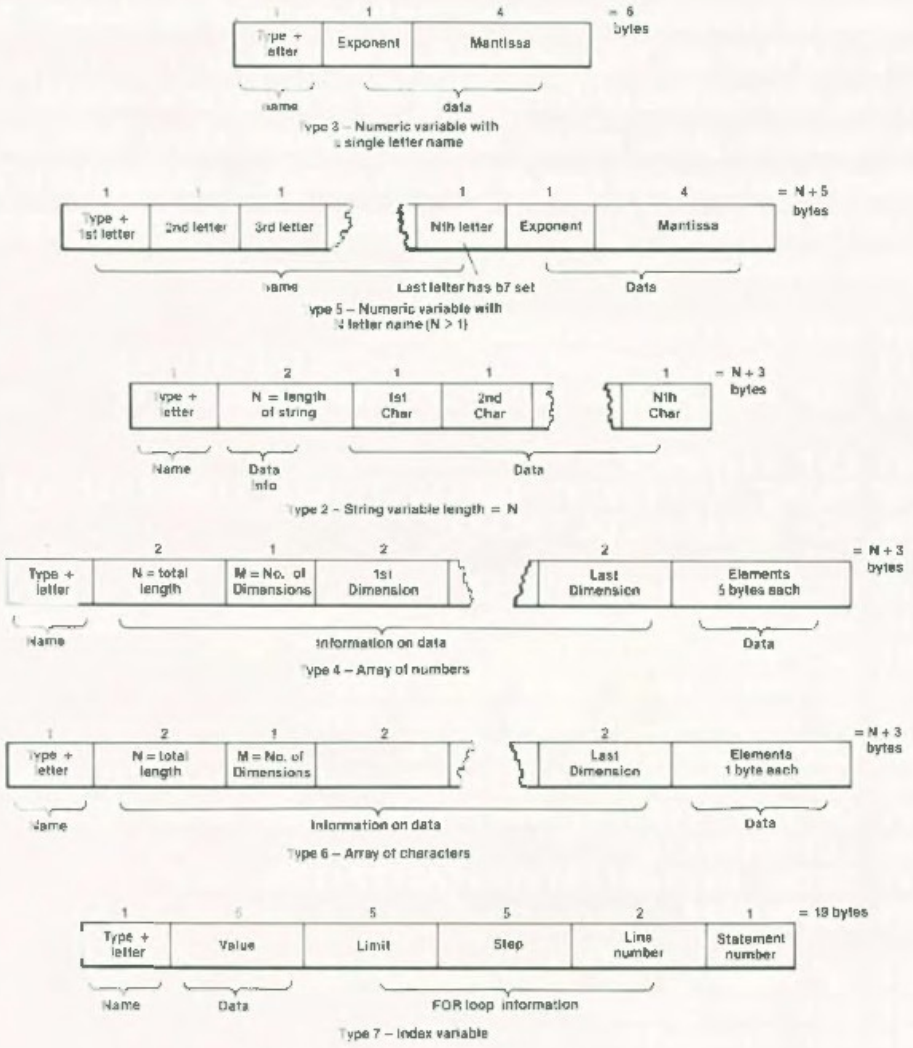


Fig. 4.1. Data formats for variables in ZX BASIC.

important to the assembly language programmer, the ZX BASIC programmer can use the standard functions VAL and VAL$ to find out what the contents of a variable are. For example, if N$ contains a non-array variable's name, then

```
PRINT VAL(N$)
```

will print its contents if it is a numeric variable, and

```
PRINT VAL$(N$)
```

will print its contents if it is a string variable. Similar expressions can be used to print any element of an array variable. For example, if N$ contains the name of a single dimensioned numeric variable then

```
PRINT VAL (N$+"("+STR$(I)+")")
```

will print the contents of element I. The idea behind printing elements of arrays is to construct the full name of the element as a string and then use VAL or VAL$ to evaluate it.

Using this method of discovering the contents of a variable, the two functions given earlier, and information about how many memory locations each type of variable occupies, it is possible to list all the variables used by a program. Such a variable dump program is given below.

```
9100 DEF FNt(A)=INT(PEEK(A)/32)
9110 DEF FNc$(A)=CHR$(PEEK(A)-FNt(A)*32+96)
9120 DEF FNv()=PEEK(23627)+256*PEEK(23628)

9130 LET V0=FNv()
9140 PRINT "Variable";TAB(15);"Type";TAB(25);"Value"
9150 DIM N$(15):DIM T$(10)

9200 IF PEEK(V0)=128 THEN STOP
9210 LET I0=1:LET N$="":LET T$="Numeric":LET N$=""
9220 LET N$(I0)=FNc$(V0)
9230 IF FNt(V0)=3 THEN GOTO 9280
9240 IF FNt(V0)<>5 THEN GOTO 9300
9250 LET V0=V0+1:LET I0=I0+1
9260 LET N$(I0)=CHR$(PEEK(V0)-INT(PEEK(V0)/128)*128)
9270 IF INT(PEEK(V0)/128)*128=0 THEN GOTO 9250
9280 LET V0=V0+6
9290 PRINT N$;TAB(15);T$;TAB(25);VAL(N$)
9295 GOTO 9200

9300 IF FNt(V0)<>7 THEN GOTO 9350
9310 LET T$="Index"
9320 LET V0=V0+19
9330 GOTO 9290

9350 IF FNt(V0)<>2 THEN GOTO 9400
9360 LET T$="String":LET N$(2)="$"
9370 LET V0=V0+PEEK(V0+1)+256*PEEK(V0+2)+3
9380 PRINT N$;TAB(15);T$;TAB(25);VAL$(N$)
9390 GOTO 9200

9400 IF FNt(V0)=6 THEN LET N$(2)="$"
```

```
9410 LET T$="Array"
9420 LET I0=0
9430 PRINT N$;TAB(15);T$;TAB(25);"DIM(";
9440 PRINT PEEK(V0+4+I0*2)+256*PEEK(V0+5+I0*2);
9450 LET I0=I0+1
9460 IF I0<>PEEK(V0+3) THEN PRINT ",";:GOTO 9440
9470 PRINT ")"
9480 LET V0=V0+3+PEEK(V0+1)+256*PEEK(V0+2)
9490 GOTO 9200
```

The first part of the program defines three useful functions. FNt and FNc$ have already been described and FNv returns the current start address of the variables area of memory. Lines 9130 to 9150 print a heading, initialise the variable V0 which is used to mark the current position in memory, and dimension two arrays used in the program. N$ is used to build up the name of each variable and T$ is used to hold a description of its type. The rest of the program is in the form of a large loop starting at line 9200. Line 9200 tests for a value of 128, which is used to mark the end of the variables area. Lines 9210 to 9295 build up the name of a numeric variable in N$ and then print its value at line 9290. If the variable is type 3 then the single letter already in N$ is the variable's name, and line 9230 passes control to line 9290 which prints the variable's details. If the type is 5 then the first character is followed by a sequence of letters making up the variable's full name (see Fig. 4.1). Lines 9250 to 9270 extract each character in turn, storing it in N$. The end of the variable's name is marked by the value of b7, which is 0 for all the characters but the last. Line 9280 adds 6 to V0 to make it point at the start of the next variable.

If the variable type is 7 then control passes through line 9300. Afterwards, line 9320 adjusts V0 to point to the next variable. The details of the index variable are printed by line 9290.

If the variable type is 2 then control passes through line 9350. Line 9370 sets V0 to point at the start of the next variable by adding the length of the string to it (see Fig. 4.1). Line 9380 prints the current data stored in the string using the VAL$ function as described earlier.

Finally, if the variable type is 4 or 6 then the variable is an array. In this case the program doesn't attempt to print the data in the array because this might be rather a lot! Instead the dimensions of the array are printed. Line 9400 adds a "$" to the name of the array if it is a character array. Apart from this, both types of array can be treated in the same way, because their dimension information is stored in the same way. The number of dimensions is contained in the fourth location of the array and this is PEEKed in line 9460 to see if the

values of all of the dimensions have been printed. Line 9440 will print the value of a single dimension, and I0 is used to count the number of values printed so far. Finally, line 9480 uses the total length of the array to update V0 so that it points to the start of the next variable.

If you add this program to the end of one of your own, then GOTO 9100 will print a list of all the variables that your program is using plus I0, V0, N$ and T$ which are used by the variable dump program itself. The arrays N$ and T$ are used in preference to strings because the variables area changes as the number of characters in a string is increased or decreased. If a string changed its size while the memory dump program was running then the location of all the variables above it would be changed, and V0 would no longer necessarily point to the start of a variable. However, a character array is fixed in size, and using it doesn't cause the variables area to be rearranged. You can add other facilities to this variable dump program, such as printing the amount of memory that each takes, but beware of using any strings within the dump program itself, or things will go wrong!

The numeric data formats

The way that numbers are stored within a computer is a very technical subject but there are two basic methods – *integer storage* and *floating point storage*. Integer storage gives a limited range of numbers but is fast and easy when used in arithmetic. It is essentially the simple binary representation of numbers that we have been using since Chapter 1, extended to include both positive and negative numbers. Floating point storage can be used to represent a very wide range of numbers, but floating point arithmetic is quite slow compared to integer arithmetic. Floating point storage is based on a binary equivalent of the decimal exponential notation used on many calculators.

Many versions of BASIC provide two different types of numeric variable – integer for whole numbers, and real for numbers with a fractional part. When to use each type of variable is left to the prgrammer's discretion. ZX BASIC also provides both types of storage, but within the one type of numeric variable. Which form of storage is used is decided by ZX BASIC. If a number will fit into the range of the integer storage provided, then it is stored as an integer. Otherwise it is stored as a floating point number. By this mechanism

the programmer gets the best of both types of storage, but need never worry about how values should be stored. The details of both types of storage are well described in Chapter 24 of the Spectrum Manual.

The dynamic management of variables

The previous sections describe the format used to store variables. However, there is another aspect to variable storage that concerns us. As new variables are created and strings altered, the variables area is rearranged. How this rearrangement is achieved can affect the efficiency of programs, so the details of the dynamic management of the variables area are important.

The variables area is emptied by a RUN or CLEAR command, and variables are created as and when they are encountered. To avoid moving things around too much, new variables are added to extend the variables area upward. Thus, initially at least, variables are stored in the variables area in the order that they are created. Imagine the difficulty of adding one character to the end of an existing string variable. If the string variable was created early in the program, then each time a single character was added all the other variables stored above it would have to be moved up by one memory location. This suggests that a program like

```
10 LET A$=""
20 DIM M(1000)
30 LET A$=A$+"X"
40 GOTO 30
```

would run faster if the array was dimensioned, i.e. created, before the string variable A$ (by swapping the order of the first two lines of the program). In the listed program, approximately 5000 memory locations have to be moved each time an "X" is added to the string. If the array were defined first, then no variables would have to be moved to add a single character to A$. This sort of problem causes many versions of BASIC to slow down when handling large arrays and strings – but not ZX BASIC! So, both versions of the above program run at roughly the same speed on the Spectrum!

The reason for this is that ZX BASIC uses an interesting method of managing the variables area. Each time a string variable occurs on the left hand side of a LET statement its old value is destroyed by moving the variables area down to 'close up' the memory space that

it occupied. Then it is re-created at the top of the variables area just as if it were a completely new variable! In short, a string variable is re-created each time it occurs on the left hand side of a LET statement. This re-creation has two effects. Firstly, unlike other systems, there are no old versions of strings hanging around in the variables area; hence there is no need to stop calculating and perform 'garbage collection' now and again. Secondly, the most recently used string variable is always at the top of the variables area, and the most frequently used string variables tend to be close to the top of the variables area. This minimises the number of moves made and the number of locations affected by each move due to string handling. You should now be able to see that the program given above, which adds a single letter to the variable A$, will only result in the array being moved once to bring the string to the top of the variables area.

The same system of management is used when an array is defined. When an array is dimensioned, an existing version of the array is removed by moving the variables above it down to close up the space that it occupied. Then a new array is created at the top of the variables area. This means that it is possible to dimension arrays more than once in ZX BASIC, whereas other versions of BASIC treat arrays as fixed-size variables.

How ZX BASIC is stored

Each line of ZX BASIC is stored using the format shown in Fig. 4.2. The first two bytes of each line contain the line number, stored in the

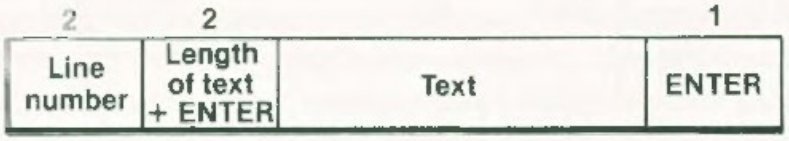


Fig. 4.2. Format of a BASIC line.

reverse order to most other numbers, i.e. with the most significant byte first. The line number is used to determine where GOTOs and GOSUBs transfer control to, and to determine where new lines are inserted in the program. Lines are stored in order of ascending line number. The second two bytes are the length of the text, including the enter character that marks the end of each line. These two bytes

are stored in the usual order, and are used to find the location in memory of the start of the next line.

If A is the address of the start of a BASIC line then the function

```
DEF FNL(A)=256*PEEK(A)+PEEK(A+1)
```

returns its line number. The function

```
DEF FNn(A)=PEEK(A+2)+256*PEEK(A+3)+A
```

will return the start address of the next line number. You can detect the end of a program when FNn(A) is equal to the contents of the system variable VARS. As well as examining line numbers, you can also change them by POKEing values. Although ZX BASIC will only accept line numbers in the range 1 to 9999, it will work with numbers in the range 0 to 61439. It will work in the sense that GOTOs and GOSUBs will correctly transfer control to line numbers in the larger range, but the editor will only allow you to edit line numbers in the smaller range. This anomaly can be turned to advantage by changing the first line number of a program to 0, thus making the line un-deletable. There are more sophisticated methods of using these semi-legal line numbers to add protection to programs but once you know about them they are very easy to defeat.

The text portion of a program line is stored exactly as it was typed from the keyboard, with a few exceptions. Firstly, any keywords within a line are stored as single bytes corresponding to their character codes, as given in Appendix A of the Spectrum manual. Thus GOTO is not stored as the four separate letters 'G', 'O', 'T' and 'O' but as the single-byte code 236. Secondly, all numeric constants are stored within the line in two different forms: as the string of digits typed in from the keyboard, and as a five-byte number in the format used for a numeric variable. The string form of the number is used when listing, and the five-byte internal form is used by ZX BASIC when the program is running, to save time converting constants to the internal format that all ZX calculations use. Character code 14 is used to indicate that a five-byte floating point number follows, and this is used by the LIST routine in the ZX BASIC ROM to skip over internal formatted numbers in listings. Five-byte floating point numbers can occur in other places, as well as following a numeric constant, so always look for code 14 when you scan a BASIC line.

A keyword finder

As an example of using information about the internal format of ZX BASIC, the following program searches the program area and prints out the number of any line that contains the keyword in C$.

```
10 INPUT C$
20 GOSUB 9500
30 STOP

9500 DEF FNp()=PEEK 23635+256*PEEK 23636
9510 DEF FNv()=PEEK 23627+256*PEEK 23628
9520 DEF FNL(A)=256*PEEK A+PEEK(A+1)
9530 PRINT C$;" AT "
9540 LET S=FNp()
9550 LET F=FNv()
9560 LET L=FNL(S)
9570 LET S=S+4
9580 IF S>=F THEN RETURN
9585 LET C=PEEK S
9590 IF C=13 THEN LET S=S+1:GOTO 9560
9600 IF C=14 THEN LET S=S+5:GOTO 9580
9610 IF C<>CODE(C$(1)) THEN LET S=S+1:GOTO 9580
9620 PRINT "line ";L
9630 LET S=S+1
9640 GOTO 9580
```

The subroutine starting at 9500 does all the work of looking for the keywords in C$. Lines 9500 to 9520 define three useful functions. FNp returns the start address of the program area, FNv returns the start address of the variables area and FNL returns the line number of the line starting at address A. Lines 9580 to 9640 form a loop that scans the program area line by line and character by character looking for character codes that match CODE(C$(1)). Line 9590 detects ENTER characters that mark the end of each program line, and line 9600 detects code 14, which indicates that the next five bytes are the internal form of a numeric constant, and should be skipped.

This simple program is of great practical use in checking that all GOSUBs and GOTOs are correct. (Note that to enter a keyword such as LET, first enter THEN to get the cursor into K mode, then enter the keyword, then delete the THEN.) You can even search for all the lines that use variables starting with a particular letter, by entering a single letter instead of a keyword. However, if you want to search for a variable with more than one letter in its name, the program will have to be extended to match each letter against the contents of memory.

As another example of how the keyword search subroutine can be used to extend the Spectrum's facilities, consider the following simple changes

```
9620 POKE 23625,L-INT(L/256)*256
9630 POKE 23626,INT(L/256)
9640 STOP
```

These new lines POKE the system variable E PPC with the line number of the first line to contain the keyword stored in C$. As E PPC is used to store the position of the editing cursor, this routine will move the editing cursor to the first occurrence of the keyword in C$. By adding a 'search from last position' option, this routine could easily be extended so that the editing cursor could be quickly positioned anywhere in a program.

A line renumber program

Renumbering a ZX BASIC program looks easy at first sight. All you have to do is scan through the program area, altering the two bytes at the start of each line that holds its number. The trouble is that this ignores the changes that must be made to line numbers quoted as part of GOTOs and GOSUBs. It is not difficult to think up a number of possible algorithms that would adjust the GOTO and GOSUB line numbers, but all of them involve scanning through the entire program and searching for every occurrence of the keywords GOTO and GOSUB. Such an algorithm in a ZX BASIC program would make it very slow to use.

As a compromise, the following subroutine renumbers all the lines of a program, ignoring the GOTO/GOSUB problem, but prints a list showing the correspondence between the new and the old line numbers so they can be corrected by hand.

```
9700 LET P=FNp()
9710 INPUT "Start number ";S0
9720 INPUT "Step size ";I0
9730 PRINT "OLD";TAB(10);"NEW"
9740 LET L=FNL(P)
9750 IF L>9000 THEN STOP
9760 PRINT L;TAB(10);S0
9770 POKE P,INT(S0/256)
9780 POKE P+1,S0-INT(S0/256)*256
9790 LET S0=S0+I0
9800 LET P=P+4+PEEK(P+2)+256*PEEK(P+3)
9810 GOTO 9740
```

Line 9700 sets P to the start of the program area using the function FNp defined in the last section. Lines 9710 and 9720 get the line number that the renumbered program should start with, and the step from one line number to the next. Line 9740 gets the old line number into L using the function FNL defined in the last section. The new line number is POKEd into the correct place in the line by lines 9770 and 9780. Line 9790 increases the new line number by the step size, and line 9800 adjusts P to point to the start of the next line, by adding the contents of the two locations that hold the length of the text part of the line. The renumber stops when the old line number reaches 9000 to avoid renumbering the renumber program or any of the other programs given in this and earlier chapters.

GOTO

The ZX BASIC GOTO statement works in much the way you would expect, but there are a few special features that are worth taking into account. When a GOTO is encountered, the program area is searched for the first line number that is equal to or greater than the line number used in the GOTO. If one is found, then control is passed to that line. If such a line isn't found then the program ends with a normal report code. This means that unlike other versions of BASIC, it is impossible to cause an error with a GOTO statement in ZX BASIC. In some ways this form of GOTO is an advantage, in others it can be a serious problem. For example, suppose the line GOTO 4000 occurs in a program without a line 4000, and the first line larger than 4000 is 5000. In this case the GOTO 4000 transfers control to line 5000, and the program may work as the programmer intended. Now suppose that at a later date the programmer innocently adds a new section starting at line 4500. The result is that the GOTO now transfers control to 4500, and the program may not work. The task of finding what has gone wrong is very difficult, because the cause of the trouble – the incorrect GOTO – is part of the program that hasn't been changed! The moral is always to make sure that GOTOs (and GOSUBs) transfer control to line numbers that exist. A second unusual feature of ZX BASIC is the way that the line number quoted in a GOTO (or a GOSUB) can be a numeric expression. For example, in ZX BASIC

```
GOTO 200+10*4
```

has the same effect as GOTO 240. This can be used to advantage in a

number of ways. For example, one of a number of routines can be selected according to the value stored in a variable using

```
GOTO L(I)
```

where L is an array containing the line numbers of the start of each routine, and the value of I governs which one control is passed to. That is, if I contains 1 the routine starting at L(1) will be jumped to, and likewise for other values of I. (This will, of course, also work with GOSUB.) In other versions of BASIC this facility is called a 'computed GOTO' and is usually written

```
ON I GOTO L1, L2, L3 ...
```

where L1, L2 etc are the line numbers that are jumped to when I is 1, 2 ... respectively.

Another use for expressions as part of GOTOs (or GOSUBs) is to make programs slightly more readable. If part of your program, starting at 3123 for instance, reads in data for further processing, then an instruction like

```
60 GOTO 3123
```

does the job of getting data but conveys nothing about what is going on to someone reading the program. However, if you define a variable with an appropriate name to hold the line number of the start of the routine, then GOTOs (and GOSUBs) become much more readable. For example

```
10 LET READDATA=3123
     .   .   .
     .   .   .
60 GOTO READDATA
```

ZX BASIC has the ability to handle more than one statement per line, using the colon as a separator. This is a very useful facility, but GOTOs can only transfer control to the start of multi-statement lines by way of line numbers. In fact ZX BASIC works with a line number, and a statement number within the line, that can be used to pinpoint any statement in a program, even if it is part of a multi-statement line. For example

```
1203 PRINT "1":PRINT "2":PRINT "3"
```

is a multi-statement line. The PRINT "1" command is line 1203 statement 1, PRINT "2" is line 1203 statement 2, and so on. Although there is no 'GOTO line number, statement number'

command that will transfer control into a multi-statement line, it is not difficult to produce one. The pair of system variables NEWPCC and NSPCC are used to hold the line number and statement number within the line that control is to be passed to. You can force a jump by POKEing NEWPCC with the desired line number and then POKEing NSPCC with the statement number within the line. For example, try:

```
10 PRINT 1;:PRINT 2;:PRINT 3;
20 LET L=10:LET S=2:GOTO 9800

9800 POKE 23618,L-INT(L/256)*256
9810 POKE 23619,INT(L/256)
9820 POKE 23620,S
```

If you run this program, you will see 123 printed followed by 23 repeating over and over again until you press BREAK. Routine 9800 will transfer control to line L and statement number S, so line 20 is equivalent to GOTO line 10, statement 2. Notice that routine 9800 should not be jumped to by a GOSUB because the transfer of control it creates would stop a RETURN from ever being obeyed!

GOSUB and the stack

The ZX BASIC GOSUB command works in exactly the same way as the GOTO command, but stores information on the GOSUB stack. This is used by the RETURN command to transfer control back to the statement following the GOSUB. To understand the GOSUB and RETURN command it is necessary to know a little about the way a *stack* works.

A stack, or to give it its proper name a 'Last In First Out (LIFO) stack', is a collection of storage locations plus a *pointer* used to mark the first free location. For example, a simple array can be used as a stack if it is associated with a variable, or *stack pointer*, that holds the index of the first free element. Data is entered to a LIFO stack by a *push* operation. This stores the data in the free location indicated by the stack pointer, and automatically moves the pointer on to the next free location. Similarly, data is retrieved from a LIFO stack by a *pull* operation. This moves the stack pointer back to the first location used and then returns the data stored in it. If the array S is being used as a stack, with P as its stack pointer set to point initially at the first element of S, then a push operation would be

```
LET S(P)=D:LET S=S+1
```

and a pull operation would be

```
LET S=S-1:LET D=S(P)
```

where the variable D is used to hold the data in both cases. Notice that neither routine checks to make sure that the bounds of the array are not exceeded. A stack can either grow upwards, as in the example, or downwards, as is the case with most Z80 stacks, using high memory locations for data storage first.

The important feature of a LIFO stack is indicated by its name. The last data item pushed onto the stack is the first item to be pulled from it. For example, if the items A, B and C are pushed onto a stack, then the first pull will return C, the second B and the third A. This is exactly the behaviour needed to implement the storage of return line numbers following a GOSUB. Each GOSUB effectively pushes a return line number onto the GOSUB stack and each RETURN pulls a return line number from the GOSUB stack. Thus if you execute GOSUB A, GOSUB B and GOSUB C in that order then the first RETURN will transfer control to the line following the GOSUB C, the second RETURN will transfer control to the line following the GOSUB B and the third RETURN will transfer control to the line following the GOSUB A. In this way a stack serves both to remember the return addresses and to supply them in the correct order as they are needed.

The GOSUB stack used in the Spectrum is a little odd; it is mixed up with another stack used by the Z80 to store the return address for the machine code equivalent of a GOSUB. To be more precise, the GOSUB stack is part of the Z80 machine stack. However, it turns out that most of the time the system variable ERR SP contains the address of the first item on the machine stack proper, and the contents of ERR SP plus two are therefore the address of the first item on the GOSUB stack.

It is interesting to note that both the line number and the statement number within the line are stored on the GOSUB stack. This means that a GOSUB within a multi-statement line will RETURN to any statement following in the same line. For example,

```
10 GOSUB 1000:PRINT "line 10 statement 2"
20 PRINT "line 20 statement 1":
```

will cause both PRINT statements to be executed as subroutine 1000 returns control to line number 10, statement 2. The exact data stored

on the GOSUB stack are first a two-byte number representing the current statement number plus one, followed by a two-byte number representing the current line number.

This information can be used to write a program that will cause a RETURN to transfer control to any line and statement number. This sort of disorderly jumping around a program is not to be encouraged, but it is sometimes useful for implementing special error RETURNs from subroutines.

```
 10 GOSUB 200
 20 PRINT "line 20"
 30 PRINT "line 30"
 40 STOP

100 LET G=2+PEEK 23613+256*PEEK 23614
110 POKE G,L-INT(L/256)*256
120 POKE G+1,INT(L/256)
130 POKE G+2,S
140 RETURN

200 PRINT "line 200"
210 LET L=30:LET S=1
220 GOTO 100
```

Subroutine 100 first PEEKs ERR SP to find the location of the first item on the GOSUB stack. Then lines 110 and 120 POKE a new value for the line number, and line 130 POKEs a new value for the statement number. In this way LET L=x:LET S=y:GOTO 100 will result in the next RETURN transferring control to line x statement y. You should be able to see this in action: the subroutine 200 given above RETURNs control to line 30 statement 1 rather than line 20 statement 1.

The FOR loop

ZX BASIC's implementation of the FOR loop is very clever and versatile, but different from that used by most versions of BASIC. To allow FOR loops to be nested one within the other the usual method is to use a stack, a *FOR stack*, to store the line numbers to which NEXT commands will transfer control (the so called 'looping lines'). The reason for using a stack to store the looping lines is similar to the reason for using a GOSUB stack to hold RETURN line numbers. Each FOR loop pushes the line number of its looping

line onto the FOR stack, and this means that a NEXT statement will always transfer control to the looping line of the last or innermost FOR loop. However, ZX BASIC does *not* use a FOR loop stack, and this makes it behave in a different way to most other versions of BASIC.

Each time that a FOR statement is encountered, the variables area is searched for any variables with the same name as the index variable used. If one is found, then it is removed. Then a new variable with the same name is created as an index or type 7 variable. The format of an index variable was given earlier in the chapter, but it is repeated in Fig. 4.3. Notice that as well as the usual five-byte

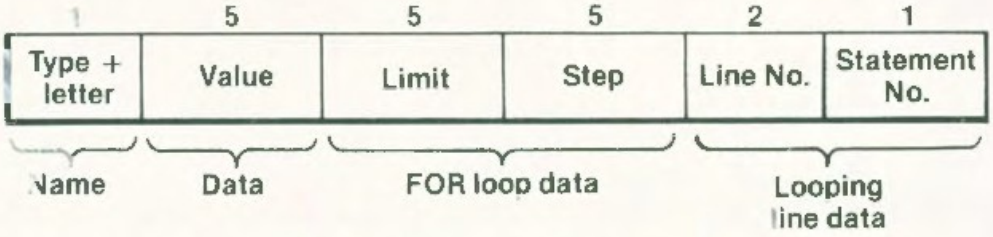


Fig. 4.3. Data format for an index variable in ZX BASIC.

value associated with every numeric variable, it contains all the information needed to implement the FOR loop. The 'limit' and 'step' are the final value and step size of the FOR loop respectively. The looping line is stored as a two-byte line number and a one-byte statement number, and this defines the statement to which a NEXT command quoting the index variable will transfer control.

The only real effect that a FOR statement has is to create a new index variable. All of the real work in a FOR loop is carried out by the NEXT statement. When a NEXT statement is encountered the 'step' is added to the 'value' and the result is compared with the 'limit'. If the result exceeds the 'limit' then the loop ends. Otherwise the control is passed to the looping line. Apart from its use by a NEXT statement, the index variable can be manipulated and used just like any other numeric variable. Thus to bring a FOR loop to a premature end you can simply set the index variable to be bigger than the limit.

There are two important consequences of ZX BASIC not using a FOR stack. Firstly, unlike most versions of BASIC, you can jump out of a FOR loop before it is completed without any worries. If you do this in a BASIC that uses a FOR stack the entry on the stack never gets removed (pulled), so the stack slowly fills up, finally giving an error message. The only penalty in ZX BASIC is that an

index variable is left hanging around, but this can be used as an ordinary variable, and a new FOR loop on the same index will also reuse it. Even though jumping out of FOR loops does no harm in ZX BASIC it is not a good habit to acquire. If you do, then your programs will be more difficult to transfer to other versions of BASIC.

The second effect of not using a FOR stack is remarkable to watch! A spinoff of using a FOR stack is that improper nestings of FOR loops are automatically detected, and an error message issued. In ZX BASIC, however, almost any nesting of FOR loops will work. For example, try

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 PRINT J,I
40 NEXT I
50 NEXT J
```

Most versions of BASIC, and most programmers familiar with other versions of BASIC, would reject the above program as being incorrect (lines 40 and 50 should be swapped to produce the correct nesting of two FOR loops). If you run the above program in ZX BASIC you will find it not only works but might even be useful! Trying to understand such an odd nesting of FOR loops should convince you to avoid them! The nesting works because each of the NEXT statements at lines 40 and 50 is obeyed without reference to the rest of the program. So line 40 transfers control to line 20 ten times for values of I from 1 to 10. Each time through the loop line 20 creates the index variable J and sets its value to 1. After this control passes to line 50 which causes the FOR loop on J (i.e. lines 20 to 50) to be carried out ten times. Each time through this loop the NEXT I at line 40 doesn't repeat the FOR loop on I because the 'value' stored in i is already bigger than the 'limit'. It does, however, increase the 'value' stored in the index variable by adding the 'step'. You should now be able to understand the sequence of numbers that this pair of loops prints on the screen!

Conclusion

The information presented in this chapter should help you to understand the inner workings of ZX BASIC. Many of the program examples given not only illustrate the ideas involved but also form

the basis of a useful collection of programming utilities. If you would like to test your understanding of ZX BASIC then there is no better way than by working on some of the many projects that spring from these examples. Much of the work can be done in ZX BASIC, but if you are learning Z80 assembler then you will find many rewarding problems that are not too difficult to solve.

Chapter Five

I/O - Channels and Streams

The Spectrum has a very sophisticated and general method of dealing with different I/O devices, based on *streams* and *channels*. The standard Spectrum has a very limited range of I/O devices, and this means it is possible to use special commands for each device. For example, to send data to the screen you use the PRINT statement, but to send data to the ZX Printer you use the LPRINT command. Once the Microdrives are added to the system, inventing special commands quickly becomes inadequate. Even without the Microdrives there are advantages to using the Spectrum's general method of defining the device to be used in an I/O operation. Surprisingly, the Spectrum manual completely fails to mention or even hint at the method of handling I/O via streams and channels.

Streams - INPUT # and PRINT

A good way of thinking about I/O is to separate it into two parts, one corresponding to the software that receives or generates the data, and the other corresponding to the hardware that receives or generates the data. In ZX BASIC the software component of I/O is referred to as a 'stream' and the hardware component as a 'channel'. The key difference is that a stream is a featureless flow of data into or out of a program, but a channel corresponds to a particular I/O device such as the ZX Printer. Think of a stream as a collection of data items on their way to or from some piece of hardware. Streams are identified by a number in the range 0 to 15, and their basic operations are reading and writing data. The instruction

```
INPUT #s;'input list'
```

will read data from stream 's' into the variables in the 'input list'. For example

INPUT #0;A;B;A$

will read data from stream 0 and store it in the variables A, B and A$.

In the same way the command

PRINT #s,'print list'

will send data to the stream 's' from the variables in the 'print list'. For example

PRINT #0;TOTAL;A$

will send data to stream 0 from the variables TOTAL and A$.

Notice that both the INPUT # and PRINT # can be used in exactly the same way as the ordinary INPUT and PRINT statements. Any item that you can use as part of a normal 'input list' or 'print list' can be included as part of the stream I/O statements. For example,

PRINT #2;"HI THERE";TAB(10);"FOLKS"

and

INPUT #0;"What is your name";N$

are both valid. The PRINT statement sends the literal string "HI THERE", then a TAB code, followed by the literal string "FOLKS" to stream 2. Notice that the data that is sent to the stream is exactly the same as the data that would be sent to the screen. The INPUT statement is a little more complicated in that it not only requests data from stream 0, but also sends data in the form of the literal string "What is your name". Each stream number is in fact associated with two streams of data – an input stream and an output stream. Data written to the stream by either PRINT or INPUT is sent to the output part of the stream, and any data read from the stream is obtained from the input part of the stream.

In practice it is possible to use a PRINT or INPUT statement that refers to more than one stream. For example

PRINT #5;"Hi there";#6;"folks"

will send the literal string "Hi there" to stream 5 and "folks" to stream 6. In other words a 'stream specifier' #s can be included in a print or input list wherever it is necessary to change streams. Although switching streams in mid-statement is possible, it is best avoided unless there are special reasons for doing it. Programs that use a number of streams in each I/O statement are very difficult to understand, debug and alter.

Channels - OPEN and CLOSE

The idea of a stream of data is easy enough to understand, but you might be wondering how the stream numbers are associated with hardware I/O devices? The answer is that before any data is sent or received over a stream it has to be OPENed. OPENing a stream serves two purposes. It associates a stream number with a particular I/O device, and signals the I/O device that it is going to be used. Often as well as just signalling that a device is about to be used, OPENing a stream involves initialising the device to get it into a state where it can be used. However, such initialisation depends very much on the device itself. To OPEN a stream, ZX BASIC provides the command

```
OPEN #s,c
```

where 's' is the stream number being opened, and 'c' is a string specifying the channel it is being associated with. Following this command, the destination of any data sent to the stream 's' will be the channel 'c', which will also be the source of any data read from the stream. Before a practical example of using the OPEN command can be given we need to know what channels the Spectrum has.

The unexpanded Spectrum (i.e. without Microdrives) recognises only three different channels:

K – the keyboard channel
S – the screen channel and
P – the printer channel

Thus,

```
OPEN #5,"K"
```

OPENs stream 5 and associates it with the keyboard. Following this command

```
INPUT #5;A;B
```

will get data from the keyboard in the same way that a normal INPUT command would. However, the command

```
PRINT #5;"HI THERE"
```

now sends data to the output side of stream 5, which is associated with the keyboard's display area at the bottom of the screen. Thus the literal string "HI THERE" is printed in the lower part of the screen normally reserved for INPUT messages. If you try this you

are unlikely to be able to see the string, as the lower area of the screen is cleared when a program halts or when an INPUT statement is encountered. If you would like to see the effect of sending data to the 'input area' of the screen try:

```
10 OPEN #5,"K"
20 PRINT #5;RND
30 GOTO 20
```

You should see random numbers printed on the screen starting from the bottom and scrolling up. The program will end with an OUT OF SCREEN error message, as the input area of the screen will not scroll in the same way as the normal print area.

Although in principle each stream has both an input and an output side, in practice the only channel that can accept both input and output is the keyboard channel. The other two – screen and printer – are output only channels, and any attempt to read data from them produces an error report J. Notice that this restriction is entirely a feature of the hardware that the stream is attached to.

You can associate more than one stream with any given channel, but if you want to change the channel that a stream is associated with then its current association must be removed by CLOSEing it. The ZX BASIC command

```
CLOSE #s
```

will remove any existing association between the stream 's' and a channel. In this sense CLOSEing is the reverse of OPENing a channel. CLOSEing a stream can also be used to inform the hardware component of a channel that it is no longer required by the stream, and any 'cleaning up' operation that it needs should be carried out ready for another channel to use it.

It is important to notice that while a channel can be used by a number of streams, a stream can only be associated with a single channel. For example, the ZX Printer might be associated with channels 4 and 6, so

```
PRINT #4;'print list'
```

and

```
PRINT #6;'print list'
```

would both send data to the printer, but it is impossible to associate, say, stream 7 with both the printer and the screen.

The use of streams - device independence

So far the only advantage to be gained from using streams is the ability to send data to the lower half of the screen. For the ZX BASIC programmer using an unexpanded Spectrum there is in fact only one other reason for using streams, but this is an important reason. *Device independence* is an idea that is usually reserved for advanced computer science courses, but it is a simple and very useful idea. Device independence is just the ability to write a program without having to worry about where the data that it needs is coming from or where the data it generates is going to. For example, you might write a program that produces listings of financial data without worrying about whether the output was going to a screen or to a printer. The device that the output was actually going to would be selected at a later data by the user of the program. If you use PRINT and LPRINT to send data to the screen and the printer respectively then it is not easy to write device independent programs, but using streams and channels it is!

Consider the problem of writing a program to print a list of random numbers either on the screen or on the printer, depending on which the user wanted. Using PRINT and LPRINT the program would be something like:

```
10 INPUT "Printer or Screen ";A$
20 IF A$(1)="P" THEN LPRINT RND
30 IF A$(1)="S" THEN PRINT RND
40 GOTO 20
```

Using streams and channels the program would look something like:

```
10 INPUT "Printer or Screen ";A$
20 OPEN #5,A$(1)
30 PRINT #5;RND
40 GOTO 30
```

Because the channel specifier can be a string variable, stream 5 is associated with either the printer or the screen. Another way of achieving the same result would have been to OPEN two different streams, one to the printer and one to the screen. You could then use the fact that the stream specifier can be an arithmetic expression to select which one was to be used, as in the following program:

```
10 INPUT "Printer or Screen ";A$
20 OPEN #5,"P"
30 OPEN #6,"S"
```

```
40 IF A$(1)="P" THEN S=5
50 IF A$(1)="S" THEN S=6
60 PRINT #S;RND
70 GOTO 60
```

Although this example is too small to be really convincing in itself, you should be able to see that in a large program it is an advantage to use streams to group together all similar PRINT or INPUT statements. If this is done, changing where they send their output is simply a matter of changing the appropriate OPEN command, or the stream number that they use. When you add the Microdrives to the system, streams are unavoidable, so it makes sense to get the maximum benefit from them even at this early stage.

The default streams

The four streams 0 to 3 are automatically OPENed by the Spectrum as part of its set-up procedure. Initially the stream-to-channel assignments are as follows:

stream	channel
0	K
1	K
2	S
3	P

So even without an OPEN command

```
PRINT #2;"HI THERE"
```

will print onto the screen. These streams are used by the Spectrum to direct program data to the correct device. For example, an LPRINT sends data to stream 3. These assignments of streams to channels can be changed using OPEN commands but the streams themselves cannot be CLOSEd. An attempt to CLOSE one of the default streams results in it being re-OPENed to its initial channel, as given in the table above.

Other stream commands

The only other two stream I/O commands that can be used with the

unexpanded Spectrum are LIST and INKEY$. The full form of the list command is

LIST #s,n

where 's' is the stream number that the program is to be listed to, and 'n' is the line number that the listing will start from. For example

LIST #1

will list a program on the bottom part of the screen normally reserved for input, but

LIST #3

is the same as LLIST.

The other stream-oriented command, INKEY$, can be used to return a single character (byte) from any stream that is associated with a device that supports input. The function

INKEY$ #s

will return a single character from the stream 's'. If no character is available from the input device, then the null string is returned. The only problem with this extended form of INKEY$ is that the unexpanded Spectrum has only one input channel – the keyboard. However, once the Microdrives are added the number of input channels increases, and so does the number of useful stream-oriented commands.

Channels and streams - memory formats

Although the ZX BASIC programmer need not worry about how channels and streams are implemented to make use of them, there are ways in which the machine code programmer can make use of such knowledge. In particular, the channel is the ideal way of extending the range of I/O devices that the Spectrum can handle without having to write code for special I/O commands.

The information that defines each channel is stored in the channel information area starting at CHANS and ending at PROG−2 (where CHANS and PROG are both system variables). Each channel has a separate 'channel record' which has the following format

address	size	
n	2 bytes	address of output routine
n+2	2 bytes	address of input routine
n+4	1 byte	channel code letter

where the input and output routines are machine code subroutines. The output routine must accept Spectrum character codes passed to it in the A register. The input routine must return data in the form of Spectrum character codes, and signal that data is available by setting the carry flag. If no data is currently available then this is signalled by resetting both the carry and the zero flag. If the channel cannot support input, or cannot support output, then the address for the routine that performs the illegal operation should be set to an error handling routine. The standard way of handling errors in ZX BASIC is via a Restart call to address 8. In Z80 assembly language this amounts to

```
ERROR   RST 0008
        DEFB errocode
```

where 'errocode' is the numeric code of the report to be given to the user.

The channel records for the three standard Spectrum channels, and an additional channel that has not yet been described, are as follows:

address	keyboard channel record
CHANS	address of lower screen printout routine
+2	address of keyboard input routine
+4	K channel K identifier
	screen channel record
+5	address of screen printout routine
+7	address of error routine
+9	S channel S identifier
	editing buffer channel record
+10	address of buffer input routine
+12	address of error routine
+14	R channel R identifier

	ZX Printer channel record
+15	address of ZX Printer routine
+17	address of error routine
+19	P channel P identifier

Notice that each channel record is in the standard format, as described, and that the only channel that can support both input and output is the K channel. The new R channel is used internally by the Spectrum to send data to the editing buffer. The OPEN command will not allow the user to associate a stream with the R channel, so its use is limited.

Important Note – the format of a channel record is different when the Microdrives and interface 1 are in use. If you are going to make sense of this information to create your own channel records and want your program to work with both the unexpanded and expanded Spectrum see Chapter 10.

Data about the association of streams with channels is stored in the system variables area of memory, in the 38-byte area starting at STRMS (address 23568). The stream table, and each pair of bytes in this table, holds a number 'x' that represents the address of the start of a channel record. Rather than simply storing the address of the channel record, 'x' is the 'distance' that the channel record is away from the start of the channel information area:

address of start of channel record = address of channel area + x − 1

So each entry in the stream table is one more than the number of memory locations that the channel record is offset from the start of the channel information area. As there are a maximum of 16 streams you would think that a maximum of 32 bytes (i.e. one channel address per stream) would be sufficient to store all of the channel and stream associations. In fact the extra six bytes are used to store channel information for three internal streams corresponding to stream numbers 255, 254 and 253. These three internal streams are automatically associated with channels R, S and K respectively, and as stream numbers are restricted to the range 0 to 15 they are inaccessible from ZX BASIC. However, the presence of these three internal streams does have to be taken into account when trying to find the address of the channel record corresponding to any of the streams 0 to 15. The first three entries in the stream table are for the

internal streams 253 to 255; the fourth entry gives the address of the channel record to be used with stream 0, and so on. This means that the start of the stream table, as far as external streams are concerned, is

STRMS+6 or 23574

and the address of the start of the channel record associated with stream s (s in the range 0 to 15) is stored in the two memory locations starting at:

23574+s*2

When a stream is OPENed to a particular channel, the OPEN command stores the difference between the start of the channel record and the channel area itself, plus one, in the correct position in the stream table. When an INPUT or PRINT command sends data to a particular stream, the stream table is examined to find the address of the channel record. When a stream is CLOSEd the number stored in its entry in the stream table is set to zero. Thus a zero entry in the stream table is used to detect an attempt to use a stream that hasn't been opened yet. Seven streams are automatically OPENed, the three internal streams and streams 0 to 3 as already described.

This system of channel records and the stream table is extended for use by the Microdrives, but its essential features remain the same. Each channel is described by a channel record, and streams are associated with channel records by use of the stream table.

Before we consider using the stream and channel I/O, it is worth mentioning the only other system variable that is connected with channel I/O - CURCHL. Each time a stream-oriented I/O command is used, the stream number is used to look up the address of the channel record in the stream table. This address, once found, is then stored in the system variable CURCHL to direct all of the data produced by the I/O command to the correct channel. Thus following a command such as PRINT #s, CURCHL contains the address of the start of the channel record associated with stream 's'.

Creating your own channels

If you have special I/O device connected to your Spectrum, or if you are planning to build a new device, then the problem of how to send data to it or receive data from it will have occurred to you. Indeed, it is usually thought easier to construct a hardware interface to the

Spectrum than a 'software interface' with ZX BASIC. Using the information about stream-oriented I/O in the last section it is comparatively easy to interface special I/O devices to ZX BASIC in a way that allows them to be treated on a par with Sinclair's own hardware.

The usual way of providing software to handle special I/O devices is to write BASIC subroutines using IN and OUT. These send and receive data directly to and from the I/O ports allocated to the device. For example, if a sound generator was allocated port 31 for its frequency control register then

```
OUT 31,f
```

would send the data (in the range 0 to 255) to the sound generator, and so set its frequency. For simple devices, and devices controlled by individual bits in the data, IN and OUT are very suitable. However, if the device is 'character-oriented' – if it receives and sends data in the form of characters – then IN and OUT are inadequate. For example, a parallel printer and a modem are both character-oriented devices, and the best way to deal with them is via the usual PRINT and INPUT statements. Even if suitable subroutines could be written using OUT and IN to send and receive numeric and string data, it is difficult to see how they could be used to LIST programs to the new devices. Clearly the way to go about providing a software interface to new character-oriented devices is via streams and channels.

Adding a character-oriented device to ZX BASIC's system of streams and channels can be done in one of two ways; either by changing the addresses stored in an existing channel record, or by creating a completely new channel record.

The first method involves POKEing new addresses into an existing channel record that *point* to your own machine code routines, positioned somewhere in memory. For example, suppose you want to interface a full-sized printer in place of the ZX Printer. Changing the address stored in the first two locations of the ZX Printer's channel record (to point to your own printer driver output routine) will make the commands LPRINT and LLIST, as well as any I/O commands referring to streams OPENed to channel P, send their data to the new printer. Writing the new printer driver is easy in principle. All it has to do is accept ASCII character codes in the A register, and use these to print the correct ASCII characters on the printer. However, the Spectrum's character set includes many characters that the standard ASCII character set lacks, and these would have to be detected and correctly interpreted by the driver. For example, all of the position

and attribute control items within a PRINT statement will be sent to the printer driver as control codes, as listed in Appendix A of the Spectrum manual. For instance, LPRINT TAB(10); sends ASCII codes 23, 10 and 0 to the printer driver. The 23 is the Spectrum's control code for TAB, and the following two codes are the least and most significant byte of the parameter of the TAB function. (The codes that are sent to the Spectrum's output drivers are discussed more fully in the next chapter.) It is important to realise that all of the Spectrum's output is converted to a stream of ASCII characters and codes before it is printed on the screen. This makes it possible for a printer driver to respond to or ignore all of the Spectrum's position and attribute control items as desired. For example, if the new printer was a colour printer then it could change the printing colour in response to

```
LPRINT INK 3;"Hi there"
```

which sends the ASCII codes 16 (for INK) followed by 03 (for colour 03) to the printer driver.

As an example of this method of interfacing a new I/O device, the following program changes the output address stored in the P channel to refer to a machine code routine stored in the printer buffer area of memory. (Notice that this only works because the ZX Printer is not in use!) The new machine code output routine doesn't do anything really useful with the data it receives; it just sends it to I/O port 254, which controls the speaker and border colour. At least this ensures that its effects can be seen and heard! The Z80 assembly language for this simple driver is

address		assembler	code	comment
23296	outdrv	LD BC,254	01,254,00	load BC reg with 254
23299		OUT (C),A	237,121	send A register to port 254
23301		RET	201	return to ZX interpreter

This is used in the following ZX BASIC program:

```
10 DATA 01,254,00,237,121,201
20 FOR A=23296 TO 23301
30 READ D
40 POKE A,D
50 NEXT A
```

```
100 GOSUB 1000
110 FOR I=0 TO 7
120 LPRINT I;
130 NEXT I
140 GOTO 110

1000 LET C=PEEK 23631 + 256*PEEK 23632
1010 LET C=C+15
1020 POKE C,23296-INT(23296/256)*256
1030 POKE C+1,INT(23296/256)
1040 RETURN
```

The machine code output routine is stored in the DATA statement in line 10, and loaded into memory by lines 20 to 50 (23296 is the start of the ZX Printer buffer). Subroutine 1000 changes the address in the channel record for channel P. Line 1000 gets the address of the start of the channel area into c and then line 1010 finds the start of the channel record for channel P. Lines 1020 and 1030 POKE the address of the new output routine into the first pair of locations in the channel record. If you enter and run this program you will see the border flash and change in a very wild manner. If you break into the program, you will obtain further proof that the new output routine is sending data to the border control port by LLISTing the program to it. (Note: disconnect the ZX printer before running this program.)

Changing the addresses stored in existing channel records is an easy method of adding new devices, but it does have the disadvantage of removing one of the Spectrum's existing I/O devices. In practice it is impossible to modify channel K (the keyboard's channel record) because its I/O addresses are restored each time an INPUT statement is executed. This leaves the channel records for channel S and channel P as the only candidates for modification, and as channel S is far too useful the only real candidate is P. This is fine as long as you don't want to use more than one extra I/O device, and you don't want to use the ZX Printer at the same time.

To add any number of extra I/O devices it is necessary to add new channel records. If you want to do this in a completely general way, then you must take into account how the Microdrive modifies the stream/channel system of operation. This is dealt with in Chapter 12. Adding a new channel record sounds very easy, but there are a few minor details to consider. Firstly, it is possible to create a channel record anywhere in memory, not just in the channel information area, but if the channel record is stored above the INPUT workspace area (starting at WORKSP) the CURCHL (current channel) system

variable will be altered as memory is added to the workspace area during an INPUT command. This, of course, will mean that the current location of the channel record will be lost, and the Spectrum will crash. However, if the channel record is stored below the INPUT workspace area everything works correctly. In the demonstration given below, the ZX printer buffer is used to store both the new channel record and the new I/O routines. In a real application the channel record would be added to the channel information area, and details of how to do this are also given in Chapter 12. A second difficulty is that the OPEN and CLOSE commands will only work with the standard channel records for K,S and P. This means that as well as providing new channel record and I/O routines, you also have to provide a subroutine to open the channel to any stream, and if necessary a subroutine to close it. Putting all this together gives the following Z80 assembler for the channel record and I/O routines:

address	assembler		code	comment
23296	chanrec	DEFB 0	0	l.s.b. of output address
23297		DEFB 91	91	m.s.b. of output address
23298		DEFB 11	11	l.s.b.of input address
23299		DEFB 91	91	m.s.b of output address
23300		DEFB "E"	69	channel identifier
23301	outdrv	LD BC,254	01,254,00	load BC reg with 254
23304		OUT(C),A	237,121	send contents of A to 254
23306		RET	201	return to ROM code
23307	indrv	RST 8	207	error restart
23308		DEFB	18	invalid device error code

The first five bytes form the new channel record. The routine starting at 23301 is the output routine and this simply sends the code in the A register to output port 254, which is the speaker and border colour port. The routine starting at 23307 is the input routine and this simply reports an error to indicate that input is not allowed with this channel. Obviously in a real application either routine could be very much more complicated. The following BASIC program uses this machine code:

```
10 DATA 0,91,11,91,69,1,254,0,237,
   121,201,207,18
20 FOR A=23296 TO 23308
```

```
 30 READ D
 40 POKE A,D
 50 NEXT A

100 LET S=5:GOSUB 1000
110 PRINT #5;RND;
120 GOTO 110

1000 LET A=23574+2*S
1010 LET C=PEEK 23631+256*PEEK 23632
1020 LET R=23296-C+1
1030 POKE A,R-INT(R/256)*256
1040 POKE A+1,INT(R/256)
1050 RETURN
```

Lines 10 to 50 load the new channel record and I/O routines into the ZX printer buffer. Subroutine 1000 will open stream s to the new channel. In other words it is the equivalent of OPEN #s,"E". Line 1000 works out the correct address for stream s in the stream table. Lines 1010 and 1020 work out new channel records offset from the start of the channel information area (plus one), and lines 1030 and 1040 store it in the stream table. Line 100 uses subroutine 1000 to open stream 5 to device E, and lines 110 to 120 provide a demonstration by sending the codes corresponding to random numbers to the sound and border control port. A further demonstration can be gained by stopping the program and typing LIST #5. This produces a flash of colour and sound indicating that the program has been listed to port 254! If you change line 110 to read

```
110 INPUT #5;i
```

then you will get the correct error message, indicating that this channel cannot be used for input.

Apart from having to write more comprehensive and specialised I/O drivers, there is nothing difficult about adding new channel records to ZX BASIC. Notice, however, that the above program will not work if the Microdrives are connected – but the modification necessary to make it work are trivial (and described in chapter 10).

The problems of writing an output driver have already been described, but before bringing this chapter to a close it is worth mentioning the extra requirements for an input driver. If an input channel is going to supply a single character code, as an eight-bit A to D convertor might, then the best BASIC command to use is

INKEY$ #, which will return a single character. However, if you are planning to use INPUT# to read in a collection of character codes then you have to be aware of two things. Firstly, INPUT statements also perform output by printing prompts etc. It is not enough to make the output routine an error return: you have to handle any data that the INPUT statement sends, even if you simply ignore it! Secondly, an INPUT # statement accepts data as if it was being typed at the keyboard. This means that if you use INPUT #s;i to read in a number to the variable i, then the device driver has to send a collection of ASCII codes corresponding to digits and ending with an ENTER code, just as a number would be entered from the keyboard. Finally, notice that even when reading data from a special piece of hardware, the INPUT command will interpret editing codes, delete etc. correctly! The best way to think about it is that INPUT always works as if it were receiving a stream of character codes corresponding to keys that are being pressed on the keyboard.

Conclusion

The Spectrum's system of streams and channels is something of a surprise bonus to the ZX BASIC programmer. Used within programs it provides the advantage of device independence, and an overall increase in flexibility, with no disadvantages. To the Z80 assembly language programmer, streams and channels are the ideal way of providing software interfaces with any new equipment.

Chapter Six

The Video Display

The hardware that generates the Spectrum's video display has already been described in Chapter 2, but there the emphasis was on general principles and how the video hardware co-operated with the rest of the machine. In this chapter the methods that the Spectrum uses to generate a video display are discussed in detail, with the emphasis on the way that the software and harware interact.

The Spectrum's video display deserves close inspection because it combines a number of interesting features in a way that produces a flexible system with reasonable memory requirements. Its flexibility comes from its use of a single high-resolution mode to display both text and graphics. This, in theory at least, permits the free mixing of text and graphics anywhere on the screen, but in practice the Spectrum's software restricts the positioning of characters to a number of predetermined character locations. The saving in memory is achieved by the use of 'parallel attributes' to control colour. This does indeed save a great deal of memory while still allowing the use of eight colours. The price to be paid for this economy is the restriction on the number of colours that can be used in each character location. Surprisingly, parallel attributes work very well and fit naturally into the way that many graphics programs organise colour.

Although the Spectrum's display is praiseworthy there is still room for improvement. Fortunately, most of the shortcomings are to be found in the software, and this is something that can be extended to provide whatever facilities are required. However, to make this possible it is important to have a good understanding of how things work.

Black and white to colour

The easiest sort of display to work with is a black and white or two-

colour display. The reason for this is that a single binary bit (i.e. 0 or 1) can indicate which one of two states something is in. The simplest graphics scheme associates one colour with each state, for example black with 0 and white with 1. In this way a bit pattern can be used to represent the colours of a collection of dots on the screen. Notice that each bit in the bit pattern controls the colour of exactly one dot on the screen. This correspondence between bits and screen dots gives this method of generating graphics its usual name, *bit-maped graphics*.

The Spectrum uses a bit-mapped graphics method, so each of the dots that make up the 192 by 256 screen is controlled by a single bit stored somewhere in memory. Notice that as each memory location holds eight bits it can control the colour of eight dots on the screen. The exact correspondence between memory locations and groups of eight screen dots is described in the next section.

This method using a single bit to control the colour (black or white) of a single dot on the screen, must be modified to include the use of more than just two colours. This is more difficult than you might think. The most obvious method for associating more than one bit with each screen dot soon uses up a great deal of memory. For example, to provide a choice of four colours for a dot takes two bits, doubling the amount of memory required. A choice of eight colours requires three bits, sixteen colours requires four bits, and so on. To produce the Spectrum's eight-colour display in this way would take 18K of memory, which would make a 16K colour Spectrum an impossibility. Besides using a large amount of memory, this extended bit-mapped technique creates other problems. It is very difficult to retrieve data from memory fast enough to supply three bits per screen dot.

The solution adopted by the Spectrum is based on the observation that most colour displays use only two colours in any given region of the screen. For example, a blue sky with white clouds and a yellow sun uses three colours, but near the cloud we have only blue/white and near the sun only blue/yellow. In the Spectrum the screen dots are grouped in eight by eight squares corresponding to the familiar 24 lines of 32 character locations. Within each of the character locations each dot can only be one of two possible colours – in the ZX BASIC jargon the 'ink' and 'paper' colours. As in the simple two-colour example, the choice of ink or paper colour for a dot is controlled by a single bit in a memory location. The extra flexibility of this new arrangement comes from the fact that the ink and paper colours within each character location are controlled by a single

memory location, an *attribute byte*. The Spectrum's colour display is a halfway house between a simple two-colour display and a true multi-colour display. Each dot on the screen corresponds to a single bit in memory that determines whether it is an ink or a paper dot. The actual colour assigned to ink and paper dots within any given character location is determined by the values stored in the corresponding attribute byte. The advantages of this parallel attribute method of producing a colour display are easy to appreciate. A great deal of memory is saved by using a single attribute byte to control the ink and paper colours for the 64 dots in a character location. However, the limitation of the scheme is equally obvious – only two colours can be present in each character location.

The video memory

There are two areas of RAM involved in the production of the Spectrum's video display – the *display file*, between 16384 and 22527, and the *attribute file*, between 22528 and 23295. As you might expect, the display file is the region of memory used to hold the bits that determine whether each dot on the screen is an ink or paper dot. Similarly, the attribute file is the area of memory where the attribute bytes are stored. Knowing this is a step in the right direction, but to control the screen directly you need to know exactly how to find the bit that controls a particular dot, or the byte that controls a particular character location. What is required is an equation that will convert screen co-ordinates into the address of the memory location concerned. Obviously there are going to be two types of equation, one for the display file and one for the attribute file.

The display file map

The most obvious arrangement for the display file is to use the first memory location (i.e. 16384) to store the first eight dots in the top row, the second memory location to store the next eight dots in the row and so on. This is indeed the case, and in general each row of 256 dots is stored in 32 consecutive memory locations. However, there are complications. The rows are not stored in the obvious order i.e. first row first, second row second and so on to the bottom row. Instead they are stored in an order that reflects the 24 lines of

character locations. After the top row of dots comes the top row of the second line of character locations, then the top row of the third line of character locations, and so on to the top row of the eighth line of character locations. In other words, the top rows of each of the first eight lines of character locations are stored first. After this the second row of dots of each of the eight lines of character locations are stored, then the third row, and so on. This pattern of storage is then repeated with the next eight lines of character locations, and finally with the last eight lines of character locations. Notice that this effectively divides the screen up into three portions of eight lines each. Each portion is then stored in the order that the rows of dots make up the lines of character locations, i.e. all the first rows, then all the second rows and so on (see Fig. 6.1).

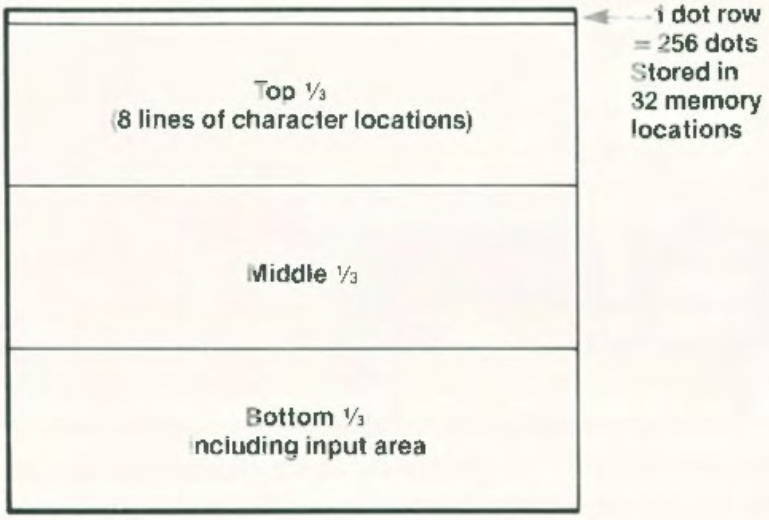


Fig. 6.1. How the video display is divided up for storage in memory.

This storage scheme is easy to understand once the basic sequence has been grasped. Perhaps the easiest way to do this is to watch the following program in action:

```
10 FOR I=16384 TO 22527
20 POKE I,255
30 NEXT I
```

This program stores 255 in each of the memory locations in the display file in turn. As 255 is 11111111 this causes the eight dots controlled by the memory location to be displayed as ink dots. In this way the position of the dots corresponding to each memory location can be seen in sequence, and the order that the dots are changed to ink is as described above.

We now know the correspondence between memory locations

and dots, but to be of any use this information has to be presented as an equation that will convert screen co-ordinates into the address of the controlling memory location. There are two ways of specifying the location of a dot on the screen: by character location and by graphics co-ordinates. For example, you might want to find the address of the memory location that controls a particular row of eight dots in a particular character location. If the character location is in line L and column C, then the memory location that controls row R is given by:

```
16384+2048*INT(L/8)+32*(L-8*INT(L/8))+256*R+C
```

To prove the accuracy of this formula, try the following program:

```
10 DEF FNm(L,C,R)=16384+2048*INT(L/8)
   +32*(L-8*INT(L/8))+256*R+C
20 CLS
30 FOR N=0 TO 7
40 FOR I=0 TO 31
50 FOR J=0 TO 23
60 POKE FNm(J,I,N),255
70 NEXT J
80 NEXT I
90 NEXT N
```

This fills the screen from top to bottom and left to right.

The alternative way of specifying a single dot uses the familiar x and y co-ordinates. This results in an even more complicated equation giving the address of the memory location that controls it:

```
16384+32*(INT((175-Y)/8)-INT((175-Y)/64)*8
 +8*(175-Y-INT((175-Y)/8)*8)+64*INT((175-Y)/64))
+INT(X/8)
```

The bit in the memory location is given by:

```
8-X+INT(X/8)*8
```

This equation may seem remarkably cumbersome, and indeed it is when written in BASIC. However, operations that involve dividing by and multiplying by powers of two are easy to implement in Z80 assembler. It is easier to understand the equation if it is written using the standard operations

```
x DIV y          meaning INT(x/y)
```

and

x MOD y meaning the remainder after dividing x by y
i.e. x−INT(x/y)*y

Using these operations and Z=175−Y the equation becomes:

```
16384+32*((Z DIV 8) MOD 8+8*Z MOD 8+64*Z DIV 64)
+X DIV 8
```

Even after all this work it has to be admitted that, apart from their use in Z80 assembler, such equations have very little value simply because they are so complex. However, knowledge of the overall structure of the video storage can be very useful indeed, as will be illustrated by the programs in the next chapter.

The attribute file map

The equation giving the location of the attribute byte that controls any given character location is very simple; a relief after the complexity of the display file map. Starting at 22528 the attribute bytes are stored in the natural 'printing' order of the character locations that they control. In other words the first attribute byte controls the character location in the top left hand corner, the next controls the next location to the right on the same line, and so on to the end of the line. This pattern then repeats for each line to the end of the screen. To see the form of this storage scheme try the following

```
10 FOR I=22527 TO 23295
20 POKE I,0
30 NEXT I
```

which stores the attribute code for black paper in each attribute byte in sequence. You should be able to see that unlike the previous programs, which manipulated the display file, each memory location POKEd alters a whole character location. The equation for the address of the attribute byte that controls the character location at line L column C is:

```
22528+32*L+C
```

This is a much more useful equation than either of the two given for the display file. In particular it can be used to change the attributes controlling a character location without changing the pattern of ink and paper dots on the screen. Try, for example,

```
10 DEF FNa(C,L)=22528+32*L+C
20 PRINT AT 10,5;"this is a message"
30 LET L=10
40 LET C=INT(RND*32)
50 LET A=INT(RND*256)
60 POKE FNa(C,L),A
70 GOTO 40
```

This program first prints a message on the screen, then uses the function FNa to POKE (line 60) random attribute codes into the attribute bytes that control the line the message is printed on.

PEEKing the display file - POINT and SCREEN$

Either of the two equations given earlier could be used to examine the current state of a bit in the display file by PEEKing the correct memory location. However, the address calculation involved is so complicated it's always faster to use the ZX BASIC function POINT. To implement POINT (x,y), ZX BASIC calculates the address of the memory location that controls the dot at x,y, then returns the value of the bit within it that controls the dot. So POINT(x,y) returns 0 if the dot is paper, and 1 if the dot is ink. It is unusual for the behaviour of a program to depend on the state of a single dot on the screen, and this limits the usefulness of the POINT function. If you need to test the state of a number of dots then the multiple use of the POINT function tends to slow things down rather a lot.

It is usually more important to discover what character is stored at any given location. Fortunately ZX BASIC has a function which solves just this problem. The function SCREEN$(line,column) returns the character displayed at the screen location line,column. This is achieved by examining each of the 64 dots that make up the character location in question and comparing them to the shape definitions stored in the Spectrum's character table. The character table is described in a later section, but essentially it is an area within the ZX BASIC ROM that contains the pattern of ink and paper dots that forms the shape of each character. The SCREEN$ function is easy to use, but it is important to be aware of one or two peculiarities. For example, as it works by comparing dot patterns within a character location with character definitions, it is only concerned with the shape that the dots form, not how the shape was produced. So, for example, if the shape of a letter A is produced by plotting individual dots using the PLOT command, or by PRINT

"A", the SCREEN$ function will still return "A". Another feature of SCREEN$ is that it checks for a character and its inverse. This means that the letter "A" will be returned if the shape is made by ink dots or by paper dots. Thus a solid block of ink dots will cause SCREEN$ to return a space character! Finally SCREEN$ will not recognise user-defined characters that are present on the screen.

Attribute codes and ATTR

In a previous section the attribute codes were POKEd to the attribute file, but unless you know exactly how the value stored in the attribute byte effects the character location it refers to, this is of very little practical use. The eight bits that make up an attribute code are used in the following way

b7	b6	b5 b4 b3	b2 b1 b0
f	b	paper	ink

where f stands for flash, b for bright and 'paper' and 'ink' are the usual colour codes in the range 0 to 7. For example, if f is set to one, then the character location that the attribute code controls will flash. Using this information, and the weights associated with each bit in a binary number (see Chapter 1), gives

```
128*f+64*b+8*paper+ink
```

for the value of the attribute code that produces the desired ink and paper colours and a flashing or bright character. So if you want a steady (f=0), bright (b=1), black ink (ink=0) and a white paper (paper=7) character you would POKE 64+7*8 into the attribute byte that controls the character location.

Just as the attribute file can be POKEd to change an attribute code, it can also be PEEKed to discover the current attribute code. In fact, it is unnecessary to calculate the address in the attribute file; the ZX BASIC function ATTR(line, column) returns the value stored in the memory location that controls the character location at line, column. Separating out the different parts of the attribute code to discover, for example, what paper colour is in effect is not difficult. To make it even easier the following user-defined functions can be used

```
DEF FNf(L,C)=INT(ATTR(L,C)/128)
DEF FNb(L,C)=INT(ATTR(L,C)-INT(FNf(L,C)*128)/64)
```

```
DEF FNp(L,C)=INT((ATTR(L,C)-INT(ATTR(L,C)/64)*64)/8)
DEF FNi(L,C)=ATTR(L,C)-INT(ATTR(L,C)/8)*8
```

where FNf returns the value of f, FNb returns the value of b, FNp returns the paper colour code and FNi returns the ink colour code.

The video driver

The previous sections have explained how the Spectrum's video display works in terms of the use and organisation of the display and attribute files. ZX BASIC insulates the user from considerations such as these by providing the PRINT statement. Within the ZX BASIC ROM there are machine code routines that examine the data items in a PRINT statement and store patterns of bits in the video RAM to produce the characters that represent the data. For example, PRINT "A" causes the machine code routines to store the dot pattern for the letter A in the character location where the printing cursor is currently positioned. For a numeric variable, the process of representing its value on screen is a little more complicated. For example, PRINT A causes the machine code routines to convert the number stored in A to a sequence of decimal digits, which are then displayed on the screen. Remember that the number stored in A, or any other numeric variable, is stored in binary form, and has to be converted to a string of decimal digits before it can be printed. In this sense PRINT A produces the same set of actions as PRINT STR$(A) (the function STR$ converts a numeric value to a string of digits).

The most obvious way to implement the machine code routines that execute the PRINT statement would simply be to write whatever was necessary to display each type of data on screen. Fortunately the authors of the ZX BASIC ROM thought before they started to program! As a result the software that implements the PRINT statement is split into two parts, the *PRINT routines* and the *video driver*.

The PRINT routines are responsible for converting each data item in a PRINT statement into a sequence of ASCII character codes. The video driver accepts these ASCII codes and is responsible for producing the pattern of dots on screen that represents each of the printable characters (see Fig. 6.2.) It is this separation into two parts that makes the Spectrum's I/O system so flexible. By the use of streams and channels (see Chapter 5) it is possible to associate a different device driver with the print routines and know that all it has

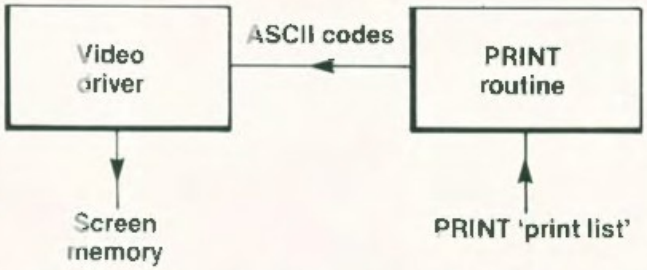


Fig. 6.2. Two separate software routines – the PRINT routine and the video driver – transfer data from RAM to the screen memory.

to handle is a sequence of ASCII codes. In the same way it is possible to send streams of ASCII codes to the video driver from another source. Without this separation it would be extremely difficult to redirect I/O.

On another level the use of ASCII codes as a means of communication between the PRINT routine and the video driver opens up a number of programming techniques. Printable characters, such as letters and digits, are not the only sort of item that occurs in a PRINT statement; there are also the control items, such as TAB, AT, INK and PAPER. Even these non-printable items are converted to ASCII codes by the PRINT routine before being passed on to the video driver. The ASCII codes corresponding to these non-printable items are referred to as *control codes*, because although they don't produce anything on the screen, they do control or affect the display. Following a control code, the next one or two ASCII codes may be interpreted as 'parameters' that govern the exact result produced. For example, the non-printable item INK 7 is converted by the PRINT routine to the control code 16 (standing for INK) and then the ASCII code 07 to represent the colour code. Notice that the colour code is sent to the video driver as ASCII code 07, not the digit 7, i.e. as CHR$(07) rather than CHR$(55). A table of control codes and their parameters can be seen below.

Non-printable item and action	code	parameters
, (move to next print zone)	6	None
cursor left	8	None
cursor right	9	None
ENTER	13	None
INK c	16	c

PAPER c	17	c
FLASH f	18	f
BRIGHT b	19	b
INVERSE i	20	i
OVER o	21	o
AT y,x	22	y x
TAB x	23	x 0

Notice that both AT and TAB are followed by two parameters, even though TAB only makes use of the first. There are control codes other than the ones listed in the table, but these extra codes are used by the program editor.

As the video driver receives nothing but a sequence of ASCII codes from the PRINT routine, it doesn't matter how they are generated. For example

```
PRINT INK 6
```

and

```
PRINT CHR$(16);CHR$(6);
```

both produce the same sequence of ASCII codes, so they have the same effect! By including control code sequences in strings it is possible to make messages that are self-positioning, or automatically set their own colours etc. For example in the following program

```
10 LET M$=CHR$(22)+CHR$(10)+CHR$(5)+"This
   message always prints in the same place"
20 PRINT M$
30 GOTO 20
```

the string M$ includes the control codes for AT 10,5, so it is always printed at the same place no matter where you try to print it!

The video driver can be used directly by the Z80 assembly language programmer to perform all the display operations for which the ZX BASIC programmer uses PRINT. To access the video driver, all that is needed is a RST 16 command after loading the A register with the ASCII code of the character that you want printed (or of the operation that you want performed). For example, to print the letter A you could use

```
LDA #65
RST 16
```

which first loads the A register with the ASCII code for A, and then does a restart (RST) to the video driver. (Note the machine code for RST 16 is 215 decimal.)

Finally, before moving on to consider other features of the Spectrum's video display, it is worth pointing out that the video driver not only changes the display file when a character is printed but also stores the current attribute code in the associated attribute byte. Commands such as INK and PAPER that occur outside a PRINT statement use the video driver. The only display commands that do not use the video driver are the CLS command and the high resolution graphics commands PLOT, DRAW and CIRCLE.

The character tables

An important part of the Spectrum's text-generating software is formed by the two character tables. These are used to hold the dot patterns of the various printable characters. Most of the Spectrum's character definitions are held in the main character table in ROM. As the table is in ROM it is not possible to change any of the definitions. However, as the address of the start of this table is held in the system variable CHARS it is possible to move the entire table to RAM and so give the Spectrum a completely user-definable character set! (see Chapter 7). The second character table is used to hold the definitions of the user-defined characters, and as you might expect, this is normally stored in RAM. However, the start address of this table is also held in a system variable UDG, and it, too, can be moved to start at any desired address.

The format of the data used to define the shapes of all the Spectrum's characters is identical to that used for the user-defined characters. That is, the 64 dots that make up a character are stored in eight memory locations. Each memory location holds eight bits that represent the state of the eight dots that make up a single row of the character. With this in mind, it is not difficult to see that if the start of the main character table is START, the eight memory locations that contain the definition of CHR$(I) are given by:

```
START+8*(I-32)
```

(The first printable character is CHR$(32).) This equation can be used to print out the dot pattern of any letter:

```
 10 DEF FNt()=256+PEEK(23606)+256*PEEK
    (23607)
 20 INPUT C$
 30 LET I=CODE(C$(1))
 40 LET A=FNt()+8*(I-32)
 50 FOR K=A TO A+7
 60 LET D=PEEK(K)
 70 GOSUB 1000
 80 IF LEN(B$)<8 THEN LET B$="0"+B$:
    GOTO 80
 90 PRINT B$
100 NEXT K
110 GOTO 20

1000 LET B$=""
1010 LET B=D-INT(D/2)*2
1020 IF B=0 THEN LET B$="0"+B$
1030 IF B=1 THEN LET B$="1"+B$
1040 LET D=INT(D/2)
1050 IF D=0 THEN RETURN
1060 GOTO  1010
```

Line 10 gives a function FNt() which returns the start address of the main character table. (The address stored in the system variable CHARS is in fact 256 less than the address of the start of the table.) The rest of the program simply PEEKs the eight memory locations that store the dot pattern for the character in C$(1), and print the pattern in binary.

The same program can be used to discover the shape of any of the user-defined characters by changing line 10 to:

```
10 DEF FNt()=PEEK(23675)+256*PEEK(23676)
```

This returns the start of the user-defined graphics table by PEEKing the system variable UDG. Line 40 also has to be changed to:

```
40 LET A=FNt()+8*(I-144)
```

There are many direct uses for the character definition tables, and some of these will be explored in the next chapter. The key facts to remember are that the start address of both tables can be altered, and if the tables are stored in RAM the character definitions can be changed using POKE.

The video system variables

The video system variables are involved in a wide range of tasks associated with the Spectrum's video display. The variables CHARS (23606) and UDG (23675) have already been described in the last section as holding the start of the main character table and the user-defined character table respectively. Other variables of interest include:

CO ORDS (23677 and 23678)

This system variable gives the x co-ordinate (in 23677) and the y co-ordinate (in 23678) of the last point plotted by a high resolution command. By PEEKing these two locations the current position of the graphics cursor can be found. For example, if you want to draw a line from the current position of the graphics cursor to the absolute location X,Y then use:

```
DRAW X-PEEK(23677),Y-PEEK(23678)
```

This works by PEEKing the current location of the graphics cursor and working out the difference between it and the desired position.

S POSN (23688 and 23689)

These two locations are used to hold the current location of the text cursor. To be precise, the current position of the text cursor is:

```
column number = 33-PEEK(23688)
line number   = 24-PEEK(23689)
```

DF CC (23684 and 23685)

DF CC holds the same information as S POSN – the current position of the text cursor – but instead of its line and column number, DF CC holds the corresponding address within the display file.

SCR CT (23692)

This system variable holds a 'countdown' to the next occurrence of the question "scroll ?". Its value is always one more than the number of scrolls that will be performed before the next "scroll ?" message appears. Repeatedly POKEing this system variable with 255 while a program runs will ensure that the message never appears.

ATTR P and ATTR T (23693 and 23695)

These two variables hold the current value of the permanent and temporary attribute codes respectively. In other words, if there are no attribute commands within a PRINT statement then the value stored in ATTR P is used to set the attribute bytes corresponding to each of the character locations used. However, if there are any attribute commands within a print statement, then the value in ATTR T is set accordingly, and used in place of ATTR P for the duration of the PRINT statement.

MASK P and MASK T (23694 and 23695)

These two system variables hold the permanent and temporary transparent attributes. Normally whenever a character is printed either the attribute code in ATTR P or ATTR T is stored in the corresponding attribute byte. However, if attribute 8 is used in any of the attribute commands (for example INK 8) then that part of the attribute byte is left unchanged. MASK P and MASK T are used to record which attributes are either permanently or temporarily set to transparence. The coding is such that any bit that is a one in MASK P or MASK T indicates that the corresponding bit will be taken from the existing attribute code rather than ATTR P or ATTR T.

BORDCR (23624)

This system variable holds the attribute code used in the lower half of the screen. The ink part of the attribute code is also used to set the border colour.

Creative video

Although much has been explained in this chapter about the workings of the video display, a great many of its features have not been explicitly discussed. Most of these are obvious once you have understood the overall workings of the display, and the next chapter presents a number of examples that use knowledge of how the video display works. Much the best way of coming to terms with the Spectrum's video display is to go ahead and use it creatively.

Chapter Seven

Video Applications

This chapter presents a number of examples of the less than obvious uses of the Spectrum's video display. A number of the examples could form the basis of routines suitable for use in applications programs. However, their main value is to suggest the range of things that can be achieved with the Spectrum's display without changing a single chip!

Functional characters

Although it is obvious that the user-defined character table is nothing more than a sequence of memory locations like any other, there is a tendency to think about altering it using only the standard statement:

POKE USR "char"+n,BIN bit pattern

This is such a familiar statement that it is worth examining its components more carefully. The parameter of the USR function is normally the address of a machine code program (to which USR transfers control). However, when the parameter of USR is a string expression, such as USR "char", it works out the address of the first memory location in the user-defined character table for 'char'. You should be able to see that this means USR "char"+n evaluates to the address of the memory location that holds the bit pattern of row n of the user-defined character 'char'. For example, if you try

PRINT USR "A"

the Spectrum will print the address of the first memory location in the user-defined character table. The rest of the statement used to define the dot pattern should now be obvious. The result of the POKE is to store the bit pattern that is the parameter of the BIN in

the memory location that defines row n of the user-defined character 'char'.

Of course, once you have the start address of the eight memory locations that control the dot pattern of a user-defined character you can change them in any way you like. For example, try

```
10 LET A=USR "b"
20 FOR I=0 TO 7
30 POKE A+I,INT(RND*256)
40 NEXT I
50 PRINT AT 10,10;CHR$ 145;
60 GOTO 20
```

which produces a moving explosion character. Line 10 finds the start address of the definition of user-defined character "b", or CHR$ 145. Lines 20 to 40 POKE random values as the definition of each row of the character, and line 50 repeatedly PRINTs the new 'random' shape. (CHR$ 145 is used to avoid any ambiguities in the program.)

This functional definition of a random explosion character can be extended to include new characters that are simple functions of existing characters. For example, try

```
10 LET A=USR "a"
20 LET B=USR "b"
30 FOR I=0 TO 7
40 LET D=PEEK(A+I)
50 POKE B+7-I,D
60 NEXT I
70 PRINT CHR$ 144,CHR$ 145
```

which will define CHR$ 145 as an upside-down version of CHR$ 144! The key to this program is to be found in lines 40 and 50. Line 40 PEEKs the definition of row I of CHR$ 144 and then line 50 POKEs it into row 7−I of CHR$ 145. Similar methods can be used to define characters that are reflections and rotations of other characters.

Changing the character set

As the system variable CHARS holds the address of the start of the standard character table (less 256) it is quite easy to move the entire character table to RAM and then change any or all of the definitions. For example

```
 10 CLEAR 32768-1024
 20 LET A=256+PEEK 23606+256*PEEK 23607
 30 LET B=32768-1024+1
 40 FOR I=0 TO 95
 50 FOR J=0 TO 7
 60 LET D=PEEK(A+I*8+J)
 70 POKE B+I*8+7-J,D
 80 NEXT J
 90 NEXT I
100 POKE 23607,INT((B-256)/256)
110 POKE 23606,(B-256)-INT((B-256)/256)*256
```

will transfer the entire standard character table to RAM while changing the order of each row of dots to invert each character. You should be able to recognise the components of this program. Line 10 reserves 1K of memory which is more than enough for the character set. If you have a 48K Spectrum then you can change 32768 to 2*32768. Line 20 finds the location of the character table in ROM, and line 30 stores its new position in B. Lines 40 to 90 do the actual job of moving the character table, and you should recognise lines 60 and 70 as being very similar to the inversion of the user-defined characters in the last section. Finally, lines 100 and 110 POKE the new value for the start address of the character table.

It is a good idea to SAVE this program before running it: it is more than a little difficult to make any changes to it with the entire character set inverted. Running the program a second time will not restore the character set to its original form; instead it mixes it up still further!

Internal animation

Although each of the character tables is organised into groups of eight memory locations that correspond to the shape of a single character, there are times when it is worth thinking about the table as a whole. For example, if the user-defined character table is set up so that each character is a letter in a message, then the message can be printed using a smooth scrolling motion by repeatedly printing the first user-defined character and moving the start address of the table itself. For example, if the system variable UDG contains the usual address of the first memory location of the user-defined character table, then PRINT CHR$ 144 will display the first user-defined character. However, if the value in UDG is increased by one,

PRINT CHR$ 144 will display the last seven rows of the first user-defined character and the first row of the second. By continually incrementing the value in UDG, the eight memory locations that define the shape of the first user-defined character can be made to move through the original table rather like a viewing window:

```
10 GOSUB 1000
20 FOR I=0 TO 20*8
30 PRINT AT 10,10;CHR$ 144;
40 POKE UDG,D+I-INT((D+I)/256)*256
50 POKE UDG+1,INT((D+I)/256)
60 NEXT I
70 GOTO 20

1000 LET UDG=23675
1010 LET D=PEEK UDG+256*PEEK(UDG+1)
1020 RETURN
```

This program will produce a smoothly scrolling message consisting of the default user-defined characters, i.e. A to U. Subroutine 1000 stores the start of the user-defined character table in D. The FOR loop at 20 to 60 PRINTs the first user-defined character and then moves the start of the table up by one memory location.

This technique of moving the start of a character table can be used to produce remarkably smooth internal animation using nothing but ZX BASIC. For a practical example of this see the 'Fruit Machine' game in *The Spectrum Book of Games* by Mike James, S. M. Gee and Kay Ewbank, published by Granada.

Free characters

The 'free' in this headline refers to the positioning of the characters rather than the discovery of any extra ones! You now know how the display file controls the dot pattern on the screen, so you should be able to see that the restriction of characters to character locations is more a convenience than a necessity. In fact, the only real reason for not allowing characters to be placed at any point specified by high resolution co-ordinates is that the attribute bytes control whole character locations. If you are not worried about a mismatch between the area a character occupies and the area an attribute byte controls, then it is easy to produce characters anywhere on the

screen. For example, the following program prints an "X" in the usual way and then 'plots' the number "2" as a superscript, so producing the familiar notation for x squared.

```
10 PRINT AT 10,10;"X"
20 LET X=95
30 LET Y=99
40 LET C$="2"
50 GOSUB 5000
60 STOP

5000 LET A=256+PEEK 23606+256*PEEK 23607
5010 LET A=A+8*(CODE C$-32)
5020 FOR I=0 TO 7
5030 LET D=PEEK(A+I)
5040 FOR J=0 TO 7
5050 LET B=D-INT(D/2)*2
5060 LET D=INT(D/2)
5070 IF B=1 THEN PLOT X,Y
5080 LET X=X-1
5090 NEXT J
5100 LET Y=Y-1
5110 LET X=X+8
5120 NEXT I
5130 RETURN
```

The actual work is carried out by subroutine 5000, which will plot the character stored in C$ at the position in X and Y. (X and Y are the co-ordinates of the top right hand corner of the 8 by 8 square of dots that constitutes the character). The principle involved is simple. The bytes that define each row of the character in question are retrieved one after the other. Each byte is broken down into its pattern of zeros and ones by the inner FOR loop 5040 to 5090, and an ink dot is PLOTted if the bit stored in B is 1. The rest of the subroutine is concerned with moving the value stored in X and Y on to form the 8 by 8 square of dots.

For another example of the use of this subroutine change the main program to:

```
10 LET A$="A MESSAGE"
20 LET X=10
30 LET Y=170
40 FOR K=1 TO LEN(A$)
50 LET C$=A$(K)
60 GOSUB 5000
```

```
70 LET X=X+4
80 NEXT K
90 STOP
```

This uses subroutine 5000 to print the message stored in A$ diagonally down the screen. The only extra information you need to understand how this program works is that after subroutine 5000 has finished, the variables X and Y contain the co-ordinates of the bottom right-hand corner of the character just plotted.

Variable size characters

A very small change to the method used to produce characters at any location will allow for characters of any size at any location! The basic principle is to PLOT more than one point for each bit in the character definition. If you make the following changes to the last program you will see a range of character sizes:

```
10 LET A$="ABCDE"
20 LET X=30
35 LET SX=5:LET SY=5
65 LET X=INT(X+SX/2)
66 LET SX=SX-1
67 LET SY=SY-1


5070 IF B=1 THEN GOSUB 6000
5080 LET X=X-SX
5100 LET Y=Y-SY
5110 LET X=X+SX*8

6000 FOR M=1 TO SY
6010 FOR N=1 TO SX
6020 PLOT X+N,Y+M
6030 NEXT N
6040 NEXT M
6050 RETURN
```

Subroutine 6000 plots a square or a rectangle of dots SX wide by SY high, and thus SX and SY are the X and Y scale factors respectively. The modifications to the main program call subroutine 5000 with a decreasing pair of scale factors to plot a diagonal line of characters, each one smaller than the last.

Smooth screen scrolling

One of the tasks carried out by the Spectrum's video software is the vertical scrolling of the screen. This apparently simple operation is in fact much more involved than you might imagine. In principle, all the software has to do is move each group of eight dot rows that constitute a line of characters to the position in memory formerly occupied by the dots that formed the line of characters immediately above. In addition to moving the dots in the display file, the scroll software must also shift the attribute bytes up by the equivalent of one text line in the attribute file. If you recall the odd layout of the display file, you will start to appreciate the difficulties in moving the data to perform a scroll. As the display file is stored in three sections, each consisting of eight character lines, the real difficulty comes from having to move the top line of each section to the bottom line of the next storage section. (See Chapter 6 if you have forgotten the details of the display file's layout.) All in all, a vertical scroll is difficult enough to leave to the Spectrum's built-in software. However, a horizontal scroll is much easier.

There are many applications programs, especially games, that involve moving graphics or text smoothly to the left or the right. For example, a typical games program might produce the effect of an attack ship flying over a landscape by holding the position of the ship fixed and 'scrolling' the landscape horizontally. A horizontal scroll by one dot (moving the display, or an area of the display, to the left or right by one dot) is not difficult, but it does need some Z80 assembly language.

To scroll the screen right by one dot all you have to do is to start at the left-hand side of each row of dots and move them all along by one dot. The dot that 'falls off the end' of the row is lost, and a paper dot is shifted into the first position of the row. As long as you start from the beginning, the display file is organised so that each group of 32 bytes holds the dot pattern for a complete row. This means that shifting a row can be achieved by shifting the contents of the 32 bytes 'up' by one bit. That is, the contents of the first memory location in the display file are shifted one bit to the right so that b1 becomes b0, b2 becomes b1 and so on. A zero has to be supplied as the new value of b7, and the value of b0 has to be saved so that it can be shifted into b7 of the next memory location. The same operation is repeated on the second memory location, and so on to the last memory location involved in storing the row of dots. Each time, all bits in the memory location are moved one place to the right and b0

from the previous memory location is moved into b7. This operation on a single memory location is called a 'rotate right' in Z80 assembly language, so shifting the screen one bit to the left is a matter of repeatedly applying a rotate right operation to each of the 32 memory locations that store the pattern of a row of dots.

An assembly language routine to shift the top third of the screen (text lines 0 to 7) one dot to the right is given below.

address		assembler	code	comment
23296		LD HL,16384	33,0,64	load the HL register
23299		LDA 63	62,63	load A with 63
23301	LOOP1	LDB 32	6,32	load B with 32
23303		ANDA	167	clear the C flag
23304	LOOP2	RR (HL)	203,30	rotate A right 1 bit
23306		INC HL	35	add one to HL
23307		DJNZ LOOP2	16,251	B=B−1 and loop if B<>0
23309		DECA	61	A=A−1
23310		JR NZ, LOOP1	32, 245	loop if A<>0
23312		RET	201	return to BASIC

This routine can be loaded into the printer buffer and called using USR 23296 each time the screen is to be scrolled by one bit. In the second line, LDA 63 sets the number of dot rows that are shifted – in this case 63 plus one, i.e. 64. Although the routine has been hand-assembled as if it was going to be run starting at 23296, it is in fact position independent and can be loaded anywhere in memory. The following BASIC program demonstrates the routine:

```
10 DATA 33,0,64,62,63,6,32,167,203,30,35,
   16,251,61,32,245,201
20 FOR I=23296 TO 23312
30 READ D
40 POKE I,D
50 NEXT I

60 PRINT AT 7,0;"ABCDE"
70 PRINT AT 8,0;"ABCDE"
80 LET A=USR 23296
90 GOTO 80
```

Lines 10 to 50 load the machine code into the printer buffer. Lines 60 to 90 PRINT something on the screen and then use the routine to shift the message at line 7 off the screen.

As an example of how this horizontal scroll routine could be used in a game try substituting

```
 60 LET Y=120
 70 LET S=1
 80 IF RND<.2 THEN LET S=-1*S
 90 IF Y=115 THEN LET S=1
100 IF Y=174 THEN LET S=-1
110 LET Y=Y+S
120 PLOT 0,Y
130 PRINT AT 2,10;"*";
140 LET A=USR 23296
150 GOTO 80
```

in the previous program. This draws an asterisk in a fixed position and a 'landscape' which is shifted across the screen. This creates the impression of an asterisk 'flying' through the landscape in a way that would be impossible to achieve using ZX BASIC alone.

Conclusion

The examples in this chapter have all been short enough to make them easy to try out. However, they are also long enough to illustrate the ideas involved and to make them worth building into your own programs. For example, the horizontal scroll program could easily be turned into a good-quality action game, with no reason to use any more assembly language than the USR routine supplied. On the other hand if your interest doesn't lie in games the same routine can be used to plot moving graphs that imitate the display on an oscilloscope.

No matter what you might have been told, computing is an experimental subject, and experiments lose much of their value if you only read about what is supposed to happen! So it is important to incorporate these examples into your own programs and experiment with them.

Chapter Eight

Tape, Sound and the Printer

The Spectrum's cassette interface and its limited sound generator use the same hardware within the ULA. However, the key factor that links together all three of this chapter's subjects - the cassette interface, the sound generator and the ZX printer - is the presence of standard software within the ZX BASIC ROM to control them. Loosely speaking, all three come under the heading of 'standard' I/O devices. Apart from these tenuous connections there is not much carry-over from one device to the next, and as a result this chapter is in three main parts corresponding to the tape system, the sound system and the ZX printer.

The tape system

One of the best features of the Spectrum is its remarkably reliable tape system. It is not complex, either; it seems to be attention to detail that produces the reliability. There is not very much that can be done to change the way the tape system works, or to add extra facilities, without becoming involved in excessively large Z80 assembly language programs. This is not to say that things cannot be changed or improved; it is just that there is nothing worth tinkering with. If you are planning to do anything with the tape system, then what you need to know depends very much on what you have in mind. For example, if you are interested in writing new tape software for the Spectrum, or if you want to read tapes produced by other machines, then you will need to know about the way that the hardware is organised. If you are going to try to read Spectrum tapes using other machines, then the format used to write the data is important. On the other hand if you are producing Z80 assembly language applications programs, then knowledge of the machine code routines that perform the reading and writing of tape files is

again crucial. To cover this range the description of the tape system is divided into three parts: hardware, tape format and software details.

Tape hardware

The main features of the tape hardware have already been described in Chapter 2, but without any attempt to explain how they are used to produce tape storage of data. Both the line that sends data to the cassette (MIC), and the line that receives data from the cassette (EAR), are connected to the same pin (28) of the ULA. This pin of the ULA responds to I/O port address 254 as already described in Chapter 2.

Writing to I/O port 254 sets the cassette line MIC depending on the state of bit 3. If bit 3 is 0 then the output voltage is 0.75 volts. If bit 3 is 1 then the output voltage is 1.3 volts. By alternately writing a 0 and a 1 to b3 of port 254 a square wave can be sent to the cassette (see Fig. 8.1). This square wave is recorded as an audio tone with a

Fig. 8.1 The square wave sent to cassette line MIC from I/O port 254 by setting b3 alternately to 0 and1.

fixed volume and a frequency that depends on the time that bit 3 remains constant. The longer the time interval between 'flips' of bit 3, the lower the pitch of the tone. For example, the BASIC program,

```
10 OUT 254,0
20 OUT 254,8
30 GOTO 10
```

changes bit 3 of port 254 from 0 to 1, and the resulting square wave can be recorded by pressing play and record in the usual manner. Notice also that while this program is running the border colour changes to black. This is because b0, b1 and b2 of output port 254 control the border colour, and both OUT instructions set these bits to 0. The frequency of this tone is very low, because ZX BASIC cannot be used to change the state of the bit often enough to produce a high frequency. However, using Z80 assembler there is no problem in changing the state of bit 3 fast enough to produce tones that are above the range of normal hearing! The lowest level tape software

uses this simple method to produce a number of tones that code the data onto the cassette recorder.

If a tape with audio tones recorded on it is played back while connected to the Spectrum, the level of the input determines the state of b6 of input port 254. In fact the input level determines b6 of any input port corresponding to an address with b0 set to 0 (see Chapter 2). If the voltage on the input line (EAR) to the Spectrum is low, then b6 is 0; if the voltage is high, then the bit is 1. (In practice the Spectrum sets the output line controlled by b3 high before reading the input line, and so the tape recorder signal pulls the normally high input low.) To see the way that the signal from the tape recorder affects b6 of the input port try:

```
10 OUT 254,8
20 PRINT IN 254
30 POKE 23692,255
40 GOTO 20
```

Line 10 sets the output to MIC high before line 20 reads and PRINTs the state of port 254. Line 30 simply stops the 'Scroll?" message from periodically halting the program. If you play a recorded tape while running this program you will see the numbers 255, corresponding to b6 high, and 191, corresponding to b6 low, printed on the screen. Notice that pressing keys on the keyboard also alters the value returned by IN 254.

The signal that results from playing back a recording of a square wave is often far from being a good approximation to the original square wave (see Fig. 8.2). However, the time between high-to-low

Fig. 8.2 Typical signal from a cassette recorder playing back the square wave shown in Fig. 8.1.

and low-to-high changes is likely to be fairly close to the original. In other words, the pitch of the square wave is likely to be the same unless the speed of the tape recorder has changed (because, for example, its batteries have run down). Thus it is the time between the changes in signal level that the Spectrum uses to retrieve the data recorded on tape.

Tape format

All spectrum tape files are recorded as two blocks of information,

the *header* and the *data block*. The header is a short burst of audio tone that is used to store information concerning the data stored in the data block that follows. For example, the header is used to store the file name and the number of bytes in the data block. The exact format of the data stored in the header block will be described later.

Each type of block begins with a burst of leader tone: roughly 5 seconds of leader for a header block, and around 2 seconds for a data block. Leader tone is a square wave with 619.4μS (1μS=one microsecond or one millionth of a second) between each change of state (see Fig. 8.3). This corresponds to a frequency of around

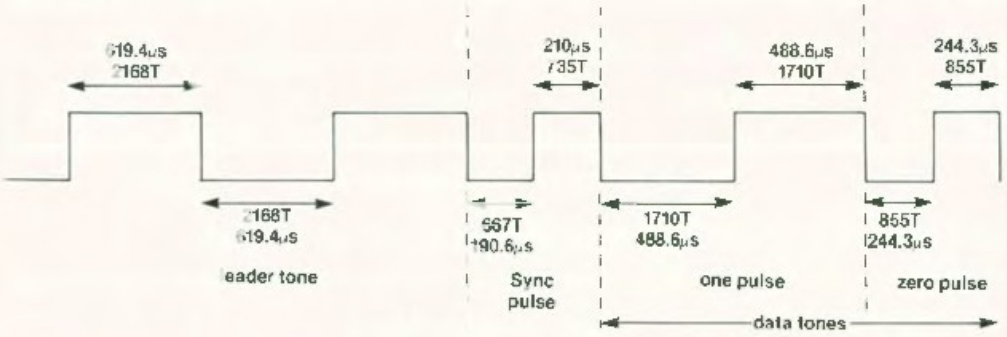


Fig. 8.3. The signals used for tape storage by the Spectrum, showing timings in microseconds and the number of Z80 T states per pulse.

307Hz. The end of the leader tone is marked by the occurrence of a pulse of very much shorter duration – the *sync pulse*. The sync pulse is low for 190.6μS and high for 210μS. Following the sync pulse, and without any break, comes the first data pulse. The length of a data pulse depends on whether it represents a 0 or a 1. If it represents a 0 it is low for 244.3 μS and high for the same amount of time. If it represents a 1 the pulse lasts for exactly twice as long. All of this timing information can be seen in Fig. 8.3 (along with the number of Z80 T states for which each pulse lasts).

Using this information it should be possible to write an assembly language program on almost any machine to enable Spectrum tapes to be read or written. The process of reading the data back in is made very reliable on the Spectrum by the use of a range of times that are acceptable for each type of pulse. The data pulses that follow the sync pulse form groups of eight bits that correspond to the bytes that are being saved or loaded. In other words, the first eight pulses that follow the sync pulse form the first byte of data, the next eight the second byte, and so on to the end of the block. The only other information necessary is the format of the bytes that form the header, and how these relate to the data block.

A header is composed of 19 bytes of data, but only 17 of these are supplied by the user. The first byte of the header or of the data block is a *FLAG byte*, generated by the SAVE routine to mark the difference between a header and a data block. The FLAG byte is 0 if the following block is a header, and 255 if the block contains data. The final byte of the header or of a data block is a *parity byte* which is used to detect any loading errors that might have occurred. These two bytes, the flag byte at the start of the block and the parity byte at the end of the block, are added by the SAVE software to both header and data blocks, thus making them two bytes longer than you might expect. The use of the 17 bytes that form the header proper can be seen in Fig. 8.4. The first

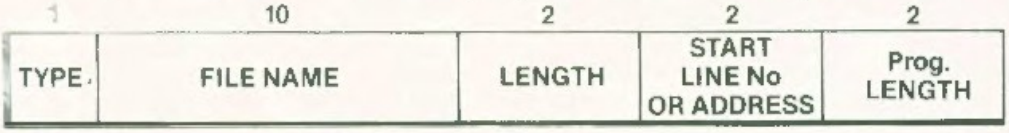


Fig. 8.4. Format of a header block.

byte is used to record the type of data block, as follows:

TYPE	type of data block
0	BASIC program
1	Numeric array
2	String array
3	Machine code or screen dump

The next 10 bytes hold the file name. Following the file name come the two bytes that record the length of the data block that follows. The use of the remaining four bytes depends on the type of data block that the header describes. If TYPE is 0 then bytes 14 and 15 hold the line number for an auto start BASIC program, and bytes 16 and 17 hold the number of bytes in the program part of the file. (Remember that SAVEing a BASIC program saves both the program area and the data area.) If TYPE is 1 or 2 then byte 15 is the only one used, and this holds the name of the array. If TYPE is 3 then only bytes 14 and 15 are used, and these hold the address from which the data bytes should be loaded.

Following the header comes the data block that it describes. As already mentioned, this contains two more bytes than recorded in the length information in the header, the leading *type byte* and the trailing *parity byte*. Following the type byte, each byte in the data block can be regarded as an 'image' of the portion of memory that was saved.

The only detail left to describe is the exact way that the parity byte is used to detect any loading errors. When either a header or a data block is saved, each byte that is written out is exclusive ORed with the parity byte. The parity byte's initial value is given by the flag byte. If on reading the data back a parity byte is built up in the same way, that is by forming the exclusive OR of the current parity byte with each byte read in, then in the absence of read errors the final value of the parity byte will be 0. Notice that this assumes that the initial value of the running parity byte was 0, and that all of the bytes that are read in, including the flag byte and the final trailing parity byte, are exclusive ORed with it.

The SAVE and LOAD routines

There are two fundamental machine code routines within the ZX BASIC ROM that can be used to SAVE and LOAD an area of memory. As with any ZX BASIC ROM routines there is always the possibility that their position might move, but for two such important routines this seems unlikely.

The save routine starts at address 1218 (or 04C2 hex). What it actually does depends on a number of parameters passed using the registers:

Register	action
DE	number of data bytes to be saved
IX	address of first data byte
A	0 for header 255 for data block

It is worth noting that this is a low level routine that will save a memory area without any frills or alterations, apart from the addition of the leading type byte and the trailing parity byte. In particular, this routine doesn't issue any messages about pressing play and record, and it doesn't automatically form a header block if you are using it to write a data block. In fact if you want to write out a header block you must create a 17-byte area of memory that contains the 17 bytes of correctly initialised header data, e.g. the file name, length etc. Unless you have some very special application in mind, the save routine is generally used twice, once to save the header and once to save the data block that the header describes.

The load routine starts at 1366 (or 0556 in hex) Once again its action depends on a number of parameters:

Register	action
DE	number of data bytes to be loaded
IX	address that first byte should be stored in
A	0 means load header 255 means load data block

If the carry flag is reset the data will not be loaded into memory; instead it will be compared to what already exists, i.e. a verify operation will be performed. Thus the carry flag has to be set to actually load data. If the wrong type of file is found, then the routine returns with both the carry flag and the zero flag reset. If a loading error is detected then both the zero flag and the carry flag are set. Notice that to use the load routine you have to know how many bytes you are trying to load. If you are trying to load a header block then this is easy, as all headers are 17 bytes long. If you are loading a data block then the only way you can know how many bytes to load is by reading the header that preceded it.

Using the save and load routines it is possible to write and read non-standard tape files. For example, you can write a file composed of a number of data blocks of fixed size that could be read in as and when they were required. However, the main problem with using the Spectrum's tape system in any way that is non-standard is the lack of cassette motor control. If a file was written as a collection of blocks, the user would have to start and stop the tape as requested by the Spectrum!

As an example of using the tape load and save routines, the following program will print out a list of file types and names. In this sense it forms a limited catalogue command. The first part of the program takes the form of an assembly language subroutine that reads headers from the tape and stores them in the printer buffer.

assembly	**language**	**code**	**comment**
23296	LOOP LD DE,17	17,17,0	length of header in DE
23299	XOR A	175	clear A
23300	SCF	55	set carry flag

23301	LD IX,23311	221,33,15,91	start of data area in IX
23305	CALL 1366	205,86,5	CALL load routine
23308	JR NC,LOOP	48,242	jump back if not header
23310	RET	201	return to BASIC

The code of this routine is loaded into the printer buffer. It also stores the header bytes in the printer buffer starting at 23311. The following BASIC program uses the routine to read in headers and print their type and filename:

```
10 DATA 17,17,0,175,55,221,33,15,91,
   205,86,5,48,242,201
20 FOR A=23296 TO 23310
30 READ D
40 POKE A,D
50 NEXT A

60 LET A=USR 23296
70 PRINT "TYPE=";PEEK 23311;
80 PRINT " NAME=";
90 FOR I=1 TO 10
100 PRINT CHR$(PEEK(23311+I));
110 NEXT I
120 PRINT
130 GOTO 60
```

Lines 10 to 50 load the machine code into the printer buffer. Line 60 uses the routine to load the header bytes and lines 70 to 120 print the type and name. It would be quite easy to extend the last part of the program to PEEK more of the header data and provide information such as length of file and load point.

Sound

The Spectrum's sound generator is closely connected to the tape system. The small loudspeaker that produces the sound is connected to the same output pin of the ULA as EAR and MIC. The only difference is that the output is controlled by b4 of I/O port 254. If b4 is 0 then the output voltage is .75 volts, and if b4 is 1 then the output voltage is 3.3 volts. Notice that the voltage range obtained by changing b4 is greater than that used with the tape system. The lower voltage used by the tape system is insufficient to drive the

loudspeaker, and so the tape signals cannot be heard, but the higher voltage used to drive the speaker does appear at both EAR and MIC.

The basic method of making a sound is identical to the method used to generate tones for the tape system. A square wave can be produced by changing b4 repeatedly from 0 to 1 and back to 0 again. The pitch of the sound that the square wave produces is related to how fast the wave form repeatedly changes from high to low. Apart from the pitch of the note, there is nothing else that can be changed. The volume is fixed by the range of voltages corresponding to the two states of the square wave, and the overall sound quality is set by the shape of the wave form. As an example of controlling the speaker directly try:

```
10 OUT 254,16
20 OUT 254,0
30 GOTO 10
```

This program simply changes b4 from 1 to 0 each time through the loop. The sound that results is very rough and low pitched due to BASIC's lack of speed. Also notice that the border colour changes to black because b0 to b2 of port 254 control the border colour.

The Spectrum's sound command, BEEP, does a remarkably good job of producing an accurate musical scale. This is yet another example of how the Spectrum's excellent software makes the most of a limited hardware feature. If you would like to know more about the creative use of the Spectrum's BEEP command then see *The Spectrum Programmer* by S. M. Gee, published by Granada.

Improving the Spectrum's sound is very difficult without the use of extra hardware. However, the following assembly language routine will drive the I/O port directly using a table of values as the data.

---

address		assembler	code	comment
23296		LD B,count	06,0	number of bytes in table
23298		LD HL,(table)	237,107,23,91	start of table
23302	LOOP	LD A,(HL)	126	load data to A
23303		OR 8	246,8	set MIC bit
23305		OUT (254),A	211,254	send data to port
23307		LD C,time	14,0	load delay time
23309	DEL	DEC C	13	delay loop
23310		JP NZ,DEL	194,13,91	jump back if C<>0

23313	INC HL	35	next data byte
23314	DEC B	5	end of table?
23315	JP NZ,LOOP	194,6,91	jump back for rest of table
23318	RET	201	return to BASIC
23319	DEFW table		address of table

The best way to explain the use of this routine is by way of a BASIC example. White noise is the sort of 'sshhing' noise you can hear on a radio that is tuned between stations. It is in fact a roughly equal mixture of a very wide range of frequencies. The Spectrum can be made to produce an approximation to white noise by changing the value of b4 of output port 254 at random. The only problem is where to get a table of random bits from. Surprisingly, the easiest source of very nearly random bits is the ZX BASIC ROM itself. The following BASIC program uses the routine given above to send 256 bytes of the BASIC ROM to the output port.

```
10 DATA 6,0,237,107,23,91,126,246,8,
   211,254,14,0,13,194,13,91,35,
   5,194,6,91,201
20 FOR A=23296 TO 23318
30 READ D
40 POKE A,D
50 NEXT A

60 POKE 23319,0
70 POKE 23320,20
80 POKE 23297,255
90 POKE 23308,128
100 LET A=USR 23296
110 GOTO 100
```

The first part of the program loads the machine code into the printer buffer. Lines 60 to 90 set up the parameters used to control the routine. Before using the routine, memory locations 23319 and 23320 should be set to hold the address of the start of the data table. Memory location 23297 should be set to the number of data bytes in the table, and memory location 23308 should be set to produce the desired overall pitch. Lines 100 and 110 repeatedly call the user routine and the result should be a hissing, crackling sound. As there is no attempt to restrict the data sent to the I/O port to b4, the border colour also changes randomly.

Using this routine with data tables set up in regular patterns it is possible to make a limited range of sound effects. For example, make the following changes to line 60 onwards in the last program:

```
  60 GOSUB 1000
  70 POKE 23319,25
  80 POKE 23320,91
  90 POKE 23307,N
 100 POKE 23308,250
 110 LET A=USR 23296
 120 PAUSE 1
 130 GOTO 110

1000 LET N=220
1010 LET S=1
1020 LET D=0
1030 LET C=1
1040 FOR I=0 TO N-1
1050 IF I=S THEN GOSUB 2000
1060 PRINT D
1070 POKE I+23321,D*16
1080 NEXT I
1090 RETURN

2000 LET S=INT(S+C)
2010 LET C=C+.25
2020 IF D=0 THEN LET D=1:RETURN
2030 LET D=0
2040 RETURN
```

Subroutines 1000 and 2000 set up a table of values in the printer buffer such that the frequency of the waveform decreases with time. The resulting noise is a sort of 'zap' sound. You can experiment with changing the pattern of 0s and 1s produced by subroutine 1000 to create your own sound effects.

The ZX Printer

The ZX printer is a remarkably cheap way of obtaining hard copy listings and graphics. Perhaps its only shortcomings are the inaccuracy of its dot positioning and the quality of the aluminium coated paper that it uses. (It is worth pointing out, however, that the aluminium paper does photocopy particularly well!)

The ZX printer works by evaporating the aluminium coating

from a roll of black paper. Where the aluminium is removed, the black shows through, and it is this rather than any sort of ink that makes the printing stand out. The ZX printer uses a spark produced by two travelling metal points or *styli* to evaporate the aluminium, and if you operate the printer in the dark, the blue electrical flashes can clearly be seen just below the tear bar. The styli are mounted on opposite sides of a moving band, so that at any time one of them is positioned over the paper. As the paper is scanned by the styli it is moved up by an electric motor, so that each scan can be used to print a new row of dots on the paper.

The software that drives the ZX printer is fairly complete, and there is very little that can be done to improve it. However, the way that the ZX printer is controlled is interesting in itself as an example of the way computers can be used to control external equipment. And knowing the way that the printer works may suggest novel applications.

The printer is connected to I/O port 251. Reading the port provides information concerning the current status of the printer. If a printer isn't connected, then b6 will be 1 (and conversely b6 will be 0 if a printer *is* connected). The state of b7 reflects the position of the styli. If either of them are positioned on the paper, b7 is 0. Thus b7 can be monitored to discover when a stylus first comes over the paper ready to print a line of dots. The speed that the styli scan the paper varies depending on the loading on the motor. To overcome this difficulty, an encoder disc is attached to the motor. This causes b0 to pulse around 256 times as a stylus scans a line. Thus if the production of dots is tied to the pulsing of b0 the dots will be evenly spaced no matter how fast or slow the motor is running.

On output to port 251, bits b1 and b2 control the motor. If b2 is 0 then the printer's motor starts. If b1 is 0 then the motor runs fast, otherwise it runs at a slower speed. This slower speed is used to print the last two scan lines so that the styli can be stopped off the paper, ready to print the first line the next time the printer is used. Finally b7 controls the voltage on the styli. If b7 is 1, then the styli are at high voltage and the resulting spark will burn a black mark on the paper.

Apart from the way that the bits signal the state of the printer and control its operation, there are one or two details of operation that are important. Firstly, the stylus voltage must be off to detect when they reach the edge of the paper. The reason for this is that the presence of the stylus voltage automatically sets b7 of the input high. Secondly b0 and b7 are both *latched* until some data is written to the I/O port. ('Latched' is electronic jargon for 'held steady until

otherwise instructed'!) So even if you want fresh information from bits b0 and b7 you have to write something to the port first.

As all of the printer operations happen very quickly there is no hope of controlling then from ZX BASIC. The following routine, written in a cross between assembly language and English, gives the fundamental method of writing a single row of 256 dots:

```
                LDA 0
                OUT 251,A       start motor at full speed
paper           IN A,251        get the printer status
                RL A            rotate one bit left to
                JP M,noprint    test for printer
                JP NC,paper     test for stylus on paper
encode          IN A,251        now read encoder bit
                RR A
                JP NC,encode    and wait for it to be 1
then either:
                LDA 0
                OUT 251,A       to print a paper dot
or
                LDA 128
                OUT 251,A       to print an ink dot
```

This process is then repeated by jumping back to 'encoder' to print the 256 dots in a line. The only other point to note is that the motor must be slowed for the last two lines that you intend printing.

As already mentioned, the description given above is of little practical use to the Spectrum programmer, as the built-in software provides all the facilities required for using the ZX printer. One project that does spring to mind is the use of the ZX printer with other computers – but that is obviously outside the scope of this particular book!

Chapter Nine

Interface I and the Microdrives

The addition of an Interface 1 and a number of Microdrives turns the Spectrum into a very powerful and versatile computer system. Interface 1 on its own adds the hardware and the software necessary for a standard RS232 serial interface and a local area network. These two features make the Interface 1 important in its own right, and the local area network is sufficiently interesting to merit a book to itself!

The Microdrives add a new capacity for handling data to the Spectrum. Based on a continuous loop of tape, the Microdrives are not as fast as floppy discs nor are they (currently) capable of storing as much data. For simple applications – for example, saving and loading programs – the Microdrives are best thought of as a faster tape system. This speed difference is not in itself sufficient reason for preferring Microdrives: their response is far from instant, and you still have to wait a few seconds while a program loads. The main reason for using Microdrives is that they open up a range of applications that were difficult, if not impossible, for the Spectrum to tackle. For example, it is very difficult to see how even small quantities of data stored on tape can be processed if the results also need to be stored on tape. The unexpanded Spectrum is mainly limited to processing amounts of data that are small enough to be held in RAM. With even one Microdrive it is possible to read from one data file while writing to another. Also the software extension to ZX BASIC in Interface 1 allows the creation of 'real' data files, not just the saving of arrays. In other words, the Microdrive may not be as fast as a floppy disc but it does open up roughly the same range of applications. At a much simpler level, the ability to store a number of programs on a single cartridge is a convenience that justifies the expansion of any Spectrum.

This chapter looks at the extensions to ZX BASIC that accompany Interface 1. The final part of the chapter gives some

short examples of how the new features can be used to create and process data files. The next chapter examines some of the internal workings of Interface 1 and the Microdrives. This is such a large subject that there is only space to give the general principles and important details. However, by this stage you should be able to use the information to good effect to create your own programs.

ZX Microdrive BASIC - file specifiers

Interface 1 contains an additional 8K of ROM that supplements the 16K ZX BASIC ROM found in the standard Spectrum. The way this addition is accomplished is described later. What is of interest at this point is the form and use of these additions.

The additional commands of the extended BASIC, which I shall refer to as 'ZX Microdrive BASIC', or ZXM BASIC, fall into four categories:

1) extended tape commands such as LOAD*, SAVE* etc
2) new Microdrive-only commands such as CAT etc
3) extended channel commands
4) ad hoc commands CLEAR # and CLS #

The form of these commands is much easier to understand and remember once you know that data is stored on a Microdrive in the form of a named 'file', in much the same way that it is on tape. The main difference is that to identify a file on tape all you have to give is its 'filename'; for a Microdrive you have to give a complete 'file specifier'. The format of a file specifier is:

device;device number;filename

where 'device' is a string that identifies the type of device that the file is stored on, 'device number' is a number that identifies exactly which device, and finally 'filename' is a string that gives the name of the file. The filename follows the usual rule of having up to 10 letters, and any of the parameters can be replaced by variables of the correct type. For example, Microdrives are specified by a device code "M" or "m" and so

"m";2;"myfile"

specifies a file called 'myfile' stored on the second Microdrive in the system. Although file specifiers that use constants are by far the most common, it is worth remembering that

D$;D;F$

s a perfectly valid file specifier as long as D$ contains a device type, D a device number and F$ a file name with a maximum of 10 letters. For the moment the only device type that will be used is "m" for the Microdrives, but other device codes, used to refer to the other devices controlled by Interface 1, will be introduced in Chapter 11.

The extensions to the tape commands

Once you know the format of a file specifier the new BASIC commands are very easy to remember. The extensions to the commands that formerly handled only the tape system are

LOAD* file specifier
MERGE* file specifier
SAVE* file specifier
VERIFY*file specifier

These commands carry out the actions that are familiar from tape operation, but using one of the devices controlled by Interface 1. For example

SAVE* "m";1;"myprog"

will save the current program on Microdrive 1 using the filename 'myprog' and

LOAD* "m";1;"myprog"

will restore it. Both VERIFY* and MERGE* work with the Microdrive in the same way as for the tape system. You can also use the other forms of SAVE with SAVE*. The following commands are all valid with the Microdrives and identical in operation to the equivalent tape commands:

SAVE* file specifier LINE number
SAVE* file specifier DATA array name ()
SAVE* file specifier CODE start,length
SAVE* file specifier SCREEN$
LOAD* file specifier DATA array name ()
LOAD* file specifier CODE start, length
LOAD* file specifier SCREEN$

The new Microdrive commands

There are four completely new Microdrive commands:

CAT drive number
This command will produce a list of existing files on the Microdrive indicated by 'drive number', where drive number can be a variable. For example, CAT 1 gives a catalogue of Microdrive 1, and CAT d will give a catalogue of the drive indicated by the value stored in d.

ERASE file specifier
This command will remove, that is erase, the file indicated by 'file specifier'. The storage space that the file occupied on the Microdrive is then reusable. For example, ERASE "m";1;"myprog" will remove the file 'myprog' from the cartridge in Microdrive 1.

FORMAT file specifier
This command erases all of the files on cartridge and prepares it for further use. A brand new cartridge has to be formatted before it can be used. The 'file specifier' used with this command selects the device that will be formatted and the 'file name' within the file specifier is the name given to the whole cartridge. For example, FORMAT "m";1;"data' will format the cartridge in Microdrive 1 and give the whole cartridge the name 'data'.

MOVE file specifier 1 TO file specifier 2
This command will copy the file indicated by 'file specifier 1' to the device and with the file name indicated by 'file specifier 2'. For example, MOVE "m";1;"mydata" TO "m";2;"mydata2" will create a copy of the file 'mydata' on Microdrive 2 and call it 'mydata2'. The MOVE command can be used to make two copies of the same file (using different names) on the same drive or two copies of the same file (perhaps even using the same file name) on different drives. It is important to note that MOVE only works with data files, that is, with files that have not been created using SAVE. To copy program files all you have to do is use LOAD* and SAVE*. There are other, more sophisticated ways of using MOVE but these are better explained later.

The channel and stream commands

difficult to understand or remember; they fit into the overall channel and stream philosophy described in Chapter 5. Indeed, it is only with Interface 1 connected that the channel and stream system of I/O becomes really useful.

The OPEN # command is still used to associate channels with streams, but now the range of channel specifiers is increased to include file specifiers. That is

```
OPEN # s,file specifier
```

associates stream 's' with the channel given by 'file specifier'. For example

```
OPEN # 5, "m";1;"mydata"
```

opens stream 5 to the channel formed by the file 'mydata' on Microdrive 1. Notice that this description extends the idea of a channel as an I/O device to include any separate and identifiable source or sink of data. In this sense, although a Microdrive is a single physical I/O device, the fact that it can hold a number of separate named files, each of which can be a source or sink of data, makes it better to think of it as a number of channels. Once a stream is OPENed to a file the usual INPUT # INKEY$ # and PRINT # commands can be used to read and write data, and the command CLOSE # can be used to break the association. You can even use LIST # to send the listing of a program to a Microdrive file.

An important point about the way any channel works is that the PRINT command sends the same stream of ASCII codes to a channel no matter what it is, and the INPUT statement interprets the ASCII codes that it receives in the same way, no matter what the channel is. This principle is obvious when the channel devices are the familiar keyboard, screen and printer, but not quite so clear when the channel is a file on a Microdrive. For example, if A=1234 then PRINT #5;A will send five ASCII codes (49, 50, 51, 52 and 13, corresponding to the digits 1,2,3,4 and ENTER) to whatever device is associated with stream 5. Even though the sequence of codes sent to the Microdrives is the same as that sent to any other device, there are some ways in which a file channel behaves differently. The best way to illustrate these differences is by an example.

Reading and writing a file — buffering

Consider the problem of writing 20 random numbers out to a file

and then reading them back in. One of the many possible solutions is:

```
 10 OPEN #5,"m";1;"mydata"
 20 FOR I=1 TO 20
 30 LET X=RND
 40 PRINT X
 50 PRINT #5;X
 60 NEXT I
 70 CLOSE #5
 80 OPEN #5,"m";1;"mydata"
 90 FOR I=1 TO 20
100 INPUT #5;R
110 PRINT R
120 NEXT I
130 CLOSE #5
```

If you run this program and watch or listen to the Microdrive you will discover that it runs its tape for a while before the random numbers are printed by line 40. Then, after all 20 numbers are printed on the screen, the motor starts up, the border colours flash, and after a wait the numbers are again printed on the screen. The reason for this sequence of operations lies in the fact that data to and from the Microdrives is *buffered*. Instead of each character being sent to the Microdrive as it is PRINTed, it is collected in an area of memory known as a 'buffer' until there are enough to make it worth starting the Microdrive. This means that data is only written out to the Microdrive when a full buffer of data has been collected. As the buffer holds 512 characters, the program given above finishes PRINTing data without filling a buffer. In this case the CLOSE statement at line 70 now has an additional job. It signals to the Spectrum not only that the association between stream and channel should be broken, but also that a partly filled buffer should be sent to the Microdrives. Without this CLOSE statement the data would stay in the buffer and never be written out. The Microdrive is switched on for the first time because the OPEN command is searching for the existence or otherwise of the file called 'mydata'. When it has scanned the whole tape without success the Microdrive is switched off, and the FOR loop PRINTs the data on the screen, and also sends it to channel 5 where it is collected in a buffer. The Microdrive is switched on again by the CLOSE statement, and the contents of the partially filled buffer are written out. The next OPEN statement again causes the tape to be searched for the file called 'mydata', only this time the search is successful, and a buffer-

load of data is read from it. When the second FOR loop starts to INPUT data the Microdrive is switched off because the data is coming from the buffer. If more than 512 characters of data are read from the file, then the Microdrive starts up again as another buffer of data is read in. To summarise:

1) The OPEN command searches the tape for the file specified. If it is found then a buffer of data is read in ready for the first INPUT command on that stream.

2) Data produced by a PRINT command to the file is collected in a 512-character buffer before being written out

3) The INPUT statement takes data from the buffer unless it is empty, when another buffer of data is read in from the drive

4) The CLOSE command will automatically send the data in any partially filled buffer to the Microdrive.

There is no real need to understand the exact operation of the buffering system that the Spectrum uses, but it does help to explain why the Microdrive switches on at times when you might otherwise not expect it to.

Using PRINT #, INPUT # and INKEY$#

There are one or two other special features of streams associated with file channels. Firstly, you can only send data to a file that did not exist before the OPEN statement, and more obviously you can only read data from a file that exists before the OPEN. A file can only be OPEN for reading or writing, and not for both at the same time. If you want to make sure that a file doesn't exist before you attempt to write to it, then you can attempt to ERASE it first. For example, add

```
5 ERASE "m";1;"mydata"
```

to the program in the last section.

Making sure that you don't write to a file being read is a little more difficult than you might imagine. An INPUT statement such as

```
INPUT #5,A
```

will send the control code for 'move to the next print zone' to stream 5, because of the comma before the A. To avoid sending data to read

files. INPUT statements should use only semi-colons as separators. Similarly, the only separator that should be used in a PRINT statement sending data to a file is the apostrophe. The reason for this is that, as already explained, the sequence of characters sent to a write file by a PRINT statement is exactly the same as that sent to the video driver (see Chapter 6). However, on reading the file back, the INPUT statement accepts characters from the file and treats them as if they had been typed on the keyboard. As you can verify very quickly, the only valid way of ending the keyboard entry of a data item to an INPUT statement is to press ENTER. For example, the only correct way to enter data in response to

```
10 INPUT A;B;C$
```

is to type a valid number, then ENTER, another valid number and ENTER, and finally a valid string of characters followed by ENTER. This rule of ending each data item with ENTER also applies for INPUT from Microdrive files, but using PRINT it is quite possible to create files that contain sequences of characters that cannot be read back! For example, try

```
10 OPEN #5,"m";1;"noread"
20 LET A=RND:LET B=RND
30 PRINT A,B
40 PRINT #5;A,B
50 CLOSE #5
60 OPEN #5,"m";1;"noread"
70 INPUT #5,A;B
80 PRINT A,B
```

The result will be a crash at line 70! The reason is that line 40 writes the two numbers separated by the control code for ',' that is ASCII 6. There is no way that this sequence of characters can be read back by an INPUT statement using numeric variables. However, it can be read back using a string variable

```
70 INPUT #5;A$
80 PRINT A$
```

which reads back the exact sequence of ASCII codes that were sent to the file by the PRINT command, and store them in the string A$. You can also read the file back character by character using INKEY$:

```
70 LET A$=INKEY$ #5
80 PRINT A
90 GOTO 70
```

In general INKEY$ # will return the next character in the file no matter what it is – printable character or control code.

What this means is that if you want to write a data item to a file and read it back as a separate item it has to be followed by an ENTER code. This ENTER code can be generated automatically at the end of the PRINT # statement, or by including apostrophes between data items. For example

```
PRINT #s;A
PRINT #s;B
```

and

```
PRINT #s;A'B
```

both write two separate numeric items to the file associated with stream s.

As a final and rather surprising example of how INPUT from a file is treated exactly like INPUT from the keyboard try:

```
10 OPEN #5,"m";1;"questions"
20 PRINT #5;"2*2"
30 CLOSE #5
40 OPEN #5,"m";1;"questions"
50 INPUT #5;A
60 PRINT A
70 CLOSE #5
```

You might think that as line 20 writes a non-numeric string (2*2) to the file the INPUT statement at line 50 would fail. What actually happens is that the expression is evaluated and the answer 4 is stored in A. This is a reflection of the fact that any numeric expression typed in response to an INPUT will be evaluated and treated as if you had typed the result instead!

The rules to remember are:

1) Each data item that you write to a file and want to read back as a separate item should be followed by an ENTER code

2) A numeric item should be a valid arithmetic expression

3) A string item can include any type of character and its end is marked by an ENTER code

Advanced CAT

The full form of the CAT command is:

CAT #s, drive number

This will send the catalogue output to stream s, so

CAT #3,1

will catalogue drive 1 to the printer, the default channel OPENed to stream 3. One of the main uses of this form of the CAT command is to set up a file on Microdrive that contains all of the information about the files on a cartridge. For example

```
10 ERASE "m";1;"ccat"
20 OPEN #4,"m";1;"ccat"
30 CAT #4,1
40 CLOSE #4
```

will create a data file containing the current catalogue of drive 1. This file can then be read back later in the program to discover if a file already existed, or simply to discover the amount of space left on the cartridge.

Advanced MOVEing - renaming and appending

The MOVE command has an extended form in which either of the file specifiers can be replaced by stream numbers. For example, the command

MOVE "m";1;"ccat" TO #2

will MOVE the data in the file 'ccat' to the screen, the default channel OPENed to stream 2. In this way MOVE can be used to list data files to the screen or the printer. It is possible to MOVE data from the keyboard to a data file, but stopping the data transfer is very messy, and it is better not to use MOVE to 'connect' streams together in unorthodox ways. For example

MOVE #1 TO #3

will move data from the keyboard to the ZX printer, but the only way to break this connection is to switch the machine off!

There is no command that will explicitly allow you to rename an

existing file, but the MOVE command can be used to the same effect. The command

Move filespec1 TO filespec2:ERASE filespec 1

will first make a copy of 'filespec1' under the new name 'filespec2' and then ERASE the old copy, thus effectively renaming the file.

If MOVE is used with file specifiers then it closes the file at the end of the operation, but if stream numbers are used then the stream is left OPEN until it is explicitly CLOSEd. This gives us a way of using the MOVE command to append one data file to another. For example

```
10 OPEN #4,"m";1;"long"
20 MOVE "m";1;"first" TO #4
30 MOVE "m";1;"second" TO #4
40 CLOSE #4
```

will append the file 'second' to the file 'first', the result being stored in a file called 'long'. As the MOVE command doesn't close a stream at the end of its operation it can transfer the whole of a file to any position within another. You could add data to a file by MOVEing it to a new file, PRINTing the new data, and then using MOVE and ERASE to give the new file the name of the original file.

CLEAR # and CLS

The two commands CLEAR # and CLS # seem to have been added to ZX BASIC to improve it rather than because they were necessary. CLEAR # will reset the streams and channels to their initial state following switch-on. In effect this CLOSEs all the streams and resets streams 0 to 3 to their default channels. However, it is important to realise that CLEAR # is not a substitute for CLOSEing any files that might be open. The difference is that following CLEAR # any data that is stored in partially filled buffers is discarded without being written out to a Microdrive! (Remember that a CLOSE will write any partially filled buffers to the appropriate file before breaking the stream/channel association.)

The command CLS # not only clears the screen in the same way as CLS, it also resets all of the screen attributes to their initial values at switch-on, i.e INK to black, PAPER to white and so on. It is a good idea to start all programs that are intended for use with Interface 1 only with

```
10 CLS #:CLEAR #
```

This will ensure that all attributes are reset and all streams, apart from the default ones 0 to 3, are CLOSEd.

The end-of-file problem

Something that has been ignored so far is the problem of knowing when a program has reached the last item while reading a file. This is important because the Spectrum will give you an error message and crash if it tries to read an item after the end of the file has been reached. Unlike other versions of BASIC, there is no built-in function to detect the end of a file in ZX BASIC, so we must either use a specially-written machine code routine or place a special marker at the end of a file. Using a special marker is quite easy in ZX BASIC, as there are a number of character codes that never occur in normal use. For example, CHR$(0) to CHR$(5) are assigned no meaning by ZX BASIC, so they can be used to mark any special points in a file. This use of markers or flags is easy enough if the data is stored on the file in the form of strings, but it is obviously not possible to include such odd characters with numeric data.

The solution to this problem is always to read numeric data into a string, and then test this string for the end-of-file flag. If the string isn't the end-of-file flag then presumably it is a valid numeric data item, and can be converted to numeric form by using VAL. For example, if CHR$(0) is being used as the end-of-file marker, the following program will write a file with a random number of data items and then read it back without causing an end-of-file error:

```
 10 OPEN #4,"m";1;"random"
 20 LET L=INT(RND*50)+100
 30 FOR I=1 TO L
 40 PRINT #4;RND
 50 NEXT I
 60 PRINT #4;CHR$(0)
 70 CLOSE #4
 80 OPEN #4,"m";1;"random"
 90 INPUT #4;a$
100 IF A$=CHR$(0) THEN CLOSE #4:STOP
110 LET A=VAL A$
120 PRINT A
130 GOTO 90
```

A prompting ERASE program

One of the most tedious occupations imaginable is trying to ERASE all the redundant files on a cartridge. To avoid repeatedly typing in ERASE etc. the following program will read the catalogue and then ask the user whether or not each file is to be ERASEd. In other words, it provides a *prompting delete* facility.

```
 10 CLEAR #:CLS #
 20 INPUT "Which drive ?";D
 30 PRINT AT 10,8;"Please wait"
 40 ERASE "m";D;"ccat"
 50 OPEN #4,"m";D;"ccat"
 60 CAT #4,D
 70 CLOSE #4
 80 OPEN #4,"m";D;"ccat"
 90 CLS
100 INPUT #4;C$
110 PRINT AT 0,10;C$"
120 INPUT #4;F$
130 INPUT #4;F$
140 IF LEN F$=0 THEN GOTO 220
150 PRINT "Delete ";F$;" y/n ?";
160 INPUT A$
170 IF A$(1)<>"N" AND A$(1)<>"Y" THEN
    GOTO 150
180 PRINT A$
190 IF A$(1)="N" THEN GOTO 130
200 ERASE "m";D;F$
210 GOTO 130
220 PRINT "No more files"
```

The first part of the program (lines 10 to 80) sets up a file 'ccat' on the drive specified, which contains the catalogue of the same drive. Notice the way that d is used to specify the drive number. The second part of the program (lines 90 to 200) then reads the file in to obtain the name of each file in turn, and asks if each should be removed or not. The double INPUT at lines 120 and 130 is not a mistake! The first entry in the file 'ccat' is the cartridge name: this is read by line 100. Then there is a null string, read by line 120, and only then comes the first proper file name read by 130. After this, each read of the file returns either a file name or a null string which marks the end of the list of files. The null string is detected by line 140 and used to end the program.

Data file handling - an example

The previous example illustrated one way in which the standard ZXM BASIC commands can be used to construct useful Microdrive operations. Handling data files appears to be such a simple application of the BASIC commands provided that examples may seem unnecessary. In practice, however, the business of handling data files often proves to be full of subtle traps. The following short example involves the creation and maintenance of the simplest type of data file – a sequential file. The actual application is a personalised telephone directory, but in many ways this is irrelevant to the example. Any sort of data that needed to be stored, added to and then examined would present the same set of programming problems.

The example consists of two small programs. The first is used to add entries to the directory:

```
 10 CLEAR #:CLS #
 20 GOSUB 1000

 30 CLS
 40 PRINT "Enter new names and numbers"
 50 PRINT "type # when all entries added"
 60 INPUT "surname ";S$
 70 IF S$="#" THEN GOTO 180
 80 INPUT "initials";I$
 90 INPUT "telephone number ";T$
100 PRINT AT 5,0;"New Entry-"
110 PRINT AT 10,0;I$;" ";S$;
120 PRINT " Tel ";T$
130 INPUT "Is this correct (y/n)";A$
140 IF A$(1)<>"y" AND A$(1)<>"n" THEN
    GOTO 130
150 IF A$(1)="n" THEN GOTO 30
160 PRINT #4;S$'I$'T$
170 GOTO 30

180 PRINT #4;CHR$ 0'CHR$ 0'CHR$ 0
190 CLOSE #4
200 ERASE "m";1;"telnum"
210 MOVE "m";1;"temp$$$" TO "m";1;"telnum"
220 ERASE "m";1;"temp$$$"
230 STOP
```

```
1000 OPEN #5,"m";1;"telnum"
1010 OPEN #4,"m";1;"temp$$$"
1020 INPUT #5;S$'I$'T$
1030 IF S$=CHR$ 0 THEN RETURN
1040 PRINT #4;S$'I$'T$
1050 GOTO 1020
```

Lines 10 and 20 get things going. Line 20 calls subroutine 1000 which reads the existing file of telephone numbers, 'telnum', and creates a new file called 'temp$$$'. The telephone number file is organised into groups of three string data items. The first records the surname, the second the initials and the third the telephone number. The end of the file is marked by a group of three items each equal to CHR$ 0. Subroutine 1000 copies the file 'telnum' to 'temp$$$' so the new items can be added at the end. You might think that the easiest and quickest way to do this is to use MOVE as explained earlier. However, MOVE would copy the entire 'telnum' file, including the three CHR$# 0 items that mark the end of the file! Obviously if the file is going to be extended the CHR$ 0 items have to be left out of the copy, and this is exactly what line 1030 ensures.

Once the 'temp$$$' file is set up, the main part of the program (lines 30 to 170) allows new names and telephone numbers to be entered to the three string variables S$ (surname), I$ (initials) and T$ (telephone number). If the new entry is correct it is written out to the file by line 160. When all the entries have been written out, line 180 adds the three CHR$ 0 characters to mark the new end of the file. Then lines 200 to 230 rename 'temp$$$' as 'telnum', so restoring the cartridge to its original state.

The first time you run this program to set up a telephone directory it will crash because it tries to read the non-existent file 'telnum'. The solution to this problem is to create a short file directly using

```
OPEN #4,"m";1;"telnum":
PRINT #4;CHR$0'CHR$0'CHR$0:CLOSE#4
```

thus preparing the cartridge for the program.

The second program reads the file of names and telephone numbers and searchers for any given surname:

```
10 OPEN #4,"m";1;"telnum"
20 INPUT "surname";N$
30 INPUT #4;S$;I$;T$
40 IF S$=CHR$ 0 THEN CLOSE #4:GOTO 10
50 IF S$<>N$ THEN GOTO 30
60 PRINT I$;" ";S$;" Tel ";T$
70 GOTO 30
```

This program is surprisingly simple. Line 10 OPENs the file and lines 30 to 70 read it through, searching for the surname in N$. Line 40 detects the end of the file. You might be puzzle by the CLOSE in line 40 being followed so closely by an OPEN in line 10. The reason for this is that each time a name is searched for, the file has to be read from the beginning again, and the OPEN command ensures that this is the case.

There is nothing else to add to the description of this example apart from pointing out that the time it takes to retrieve the telephone number doesn't depend very much on the size of the file. The reason for this is that each time a name is searched for, not only the entire file but the entire tape is read! The OPEN command that is essential to the re-reading of the file scans through the remainder of the tape to get us back to the beginning of the file. The consequences of this method of re-reading a file are discussed further in the next chapter.

Putting the Microdrives to work

This chapter has described the sort of operations that Microdrives are capable of. They open up a whole new world of Spectrum programming, and it is important not to ignore the challenge of producing good and usable applications that take advantage of their facilities. The Microdrive cannot be treated as a traditional data storage device, beause it is really nothing more than a fast tape drive with a well-developed set of extensions to ZX BASIC. With such a device, achieving reasonable response time and user-friendly operation is decidedly possible, although it does require a great deal of understanding of both the working of the Microdrive and the application in hand.

Chapter Ten

Principles of Interface I and the Microdrives

There are two main areas of interest concerning the way that Interface 1 and the Microdrives work. Firstly, there is the interesting question of how Interface 1 can provide 8K of ROM routines to add the new ZX BASIC commands (and extend some of the old ones) when a 48K Spectrum has no spare address space! Secondly, there is the way that the system of channels and streams are extended to accommodate the Microdrives. Both these topics are dealt with in this chapter, and some of the very many applications that they open up are explored.

The ROM paging

It is difficult to extend the machine code routines contained in the 16K ZX BASIC ROM because all the Spectrum's available 64K's worth of addresses are already allocated either to RAM or ROM. Shortage of addresses is becoming a fairly common problem as microcomputers become increasingly sophisticated. The standard solution is to use *paging*. Paging is a technique whereby a block of addresses can be shared by a number of blocks of memory. Of course, at any one time only one of the memory blocks can be addressed, and this implies that to make use of the other blocks there has to be a way of switching one block of memory out and another in. This switching out and in of a memory block is usually referred to as 'paging'. For example, the BBC Micro uses paging to select one of a number of 16K ROMs, each of which might contain a different application or language. Paging is also used by the Spectrum to add Interface 1's extra 8K of ROM. At any one time either the usual 16K ZX BASIC ROM is present, or the new 8K ROM is switched in. The actual electronics of paging is not elaborate because the Spectrum was designed with a ROM disable line that is brought out to the

expansion connector (see Chapter 2). Holding this line at +5 volts will stop the 16K ROM responding to any address, and thus allow another ROM to take its place.

This all sounds very simple, but the Spectrum's use of ROM paging is very different from most in that it extends the existing ZX BASIC commands using the machine code routines in the paged 8K ROM. This implies that the paged ROM has to be switched in and out automatically as a program is running. The question is, how? Consider for a moment what happens when the Spectrum comes across a statement that isn't in its normal repertoire. It immediately signals an error by doing a RST 8 to call the error handler routine in the ZX BASIC ROM. If this jump to memory location 8 is detected, and used to page in the new ROM, then the routines that it contains can check the form of the command and see if it corresponds to something that it can handle. This is in fact what happens. Interface 1 continuously monitors the Spectrum's address bus for the occurrence of address 8, which it immediately takes as its cue to page in the new 8K ROM. Thus the command

```
CAT 1
```

will cause an unexpanded Spectrum to give an error message by doing a RST 8, but in a Spectrum connected to Interface 1 the RST 8 pages in the new 8K ROM, which carries out the catalogue operation and then returns to ZX BASIC after clearing the error flags. Of course, if the command line is not recognised by the new ROM it passes the error back to the ZX BASIC ROM, which then produces an error message.

Interface 1 will also page in the new ROM if address 5896 is used. The reason for this is that 5896 is within the ZX BASIC ROM's CLOSE routine, and this has to be intercepted before it even attempts to CLOSE a Microdrive channel. The 16K ROM is paged back in by the new 8K ROM using address 1792. Methods of paging the ROMs and using the facilities in the new 8K ROM will be discussed later in this chapter.

The Microdrive data format

The Microdrive is essentially a fast tape drive with the tape formed into a continuous loop, so that any part of it can be written or read without rewinding. Data is written and read on two tracks to achieve a reasonable level of data storage. Details of the exact physical

format that is used to store are unlikely to be of use, because the Microdrive is a device unique to the Sinclair range of computers. However, the organisation of data on the tape is of interest. Unlike the standard tape cassette system, data is stored on the Microdrive in blocks of fixed size known as *sectors*. Simply calling a 'sector' a 'block of data' is understating the case a little. A sector is better thought of as an area of tape where data can be stored. When a cartridge is FORMATted, as many sectors as can be accommodated are written to the tape. At first all these sectors are 'marked' as being free for use; when you write data to the drive, sectors are used and marked as used. If it helps you to visualise what is going on, you can think of a free sector as containing arbitrary data that is of no interest, and a used sector as containing data that you value. In actual fact the FORMAT command marks some of the sectors that it creates as used, because they lie on a part of the tape that is faulty – through being near the splice that joins up the loop, or through having a damaged surface for some other reason.

As the sector is the fundamental unit of data storage on the Microdrive, it is obviously worth examining it in more detail.

The sector format

Each sector on the tape is made up of two parts a *header block* and a *data block*. The purpose of the header is to identify the particular sector that is currently passing under the read head. The format of a header block is:

---

12 bytes of lead-in signal
1 byte flag
1 byte sector number
2 bytes unused
10 bytes cartridge name
1 byte check sum

It is important to realise that the sole purpose of the header is to mark the current position on the tape, and in this sense its most important component is the 1-byte sector number. When the FORMAT command creates the sectors, it assigns each one a unique sector number between 0 and 255. However, not all these sector numbers exist on any given cartridge, as the tape is simply not

long enough. Header blocks are read by both read and write operations, but the only operation that writes them is a **FORMAT**. The header blocks form unchanging 'signposts' to the data blocks that follow them.

The format of a data block is best thought of in two parts: a record descriptor that stores information about the data that follows and, at long last, a record which stores useful data. The detailed format of a data block is:

---

record descriptor

12 bytes of lead-in signal
1 byte flag
1 byte record number
2 bytes record length
10 bytes file name
1 byte check sum

record

512 bytes of data
1 byte check sum

Notice that the record descriptor part of the data block has the same format as a header block, and so can be read by the same software. It contains a number of pieces of information that are essential to the organisation of sectors into *named files*.

A named file is a collection of sectors. The file name is stored in each sector in the ten bytes set aside for it in the record descriptor. The order in which the sectors should be taken to make up the file is given by the 1-byte record number. For example, a file might consist of five sectors: the first would be record 0, then record 1 and so on to record 4. Notice that the record number has nothing to do with the sector number that the data happens to be stored in. For example, record 0 might be stored in sector 57, record 1 in sector 66 and so on. The complication in this simple picture is the possibility that a sector's data area may not be completely used. As a file is created, a new sector is written only when a buffer is full, so the only time a partially-filled buffer can be written out is at the end of a file. The two record-length bytes are used to hold the number of bytes of the data area that actually hold data. For all but the last record in a file, the record-length number will be 512 bytes.

Microdrive maps

There is a fundamental problem with the sector format used with the Microdrives. It is not evident until you try to work out how sectors are written during file creation. The problem is, how do you know whether or not the sector that is just about to pass under the read/write head is free or used? The header block is never re-written, so it cannot be used to hold the change in status of a data block that has just been made part of a file, or freed by an ERASE operation. You might think that the obvious place to store the information about whether or not a data block was free was in the data block itself. This is indeed the only place where such varying information can be stored, but using it brings another problem. Due to timing problems, the Microdrive can only rewrite an entire data block. Suppose there is a full buffer ready to be written out to a free sector. The built-in software reads the record descriptors as they pass under the read head to discover if the data block that follows is free. When a free block has been found, it is too late to begin writing the data out: the first part of the block (the record descriptor) has passed the reading head, and there is no way you can write a fragment of a data block. One solution would be to wait until the header of the free block that had been identified came round again! This, of course, would mean that each write operation would involve at least one complete scan through the tape, and overall operation would be very slow. The solution adopted by the Spectrum is to construct a Microdrive map that shows which sectors on a cartridge are free and which are used. A Microdrive map consists of a block of 32 bytes, and each of the theoretically possible 256 sectors is represented by a single bit. If the bit that represents a sector is set to 1, this means either that the sector is used or that it doesn't exist on this particular tape. On the other hand, if the bit that represents a sector is set to 0 then the sector is free and can be used to construct a file. You should be able to see that, given a correct Microdrive map, the Spectrum can tell if a sector is free for use simply by reading the sector number in the header, and be ready to rewrite the entire data block if it is.

The Microdrive map is a very clever way around the problem of knowing when a sector is free. A new map has to be constructed for a drive every time that a file is OPENed, because there is always the possibility that the cartridge has been changed since the last time the map was produced. During file operations the map can be kept up to date by setting the bits that represent any sectors used. Thus the only costs encountered in the use of Microdrive maps are the time needed

for a complete read of the tape with each OPEN command, and the 32 bytes of memory required to hold the map itself.

The Microdrive channel

The final component you need to understand is the Microdrive channel. If you refer back to the description of the standard channels and streams in Chapter 5, you will see that all that is necessary to extend the system to include files held on a Microdrive is the introduction of a new type of channel record or descriptor. In fact it is also necessary to extend the software that handles streams and channels, but the new 8K ROM takes care of this.

A Microdrive file channel descriptor has the following format:

byte	name	use
0	–	0008 error routines address
2		0008 error routines address
4	–	'M' channel identifier
5	–	address of output routine
7	–	address of input routine
9	–	595 length of channel descriptor
11	CHBYTE	next byte in record
13	CHREC	current record number
14	CHNAME	10 byte file name
24	CHFLAG	flag b0=0 open for read
25	CHDRIV	drive number
26	CHMAP	address of microdrive map
28	–	12 bytes of lead-in signal
40	HDFLAG	header flag b0 set to 1
41	HDNUM	sector number
42	–	not used
44	HDNAME	cartridge name
54	HDCHK	header check sum
55	–	12 bytes of lead-in signal
67	RECFLG	record flag b0 set to 0
68	RECNUM	record number
69	RECLEN	number of bytes in record
71	RECNAM	10 byte file name
81	DESCHK	record descriptor check sum

32	CHDATA	512 byte data buffer
594	DCHK	data checksum

There are many interesting features in this channel descriptor. In overall structure it falls into three parts. Bytes 0 to 27 form a collection of general channel information, bytes 28 to 54 form a sector header, and bytes 55 to 594 form a data block. The fact that a channel descriptor contains data formulated as a sector header and a data block is, of course, no accident. When a sector is being written or read, the header is stored in bytes 28 to 54 and the data block in bytes 55 to 594. In this sense the last part of the channel description is a memory image or copy of the sector on tape.

The first part of the channel descriptor has roughly the same format as the channel descriptors introduced in Chapter 5. In fact the channel and stream software still treats the first four memory locations as the addresses of the output and input routines to be used with the channel. As these locations now hold address 8, the error routine, any attempt to use the channel's output or input routine causes the new 8K ROM to be paged in. When this happens, the routines in the shadow ROM then use the four locations following the channel identifier (byte 4) as the address of the output and input routines within the new 8K ROM. (Notice that the first four memory locations can still be used to hold the addresses of output and input routines in the main 16K ROM or anywhere in RAM. This idea is explored in the final chapter.)

Byte 9 holds the length of the entire channel descriptor. This is necessary because the software might have to search though the channel's area of memory, and in the extended system, channel descriptors can have different lengths.

The location of the buffer described in the last chapter can now be seen at the end of the channel descriptor for the file concerned. This buffer is filled or emptied as data is sent or retrieved from the file. Bytes 11 and 12, CHBYTE, are used as a pointer to the next byte to be added or removed. When an attempt is made to add the 513th, byte, the entire data block is written out. If a 513th byte is requested, then the next record in the file will be read into the channel descriptor.

The only other byte worth describing is byte 67, RECFLG. This records the fact that the block passing under the head is a data block (b0=0); it also holds one or two other pieces of information. If b1 is set to 1 then the record that has just been read in is the last record in

the file: in other words b1 is an end of file flag. Bit 2 is set to 1 if the file being read is not a PRINT file i.e. if it has been created by a SAVE* command.

Summary

All the important features of Microdrive operation have now been described, but it may be helpful to give a summary of operations.

(1) Data is stored on the tape in the form of fixed-size blocks called sectors.
(2) Each sector is composed of two main parts: the header, which contains the sector number and is not changed during normal operation, and the data block, which contains the filename, record number and the actual 512 bytes of data that each sector can store.
(3) A Microdrive map is used to discover if a sector is used or free. The map is built up each time a file is OPENed by scanning the entire tape, and then kept up to date as sectors are used.
(4) The Microdrive file channel descriptor contains the same data format as a sector, plus a number of extra pieces of information concerning the construction of the file.

The best way to make sure that this method of operation is understood is via the examples in the following sections.

A record/sector lister

It is quite easy to find out which sectors have been used to store the records of a file. All you have to do is read through the file and PEEK the sector number stored in the file channel descriptor each time a new record is read.

There are two questions that have to be answered before this is possible:

1) Where is the file channel descriptor stored?
2) How can the reading of a new record be forced?

The solution to the first problem can be found in Chapter 5. The address of any channel descriptor can be found by examining the correct entry in the stream table. If the channel descriptor has been

opened to channel S, the start address of the descriptor can be found using the following subroutine:

```
1000 LET A=23574+2*S
1010 LET C=PEEK A +256*PEEK(A+1)
1020 LET D=PEEK 23631+256*PEEK 23632
1030 LET C=C+D-1
1040 RETURN
```

Line 1000 finds the address of the entry in the stream table; this holds the offset, from the start of the channel's area of memory, of the channel associated with stream S. Line 1010 stores this offset in C, line 1020 stores the address of the start of the channel's area in D, and line 1030 finally uses all this information to calculate the address of the first byte of the channel descriptor in C.

The second problem is easily solved. An INPUT # on the stream will cause a new record to be read into the buffer if all the data in the buffer has been processed. Bytes 11 and 12 in the buffer, CHBYTE, act as a pointer to the next byte that will be brought from the buffer to satisfy an INPUT #. If this POKEd with a large value (>512) the software will be tricked into thinking that all the data in the buffer has been used, and a new record will be read. Thus:

```
POKE C+12,5
INPUT #S;A$
```

will always cause a new record to be read in (assuming that C contains the address of the first byte of the channel descriptor.)

Now that these two problems have been solved, the program is easy:

```
10 OPEN #4,"m";1;"big"
20 LET S=4:GOSUB 1000
30 PRINT "record ";PEEK(C+68),
40 PRINT "sector ";PEEK(C+41)
50 POKE C+12,5
60 INPUT #4;A$
70 GOTO 30
```

Line 20 uses subroutine 1000 to store, in C, the address of the channel descriptor that is associated with stream 4. Lines 30 and 40 then PRINT the record and sector number by PEEKing the appropriate bytes in the channel descriptor, and finally line 50 and 60 force a new record to be read in.

If you use this program on a tape that contains only one file you

will discover that the records are not stored on sequential sectors. For example, record 0 might be stored on sector 20, record 1 on sector 22, record 2 on sector 24, and so on. The reason for this is that the sectors pass under the Microdrive head faster than data can be built up in the buffer ready to be written out; the 'missing' sectors are missed opportunities!

Looking at the map

The Microdrive maps are stored within the region of memory starting at 23792 (see the next section). The exact address that any map is stored at can be found from an examination of bytes 26 and 27 (CHMAP) of the channel descriptor. Using this information it is easy to print out the bit pattern so that the positions of free and used sectors can be seen. Try the following program:

```
 10 OPEN #4,"m";1;"b"
 20 LET S=4:GOSUB 1000
 30 LET M=PEEK(C+26)+256*PEEK(C+27)
 40 FOR I=0 TO 31
 50 LET B=PEEK(M+I)
 60 FOR J=1 TO 8
 70 PRINT B=2*INT(B/2);
 80 LET B=INT(B/2)
 90 NEXT J
100 NEXT I
110 STOP
```

Line 30 stores the address of the Microdrive map in M. Lines 40 to 90 then print the 32 bytes as a continuous stream of bits so that you can see which sectors are in use.

If you run this program and file 'b' already exists you will discover that something odd happens to the map. The reason for this is that although every OPEN command constructs a Microdrive map, it is only kept if the file is discovered to be a write file at the end of the scan through the tape.

Ad hoc channels and non-PRINT files

Most ZXM BASIC commands such as MOVE, SAVE etc. need to use a channel descriptor in the course of their operation. Such descriptors are created by the commands and then destroyed at the

end of the operation, and are called *ad hoc channels*. The only difference between a normal channel descriptor, as created by an OPEN command, and an ad hoc channel is that the M channel identifier (at byte 4) is replaced by CHR$(205), i.e. CHR$(CODE ("M")+128).

Another feature of Microdrive files is their segregation into *PRINT* and *non-PRINT* files. PRINT files are created by the commands OPEN, PRINT and CLOSE; non-PRINT files are created by SAVE. The only real difference between PRINT and non-PRINT files is that the non-PRINT files store certain items of information about their nature in record 0 of the file. To be precise:

record 0 of file	
byte 1	flag byte # 0= BASIC 1/2= array data 3= code
bytes 2 and 3	number of data bytes in file
bytes 4 and 5	start address
bytes 6 and 7	length of program area
bytes 8 and 9	auto start line number

You should be able to see the similarity between this data and the format of the tape header described in Chapter 8. Even taking this difference into account, there is no reason why a non-PRINT file couldn't be read, using INKEY$, as a sequence of ASCII characters. But the Spectrum's software makes a clear distinction between PRINT and non-PRINT files, and will not allow you to OPEN a non-PRINT file. This is a pity, because it would add yet another dimension to Microdrive data handling if it were possible to read and write program files. It *is* possible to fool the system into writing a non-PRINT file using PRINT statements by POKEing byte 67 (RECFLG) in the channel descriptor with 4 immediately after OPENing the file. The value of RECFLG is the only way the system has to recognise a non-PRINT file. If you do this, you must be sure to write the information described above into the first record before trying to write the program or other data.

The new system variables

When the new 8K ROM is paged in for the first time it creates 58

extra system variables that are necessary to its operation. These are added to the end of the usual system variables area, and take up part of the memory area set aside for Microdrive maps in the unexpanded Spectrum. Rather than give a complete list of all the new variables (one can be found in Appendix 2 of the Interface 1 and Microdrive manual) it makes more sense to describe the few that are useful. Some of the system variables are concerned with the other features provided by Interface 1, and these will be treated in the following chapters. Some are used as temporary work areas for the extended commands and the machine code routines in the new 8K ROM. These are described as required in the section on using machine code. After taking all these out of consideration there are only two interesting new system variables that have anything to do with the Microdrives!

IOBORD (23750)
This simply sets the border colour that the screen flashes during any Interface 1 controlled I/O. You can POKE any colour code that you like into this variable to change and even remove the border flashing.

COPIES (23791)
The value stored in this system variable sets the number of copies of a file that are generated by the SAVE* command. If you make more than one copy of the file it will take more space, but it can increase the speed of loading. Each copy that is made has to be ERASEd separately, i.e. three copies of a program will take three ERASE commands before the program is lost forever!

Using assembly language

Using the routines in the new 8K ROM from Z80 assembly language could be very awkward if it were not for the provision of a special calling mechanism. The 8K ROM is paged in by the ZX BASIC ROM when the error handler is called by a RST 8 instruction. The error code is stored in the memory location following the RST 8 instruction and this is examined by the 8K ROM to find out the type of error that has caused it to be paged in. However, the error codes only use a limited range; and this fact has been used to allow assembly language programs to call Microdrive routines that are stored in the new 8K ROM.

The program

```
RST 8
DEFB code
```

will call a particular Microdrive routine in the new 8K ROM depending on the value of 'code' as given in the following list:

code	action
33	Switch a Microdrive motor on (The A register contains the drive number; if the A register contains zero than all motors are switched off)
34	OPEN an ad hoc channel
35	CLOSE a file
36	ERASE a file
37	Read next record of a PRINT file
38	Write next record of a PRINT file
39	Read a given record of a PRINT file (The record is specified in the channel descriptor)
40	Read a PRINT file sector (The sector specified in CHREC is read)
41	Read next sector on tape (The next sector that passes under the read head is loaded into the channel descriptor)
42	Write a sector (The sector number is stored in CHREC in the channel descriptor)

The routines corresponding to codes 34 and 36 – OPEN and ERASE – use the new system variables D_STR1 (23766) and N_STR1 to hold the drive number and the file name respectively. The first two locations of N_STR1(23770) hold the length of the file name, and the last two (23772) hold the address of the first letter of the file name. On exit from the OPEN operation, the address of the channel descriptor is in the IX register and its displacement (as stored in the stream table) in the DE register. The rest of the routines all expect the address of the channel descriptor to be held in the IX register.

When using any of these routines, it is worth being aware that none of the registers is saved, and that the state of the maskable interrupt is, in general, not predictable. Before using any of the routines, it is good practice to save the HL register pair and disable the interrupts. On return, restore both the HL register pair and enable the interrupts.

A rewind command

As an example of using the new 8K ROM routines, consider the problem of producing a 'rewind' operation. In this context, rewinding a file refers to making the current record into record 0. Using the read record operation (code 39) it is possible to read any record of a file into the channel descriptor's buffer. The following assembly language routine makes this operation available as a USR function:

address	assembly language	code	comment
23296	PUSH HL	229	save HL
23297	LD IX,chan	221,33,0,0	load IX with channel address
23301	DI	243	disable interrupt
23302	RST 8	207	read record
23303	39	39	code
23304	FOR A	175	clear A
23305	RST 8	207	motor off
23306	33	33	code
23307	EI	251	enable interrupts
23308	POP HL	225	restore HL
23309	RET	201	return to BASIC

To use this routine, the address of the channel descriptor has to be POKEd into 23299 and 23300. The only other point of interest is the use of code 33 to stop the motor after the read.

This routine to read any record can be used to advantage in the telephone number program given at the end of the previous chapter. The method used there to re-read the file was to CLOSE the stream and re-OPEN the channel. Of course OPENing the channel means that all the sectors on the tape have to be read to build up a Microdrive map. Time can be saved by avoiding this OPEN command using the 'read any record' routine to read record 0, i.e. rewind the file. Using this idea gives the following modifications to the program

```
  5 GOSUB 1000

 40 IF S$=CHR 0 THEN GOSUB 2000:GOTO 20
```

```
1000 DATA 229,221,33,0,0,243,207,39
     175,207,33,251,255,201
1010 FOR A=23296 TO 23309
1020 READ D
1030 POKE A,D
1040 NEXT A
1050 RETURN

2000 LET S=4
2010 LET A=23574+2*S
2020 LET C=PEEK A+256*PEEK(A+1)
2030 LET D=PEEK 23631+256*PEEK 23632
2040 LET C=C+D-1
2050 POKE 23300,INT(C/256)
2060 POKE 23299,C-INT(C/256)*256
2070 POKE (C+13),0
2080 LET A=USR 23296
2090 POKE (C+11),0
2100 RETURN
```

Subroutine 1000 is the familiar machine code loader used in other examples. Subroutine 2000 performs the rewind operation. Lines 2000 to 2040 get the address of the channel descriptor in C using methods already described. Lines 2050 and 2060 POKE this address into the 'read any record' routine. Line 2070 sets the record number to zero so that line 2080 will load this sector into the record descriptor. Line 2090 resets CHBYTE so that the first byte in the buffer is returned in response to the next INPUT command.

You should find an improvement in the running time of the program because there is no longer the need to read the entire tape just to build up a map.

Random access files

The 'read any record' routine could be used to read the record of a file in any order. This is all you need to implement random access files. However, as already mentioned, the Microdrive is essentially a serial device. If you read record 3 and then want to read record 2 the tape will not 'back space'. Instead the entire tape will run past the read head until record 2 comes round again. The worst possible case is found when reading a file backwards, when each record requires the entire tape to be read before it is found! Because the time to read the entire tape is reasonably low (one or two seconds) there is less

reason to consider random access techniques with the Microdrives. The best processing times are achieved by reading records in their normal sequence and processing as little data as possible. For example, one possible organisation for a telephone number directory is random access, with say, one record of a file assigned to each letter of the alphabet. Using this organisation a telephone number would be found by reading the record that held the entries for all names that began with the same first letter. On average, half the tape would have to be read to find the required record using the 'read any record' routine. If the file were read sequentially, record by record, to get to the desired position then the average amount of tape read is roughly the same. However, if every item of data in each sector was processed in some way, then the time needed to read the file sequentially would be very much greater. Using a sequential read coupled with a forced skip of any irrelevant sectors (see the sector/record list example earlier in this chapter) is likely to be as fast as any random access method.

The continuing saga of Interface 1

The principles of the paged ROM and the Microdrive have been described in this chapter, but this is not the end of the Interface 1 story. There are still two extra types of channel, more 8K ROM routines and a way of customising ZX BASIC to be discussed in the two final chapters.

Chapter Eleven

Interface I and Communication

This chapter looks at the two other features introduced by Interface I – the *RS232 port* and the *network*. Both these facilities are in fact different forms of serial communication. The RS232 port can be used to connect a standard printer to the Spectrum, or for communication with other computers. The network is mainly intended as a way of establishing communications between a number of Spectrums, but it is possible to write software to extend the network to include other computers. The first part of the chapter deals with the RS232 port, and some of the pitfalls of using it, and the second part describes the Spectrum Network. Much of the discussion of how things work builds on the ideas introduced in the last chapter.

RS232 - almost a standard

There are a number of accepted ways of passing data between computers. Some of them even have specified standards, but very few of them are standard in practice. It is rare to be able to connect two computers, or a computer and a peripheral such as a printer, together in such a way that the connection will work immediately. Normally it takes only a few minutes to work out the trouble and put things right. Sometimes it can take a lot longer; you may even need a soldering iron. In the worst cases it can prove impossible to make the connection, but this is very rare.

RS232 is a standard for serial interfaces that specifies a great many things in great detail. The main reason for the incompatibilities between different RS232 interfaces is the choice of the parts of the standard that have been implemented. For example, the simplest RS232 interface consists of three wires – one for a signal from the computer, one for a signal to the computer and one for an earth

connection. The Spectrum uses two more wires than this, one to signal that it is ready to receive data, and one to carry a signal that the device it is connected to is ready to receive data. These two connections are often included in an RS232 interface and are called *handshake lines*. However, other connections are often included to signal, for instance, that the device at the other end of the cable is switched on or off. In the same way, many RS232 interfaces will provide only one of the two handshake lines that the Spectrum uses. All this variation can cause a great deal of difficulty in knowing what to connect to what.

Of course, the trouble with offering advice about how to connect something to the Spectrum is that any problems that might arise depend as much on the 'something' as on the Spectrum. This means that the only real way to tackle the problems that occur in using the RS232 is to understand what is essential to make it work.

The Spectrum's RS232 interface

The pin connections to the 9-way socket that carries the RS232 signals at the rear of Interface 1 are:

Pin no.	use
1	not connected
2	TX - input data to Spectrum
3	RX - output data from Spectrum
4	DTR - 'ready' signal to Spectrum
5	CTS - 'ready' signal from Spectrum
6	not connected
7	ground
8	not connected
9	+9 Volts

If you examine this list you will see that TX (the input data line) pairs with CTS (the 'ready' output line), and RX (the output data line) pairs with DTR (the 'ready' input line to the Spectrum). When the Spectrum is receiving data, the CTS line is used to indicate that it is ready to receive data. The device the Spectrum is connected to must not send data to the Spectrum while CTS is low (i.e. at 0 Volts). If data is sent while CTS is low, it will be ignored or read incorrectly. In

the same way the Spectrum will not transmit data when DTR is held low by the device that is receiving the data.

To summarise:

(1) The Spectrum will only receive data correctly when CTS (pin 5) is is high and this signal should be used to enable the device transmitting data.

(2) The Spectrum will only transmit data when DTR (pin 4) is high and this signal should be used by the receiving device to indicate that it is ready for data.

Handshaking and no handshaking

The conditions for transmitting and receiving data given in the last section are easy enough to understand, but there is one complication that arises even when you try to connect two identical machines together. This can best be described by saying that one machine's output is another's input. For example, if you want to connect two Spectrums together you have to connect pin 3 on the first to pin 2 on the second. That is, the output data signal has to go to the input data pin on the other Spectrum. This seems clear enough in the case of the data lines – i.e. pin 2 connects to pin 3 and pin 3 connects to pin 2 – but this 'crossover' also applies to the handshaking lines. In other words, the DTR and CTS lines on the first Spectrum should be connected to the CTS and DTR lines on the second Spectrum.

This crossing over of connections is the exception rather than the rule. Most peripherals, printers, VDUs etc. are already wired up to take the crossover into account. On a printer you might find that pin 3 is still called 'RX data', but now it is a data input to the printer. In this case it is clear that the Spectrum's pin 3 should be connected to the printer's pin 3. If you buy the RS232 connection cable from Sinclair you will find that the scheme used is:

Spectrum	other equipment
TX data	pin 2 (TX data)
RX data	pin 3 (RX data)
CTS	pin 5
+9V	pin 6 (DSR)
ground	pin 7 ground
DTR	pin 20 DTR

which will work with most printers and VDUs.

In general, if you are making up a cable to allow the Spectrum to work with another piece of equipment, you need to have details of how the other's RS232 interface is arranged. Assuming it uses a 25-pin D plug or socket, first discover whether pin 2 or pin 3 on the device is its output (TX) pin, and connect this to pin 2 on the Spectrum. The other pin should be connected to the Spectrum's pin 3. The next problem is to decide where the handshake lines have to go. Normally DTR from the Spectrum should be connected to either DTR (pin 20), DSR (pin 6) or RTS (pin 4). The Spectrum's CTS signal should be connected to CTS (pin 5) on the other equipment.

On some equipment there are no handshake lines at all. As already mentioned, the simplest RS232 interface uses only the RX, TX and ground connections. In this case the Spectrum's CTS (pin 5) line can be left unconnected, and DTR (pin 4) should be connected to +9V (pin 9). Not using CTS will mean that the device that the Spectrum is connected to will transmit data whenever it likes, even if the Spectrum is not ready to receive. The reason for connecting DTR (pin 4) to +9V (pin 9) is to allow the Spectrum to transmit data whenever it wants to. Of course, for this to work the receiving device must be capable of receiving data at any time! In practice, intermediate situations are often encountered, with RS232 interfaces having only, say, a CTS line. In this case you can only connect up such handshaking lines as there are, and connect the remaining input lines either to ground or +9V depending on whether they have to be held low or high to enable data transmission or reception.

You will appreciate from the above discussion that many different types of problem can be encountered with RS232 connections. In practice, things are not quite as bad as you might expect, and as long as you identify the purpose of each connection on the other device you should have no trouble. It is sometimes useful to connect only the RX, TX and ground lines between the Spectrum and the other device, and connect the handshake line inputs to either ground or +9V to get the interface working without handshake lines. You can then refine the interface connecting the handshake lines one by one.

RS232 data format

The RS232 interface is a serial connection. That is, when data is passed from a computer to another device it is transmitted bit by bit. Although transmission is one bit at a time, it is standard to send a

group of bits in sequence to represent a character. There are a number of choices that can be made about the way the bits are transmitted. You can select one of a number of transmitting speeds or *baud rates*, i.e. the number of bits you transmit per second. You can also select how many bits you are going to send to represent a single character, how many bits to signal the end of transmission of a character, and whether or not you are going to send a *parity bit* to check for transmission errors. In the case of the Spectrum the transmission format used is

8 data bits
no parity check bit
2 stop bits

and the baud rate can be set by the user. Thus as well as making the correct electrical connection between the Spectrum and the other device, it is also important that it is set to receive data in the Spectrum's format. In most cases this only involves making sure that both the Spectrum and the other device are using the same baud rate.

The BASIC RS232 commands

ZX BASIC treats the RS232 interface as just another type of channel or, to be more precise, as two new types of channel. The channel identifiers that are used are:

b or B for binary RS232 channel and
t or T for text RS232 channel

Either channel can be OPENed to a stream in the usual way. For example,

```
OPEN #4, "b"
```

will OPEN stream 4 to the binary RS232 channel and

```
OPEN #5, "t"
```

will OPEN stream 5 to the text RS232 channel. Once a stream has been associated with a channel the usual stream I/O commands – PRINT #, INPUT # and INKEY$ # can be used to send and receive data.

Both the b and t channels behave in the same way when the data that is being transmitted or received is composed of nothing but

printable characters. The difference comes from the way they treat the Spectrum's control codes, and other non-standard assignments of character codes. The b channel will transmit the full 8-bit character code of anything that is PRINTed to it, but the t channel will only send the code if it is a printable character, or it can convert the item into a sequence of printable characters. That is,

PRINT #4; THEN

where THEN is entered as a single keystroke, sends CODE(THEN) or 203 over the b channel. However, if stream 4 was OPEN to the t channel the ASCII codes for the letters T, H, E, N would be sent instead.

The exact rules are:

For transmitting:

The b channel transmits the 8 bit character code of everything it is asked to PRINT

The t channel will not send control codes 0 to 31 or the graphics characters 128 to 164, and will expand all the keywords 165 to 255 to their corresponding strings of ASCII characters

For receiving:

The b channel receives the full 8 bit code that is sent to it

The t channel will ignore the 8th bit of any code it receives, thus restricting it to the standard 0 to 128 ASCII character set

It should be clear that the t channel is an attempt to reconcile the Spectrum's extensions to the standard ASCII character set. For example, if the RS232 interface is connected to a printer,

OPEN #4, "b" :LIST #4

will list the current program; but as the keywords will be sent as single character code, they will either not be printed or cause the printer to do something strange. But

OPEN #4, "t" :LIST #4.

will give a perfectly readable listing, as the codes for the keywords will be expanded to the sequence of characters that normally represents them.

As the b channel works with the full range of character codes, it can be used to transmit the contents of memory locations. To make

this easier, the SAVE*, LOAD*, VERIFY* and MERGE* can all refer to the b channel. For example, SAVE* "b" and LOAD* "b" are both permissible. Of course, without special software these commands only permit the exchange of programs between two Spectrums. (The network provides an easier method of program exchange between Spectrums, but the RS232 interface does have the advantage that it can be used to transfer programs over a telephone line with the aid of a modem.)

There is one other difference between the b and t channels that is important, and that is the treatment of the ENTER character code. As always, the b channel passes every code unaltered, but the t channel replaces each ENTER (code 13) by the two-character sequence 13,10 which is ENTER followed by line feed. Some printers and VDUs will automatically start a new line when they receive ENTER; others also need the line feed code. If the printer doesn't need the line feed code it will throw a blank line between each line of text. There is nothing that can be done about this problem apart from stopping the printer from starting a new line when it receives the ENTER code. (That is, if possible, turn off the auto line feed facility.)

Notice that the t and b channels can also be used with the MOVE command. For example, to send the data from the RS232 interface to the screen use;

```
MOVE "b" TO #2
```

Setting the baud rate

Even though the commands for using the RS232 channels have been given, they remain unusable because the method of setting the baud rate hasn't yet been described. When the Spectrum is first switched on, the baud rate is initialised to 9600. To change it to another value use

```
FORMAT "b";baud
```

or

```
FORMAT "t";baud
```

where 'baud' is one of 50, 110, 300, 600, 1200, 2400, 4800, 9600 and 19200. These are the standard baud rates found on most computer equipment. In the case of the Spectrum, the baud rate can be roughly

interpreted as ten times the number of characters transmitted per second. So 300 baud is about 30 characters a second. Handshaking will sometimes stop the transmission of data, so the actual rate may be less. In most cases it is advisable to use the highest baud rate that the two pieces of equipment can both be set to. A high baud rate means less time waiting for data to be transferred. However, if for any reason you are not using the handshaking lines then the Spectrum will not receive data accurately at baud rates above 300. In fact there is no guarantee that it will receive every character at 300 baud, but without handshaking the slower the better!

To set a non-standard baud rate you can POKE an appropriate constant into the new two-byte system variable BAUD. The constant is given by:

(3500000/(26*baudrate))−2.

Using both t and b

There is nothing against using both the t and b channels at the same time. For example, many printers use ASCII codes in the region 0 to 31 as control codes to produce special printing effects, such as enlarged characters or graphics characters. Apart from these codes a printer is best handled via the t channel.

```
10 OPEN #4,"t"
20 OPEN #5,"b"
30 PRINT #5;CHR$(c);#4;A$
```

c is a control code to be sent to the printer unchanged, and A$ contains text.

Principles of RS232 operation

The RS232 interface is handled by the usual system of I/O channels and streams described in Chapter 5. The only new feature is the addition of another type of channel descriptor:

Byte	use
0	address 8 (the error handler)
2	address 8 (the error handler)

4	t or b channel identifier
5	address of output routine
7	address of input routine
9	11 (length of channel descriptor)

This channel descriptor is the shortest and simplest of the newly introduced channel descriptors. For this reason it is the one most often copied when introducing user-defined channels, as in the example in Chapter 12. Notice that there is no data buffer, so RS232 transmission and reception occurs without delay, unlike Microdrive operations.

The Spectrum RS232 interface is interesting because it is mostly software. Most other machines use special chips that will accept data bytes and transmit them over an RS232 interface without any help from the CPU. The Spectrum contains no such special chips; instead, software creates the necessary signal pulses on the RS232 line in the same way that the sound or cassette pulses are created (see Chapter 8). In this case the I/O port involved is 247, and the RS232 output line RX is controlled by the state of b0. Reading port 247 returns the state of the data input line TX, also as b0. The two handshaking lines are associated with I/O port 239. Reading the port returns DTR as b3, and the state of the CTS line is set by b4.

The sequence of operations involved in sending a byte of data is:

Wait until the DTR line is high

Send the byte bit by bit using the value stored in BAUD to time the length of each pulse.

The sequence of operations involved in receiving a byte of data is a little more complicated:

Examine the value in the new system variable SER_FLG at 23751.

If the value in 23751 is non-zero, then the next memory location 23752 contains the character required. After setting 23751 to zero this is returned as the input.

If the value in SER_FLG is zero, wait until CTS is high, then wait for the start of transmission, signalled by the TX data line going high for one pulse time. Following this the eight data bits are read in and the two stop bits are monitored. Then set the CTS line low to stop anything else being transmitted.

If this fails to stop the transmitting device in time, another complete character will be read in and stored in SER_FLG+1. SER_FLG will be set to 1 to indicate that there is already a data byte waiting. The first byte read in is returned as the input.

The description of the RS232 INPUT operations reveals that RS232 INPUT is in fact buffered, but only to the extent of one character. This is necessary because some sending devices do not respond quickly enough to the lowering of the CTS line to ensure that a second character will not be sent.

Assembler and the RS232 interface

The routines within the new 8K ROM that send and receive data using the RS232 interface can be used in assembler. The method is basically the same as that used in calling the Microdrive routine (see the previous chapter). The routines are called using

```
RST 8
code
```

where the action produced depends on the value of 'code'.

code	action
29	Read a byte from the RS232 interface. The carry flag is set if a byte is read before time out. The result is returned in the A register.
30	Write the byte in the A register to the RS232 interface.

Three general purpose I/O routines are also useful when writing RS232 programs.

code	action
27	Read keyboard. Wait until key is pressed and return its code in the A register
28	Write character in A register to screen without counting scrolls.

31	Write character in A register to the ZX printer.
32	Test the keyboard. Return with the carry flag set if there is a key pressed.

An example of the use of these routines can be found in the next section.

A Spectrum VDU

The main problem with using the Spectrum's RS232 interface for anything other than driving a printer is timing. As all the signals are controlled by software, the handshake lines are absolutely essential for reliable operation. Unlike other machines, there really are times when the Spectrum is not capable of receiving or transmitting a character.

Consider, for example, the problem of turning the Spectrum into a VDU. Logically, the problem is quite simple. Any character received over the RS232 interface should be PRINTed on the screen, and any character typed on the keyboard should be transmitted on the RS232 interface. In ZX BASIC this gives:

```
10 FORMAT "t";baud
20 OPEN #4,"t"
30 LET A$=INKEY$ #4
40 IF A$="" THEN GOTO 60
50 PRINT A$;
60 LET A$=INKEY$
70 IF A$="" THEN GOTO 30
80 PRINT #4;A$;
90 GOTO 30
```

This program works quite well as long as the handshake lines are in use, but even then it is a little sluggish. The same idea can be implemented in assembler as:

address		assembly language	code	comments
23296	LOOP	RST 8	207	RS232 Input
23297		29	29	
23298		JR NC,SKIP	48,2	no input

```
23300        RST 8          207      screen output
23301        28             28
23302 SKIP   RST 8          207      test keyboard
23303        32             32
23304        JR NC,LOOP     48,246   no key pressed
23306        RST 8          207      read key
23307        27             27
23308        RST            207      RS232 output
23309        30             30
23310 NKEY   RST 8          207      test for key released
23311        32             32
23312        JR C,NKEY      56,252
23314        JR LOOP        24,236
```

All that is needed to use this is a BASIC loader:

```
10 FORMAT "b";1200
20 DATA 207,29,48,2,207,28,207,32,
   48,246,207,27,207,30,207,32,56,252,
   24,236
30 FOR A=23296 TO 23315
40 READ D
50 POKE A,D
60 NEXT A
70 LET A=USR 23296
```

The performance of this program still leaves something to be desired. The handling of the Spectrum's keyboard using the keyboard test and keyboard read routines works, but it disables the auto-repeat, and it is still possible to 'get stuck' in the read key routine. The solution is to write a complete custom 'keyboard test and read routine' that mimics the behaviour of INKEY$. A much more serious problem is the way the SPACE key is treated as BREAK by the RS232 routines. Normally you have to press CAPS SHIFT and BREAK to stop a program, but during RS232 operations just the BREAK/SPACE key will do. This makes it virtually impossible to write any serious communication program, such as the VDU program given above, unless you can find a way to generate the character code for SPACE without pressing the SPACE key!

The Spectrum's RS232 interface is excellent for driving printers and transferring programs between the Spectrum and other computers, but other applications require a considerable amount of

software development. The problem of SPACE acting as BREAK may be cleared up in later issues of the new 8K ROM I cannot imagine that this is a feature rather than a bug!

The Sinclair Network

Whereas the RS232 interface is intended to provide communication between two devices, the *network* is intended to allow data transfer between any number of Spectrums. The method of communication used by the network is the same serial format used by the RS232 interface, but there are a number of additions to make n-way communication possible. The hardware characteristics are altered to allow two-way communication over a single pair of wires. At any one moment only one of the Spectrums connected will be transmitting data, and a number of the others will be receiving, but the role of the data transmitter can be adopted by any machine.

The software has been extended to include two extra facilities. Firstly, there is a way for any Spectrum to 'claim' the network and become the transmitter. Secondly, with each 'chunk' of data transmitted there is an address that identifies which of the other machines the data is intended for. These two software features form a sort of 'rule book' for Spectrums trying to use the net – in the jargon they form the *communications protocol*. The Sinclair network's communications protocol is not as sophisticated as that used by other nets, such as Ethernet, but it is suitable for many 'group' computer applications such as education and program development.

The BASIC net commands

The network extensions to ZX BASIC follow the usual lines of the channel and stream extensions to allow for the Microdrives and the RS232 interface. Of the two, the RS232 commands are closer to the network commands. The network channel is identified by "N" or "n" and also has to identify the station that communication is to be with. To make this identification possible, each Spectrum connected to the net has to be assigned a 'station number' using

```
FORMAT "n"; statnum
```

where 'statnum' is a number between 1 and 63. When any Spectrum is first switched on it is initialised to be station 1, so it is important

that everyone using the net has agreed to use a unique station number and uses FORMAT to enforce it. In fact 'statnum' can be 0, but this has a very special use that will be described later. The station number is stored in the new system variable NTSTAT (23749), so FORMAT "n";statnum is equivalent to POKE 23749, statnum. To find the station number in use:

```
PRINT PEEK(23749)
```

To OPEN a channel to send or receive data to station 'num' use:

```
OPEN #s,"n";num
```

Following this OPEN, the command PRINT # can be used to send data and INPUT # and INKEY$ # can be used to receive data. Notice that the OPEN command must be thought of as creating a communications link between the Spectrum that uses the command and the station that is referred to in the command.

There is one complication in using the net to send and receive data: the other station must be aware it is being involved in a communications link with another machine. If you OPEN a net channel to, say, station 13 and station 13 isn't interested, doesn't exist or is doing something else, your Spectrum will wait, possibly for ever (or until the BREAK key is pressed), trying to receive data or transmit data to the missing station. In other words, data transmission on the net uses full handshaking to make sure that when data is sent it is successfully received. This need for one station to know what another station is doing suggests that Spectrum networks are best confined to a single room! However, it is possible to imagine additional machine code software that would add message switching and other sophisticated facilities found on other networks.

The exception to the full handshaking protocol is the INKEY$ # command. This will return a null string if there is no data being transmitted from the station to which the stream refers. This could be used to scan through all the stations on the net to see if any of them are waiting, trying to send data to your station. Otherwise INKEY$ # works in the usual way and returns the next single character sent.

As well as handshaking, another important characteristic of the network channels is that they are all buffered. As with the Microdrive channel, this buffer is part of the channel descriptor (see later) but it is only 256 bytes. The buffering action has the same sort of effect on the network channels that it does on the Microdrive

channels. That is, you can write 256 characters to a network channel before anything is sent to the receiving station, and you can read 256 bytes before the transmitting station has to transmit another buffer of data. Also, partially filled buffers are only sent as the result of a CLOSE# command.

In addition to the channel and stream commands, the network can also be used to exchange ZX BASIC programs. The command

```
SAVE* "n";num
```

will send a program to station 'num' which in turn would use the command

```
LOAD* "n";org
```

to receive it, where 'org' is the number of the station sending the program. Once again, the communication is with full handshaking, and both the receiving and transmitting stations will wait until their counterpart is ready. You can also use MERGE* and VERIFY* in the same way.

Station 0 and broadcasting

The network commands described so far enable data and programs to be exchanged between any two stations. However, it is often necessary to transfer the same program from one Spectrum to a number of others. This can be achieved using station number 0, the *broadcast station*. Data transmitted to station 0 will be transmitted at once, without handshaking, and any number of stations may receive it. For example, to broadcast a program all the receiving stations should first enter

```
LOAD* "n";0
```

They will then wait for the transmitting station to enter

```
SAVE* "n";0
```

and send the program currently in its memory. Notice that it is important for all the receiving stations to have entered LOAD* before the transmitting station sends the program.

Principles of operation

The network uses a two-wire connection: a signal line carrying

pulses varying between 0 and 5V, and a ground return line. The most difficult part of the network's operation is making sure that only one Spectrum is transmitting data at any one time. If two machines do transmit at the same time the high state (5V) has precedence over the low (0V) state. In other words, if one machine is trying to drive the net high (i.e. 5V) and one is trying to drive it low (i.e. 0V) the net will adopt the high state. However, this condition, known as *net contention*, has to be avoided by the use of the net protocol. Before a machine transmits data it has to gain control of the net and so stop any other machine using it.

When a station wants to send data it first monitors the state of the net long enough to detect data pulses if the net is currently in the middle of transmitting a block of data. If no pulses are detected, the station transmits a single byte containing the station number. As it transmits each pulse, it monitors the net to make sure the pulses are as it intended. If it finds a discrepancy – if, for instance, it has sent a low pulse and the net is in a high state – this can only mean that another station is trying to gain control of the net at the same time. When this happens the station that detects the error stops transmitting and starts the process of trying to claim the net all over again.

Once the net is claimed, a header is sent which contains the number of the station the data is intended for and the number of the station that wants to send the data. The byte that was sent to gain control of the net is detected by all the stations trying to read data from the net, and they all examine the header. Any station that finds that the header matches its station number, and comes from a station from which it expects data, will then send an acknowledge byte (set to 1). If this is received by the transmitting station a data block is sent, and the receiving station reads it in. If the acknowledgement byte is not received, the transmitting station repeats the whole operation, including claiming the net. The only time that this protocol can go wrong is if two stations try to claim the net at the same time. In this case the station with the lowest station number will be the first to detect the error and stop transmitting. The other station will then continue sending its claiming byte and complete its data transmission. Using this protocol a number of machines can be sending data over the net, each one waiting its turn to claim the net and transmit its data block.

The network channel descriptor

The network introduces yet another channel descriptor to ZX BASIC. Its format is:

byte	name	comments
0	–	address 8 the error handler
2	–	address 8 the error handler
4	–	"N" the channel identifier
5	–	address of output routine
7	–	address of input routine
9	–	276 length of the channel descriptor
		the header block
11	NCIRIS	the destination station number
12	NCSELF	the station number at the time the channel was OPENed
13	NCNUM	data block number
15	NCTYPE	type of data block (0=data 1=EOF)
16	NCOBL	number of data bytes in block
17	NCDCS	data checksum
18	NCHCS	the header checksum
		general infomation
19	NCCUR	the position of the last character taken from buffer
20	NCIBL	the number of bytes in the input buffer
		data block
21	NCB	255 byte data buffer

The format of the channel descriptor is straightforward, and should be compared to the channel descriptions for the Microdrives and the RS232 interface. Notice that NCSELF contains the station number at the time of OPENing. This means that it is quite possible to have a number of net channels OPEN, each with a different station identifier.

The net from assembler

There are a number of machine code routines in the new 8K ROM that can be used by the assembly language programmer. The calling procedure is the same as for the Microdrives and the RS232 interface, that is,

```
RST 8
code
```

where 'code' can be any of:

code	operation
45	OPEN a temporary net channel. The system variable D_STR1 should contain the destination station number and NTSTAT (23749) the current station number.
46	CLOSE a network channel. The IX register should contain the address of the channel descriptor.
47	READ a net record. The IX register should contain the address of the channel descriptor.
48	Write a net record. The IX register should contain the address of the channel descriptor. A=0 will write data. A=1 will send an end of file record.

Note that in the above descriptions a net record is a complete network transaction, including the initial control byte, header and data block. The 'read a net record' routine should return with the carry flag set if no record is received in a reasonable amount of time. However, there seems to be a bug that corrupts the carry flag and makes the routine almost unusable. This may be corrected in later versions of the new 8K ROM.

Service Spectrums

One of the desirable features of a network is the sharing of peripherals. Obviously if the same program is to be loaded into all the machines connected to a net, then only one machine needs to have Microdrives of its own. In the same way, it would be useful to be able to share a printer between all the machines on the net. This

can most easily be achieved by designating one machine as a *printer and Microdrive server*. This machine simply runs a program that accepts data from the network and routes it to the appropriate peripheral. There are many ways of implementing a server program (one can be found in the Interface 1 manual), but none of the methods that I have seen are entirely satisfactory. However, it is important to realise that to share peripherals between a number of Spectrums a machine must be set aside to run the server program, and this reduces the number of available machines by one.

Chapter Twelve

Advanced Programming Applications

This final chapter presents a collection of self-contained applications. Most of them use information from the earlier chapters, but some new material is also introduced. Advanced programming can take one of two forms. The first is concerned with writing good, clear, easy-to-use and bug-free programs. The second is that described in this chapter, and is concerned with using the facilities of the machine in novel ways. However, this sort of advanced programming assumes you have mastered the art of writing simple programs that have a clear structure, operate in a user-friendly fashion, and contain a minimum of bugs. Being clever with a machine is no reason for abandoning good programming style!

Byte arrays

Sometimes the need to store a large array of numbers with a limited range makes the direct use of a BASIC numeric array very inefficient. Each element of the array uses five bytes, but if the numbers lie within the range 0 to 255 then theoretically each element need only occupy a single byte. In practice it is quite easy to create special byte arrays using nothing but PEEK and POKE. Our requirements are for a statement that will 'dimension' the array by reserving N bytes for it, a function that will return the Ith element, and a function that will store the Ith element. The dimensioning is not difficult, as the CLEAR command can be used to reserve any number of bytes for special use. However, for the subroutine to work without modification in a 16K or 48K Spectrum it must automatically find the highest memory location in use. This can be done by PEEKing the system variable RAMTOP at 23730. Thus the function

```
DEF FNd(N)=PEEK 23730+256*PEEK 23731-N
```

returns the address N bytes lower down than the highest memory location currently in use, and the statement

```
CLEAR FNd(N)
```

will reserve N memory locations for the byte array. The functions to store and retrieve data are

```
DEF FNs(I)=PEEK 23730+256*PEEK(23731)-I
DEF FNr (I)=PEEK(FNs(I))
```

The statement

```
POKE FNs (I),D
```

will store the data D in array element I and

```
LET D=FNr (I)
```

will retrieve the data stored in element I and store it in D.

For an example of the use of these ideas, the following program stores 256 numbers in a byte array:

```
20 CLEAR FNd(256)
30 FOR I=0 TO 255
40 POKE FNs(I),I
50 NEXT I

60 FOR I=0 TO 255
70 PRINT FNr(I)
80 NEXT I
```

Using a byte array only takes .25K; a standard array would need 1.25K to store the same data.

The same technique can be used to store numbers greater than 255 by using more than one memory location per element.

Passing parameters to USR functions

The advantage of machine code routines implemented via USR functions has been proven many times in earlier chapters. However, most of the examples have carried out some action without attempting to return a value in the manner of a normal function. In fact USR functions return the 16-bit number in the BC register pair. For example, the program

```
LD BC,42
RET
```

will return the value of 42 if called as a USR function. What limits the usefulness of machine code USR functions is the difficulty of passing parameters to the routines. One method that has been used a number of times in earlier chapters is to use fixed memory locations as *post boxes*. A post box is used to pass data to machine code user routines by POKEing it into the locations before calling the routine with USR. This works, but it isn't very flexible and doesn't fit in with the way other functions work.

There is a way of writing machine code routines so that they accept standard ZX BASIC parameters. The method relies on building the USR call into a user-defined function with the required number of parameters. For example, if you want a machine code routine that will add two 16-bit positive numbers together you could define a function

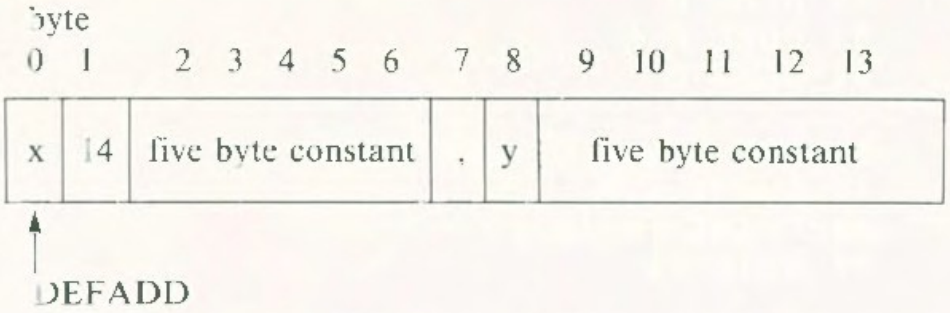
```
DEF FNa(x,y)=USR 23296
```

assuming that the machine code is stored in the ZX printer buffer. The only problem that remains is how the USR function is to gain access to the values of the parameters 'x' and 'y'. The solution lies in the system variable DEFADD (23563), which contains the address of the first parameter of a user-defined function while the function is being evaluated. Thus, in the program

```
10 DEF FNa(x,y)=USR 23296
20 PRINT FNa(2,3)
```

DEFADD will hold the address of the 'x' in line 10 when the function at line 20 is executed. This means that the USR routine can use DEFADD to find the memory location that holds the 'x' in line 10. You may be wondering why the location of the parameter name used in a function is of any use in finding its value. The answer is that when a user-defined function is being evaluated by ZX BASIC, each of the actual parameters used are themselves evaluated and then stored in five bytes following each parameter name in the function definition. This means that each of the parameters is evaluated in line 20, giving the result 2 for x and 3 for y. (Of course the evaluation is often much more complicated, involving full arithmetic expressions and other functions.) Then the result 2 is stored in the five bytes following the letter x in line 10, and the result 3 is stored in the five bytes following the letter y in line 10. Each of these five bytes is preceded by a byte containing 14, the control code indicating that a number follows. This stops the parameter values appearing in program listings. Thus at the time the machine code USR routine is

called, the data stored in line 10 is:



By using the value in DEFADD the USR routine can easily pick up the values of the parameters.

Although it is possible to write routines that process full five-byte floating point numbers, it is much easier if parameter values are restricted to 16-bit integers. A 16-bit integer value is stored in a special format using the second, third and fourth bytes, In fact, if only positive integers are used the 16-bit value can be found in the third and fourth byte of the five bytes

The routine to add two 16-bit positive numbers is now easy to write:

address	assembly language	code	comment
23296	LD IX,(23563)	221,42,11,92	load IX with the address of 1st parameter
23300	LDA A, (IX+4)	221,126,4	load A with 1st byte of 1st parameter
23303	ADD A, (IX+12)	221,134,12	add 1st byte of 2nd parameter to A
23306	LD C,A	79	store result in C
23307	LD A, (IX+5)	221,126,5	load A with the 2nd byte of the 1st parameter
23310	ADC A,(IX+13)	221,142,13	add 2nd byte of 2nd parameter
23313	LD B,A	71	store result
23314	RET	201	return to BASIC

The following ZX BASIC program loads the routine and gives an example of its use:

```
10 DATA 221,42,11,92,221,126,4,221,134,12,
   79,221,126,5,221,142,13,71,201
```

```
20 FOR A=23296 TO 23314
30 READ D
40 POKE A,D
50 NEXT A

60 DEF FNa(x,y)=USR 23296
70 INPUT A,B
80 PRINT FNa(a,b)
90 GOTO 70
```

If you enter integer values in response to line 70 you will find that their sum is PRINTed by line 80. You might like to experiment with using FNa in more complicated expressions. For example, change line 80 to

```
80 PRINT FNa(A,FNa(A,A))
```

to add A to A+A. The point is that this method of passing parameters to a machine code routine results in a function that can be mixed with other functions, and used in exactly the same way that they can. Of course, adding two 16-bit numbers together is not a very useful operation for a machine code function, but in the next section the same idea will be used to add the standard logical functions to ZX BASIC.

Bit manipulation - AND, OR and NOT

One of the common features of programs that use a machine's hardware directly is *bit manipulation*. The reason for this is that the state of a particular bit or group of bits often reflects or controls the condition of some hardware. Another reason for wanting to examine and change bits, or groups of bits, is the use of different parts of a byte to hold different pieces of information. For example, an attribute byte uses b7 for flashing on/off, b6 for bright on/off, and b5 to b3 and b2 to b0 for paper and ink colours respectively.

In other versions of BASIC, bit manipulation is performed using the logical operators AND, OR and NOT, but in ZX BASIC these operators behave differently. In normal use in ZX BASIC these operators work with the values 0 and 1, representing false and true respectively. For example, the result of x AND y is 1 if both x and y are 1, and 0 if either of them is 0. This corresponds to the usual English interpretation of AND that 'x and y' is true only if both x is true and y is true. However, ZX BASIC interprets any non-zero

value as true, and this gives rise to the following results when x and y are other then 0 or 1:

x AND y	= x if y is non-zero
	= 0 if y is 0
x OR y	= 1 if y is non-zero
	= x if y is 0
NOT x	= 0 if x is non-zero
	= 1 if x is zero

These results are useful for writing conditional expressions as described in Chapter 13 of the Spectrum manual, but they are not suitable for bit manipulation.

Other versions of BASIC implement AND, OR and NOT with *bitwise* operations that are much more useful in bit manipulation. For example, the result of a bitwise AND operation is arrived at by ANDing the corresponding bits in each of its operands: b0 of the result is arrived at by ANDing b0 of the first operand with b0 of the second, and so on. Thus the result of a bitwise AND of 7 and 12 is

```
7           = 00000111
12          = 00001100
7 AND 12 = 00000100
```

or 4 in decimal. The Spectrum's AND operation gives the result 7.

The importance of bitwise operations for bit manipulation is that you can set any bit or group of bits to zero by ANDing them with a *mask* value, and you can set any bit or group of bits to one by ORing them with a mask value. To be precise:

(1) To set any bits to zero, construct a mask value consisting of ones in every bit position apart from the bit positions that you want to set to zero. This mask should then be bitwise ANDed with the value that contains the bits that are to be set to zero.

(2) To set any bits to one, construct a mask value consisting of zeros in every bit position apart from the bit positions that you want to set to one. This mask should then be bitwise ORed with the value that contains the bits that are to be set to one.

For example, to set b7 to b4 to zero the byte would have to be bitwise ANDed with

```
b7 b6 b5 b4 b3 b2 b1 b0
0  0  0  0  1  1  1  1
```

i.e. 15 in decimal. To set b7 and b6 to one the byte would have to be bitwise ORed with

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

i.e. 192 in decimal.

Of course, the trouble with these methods is that ZX BASIC doesn't have bitwise AND, OR and NOT operators. This can easily be remedied using the technique described in the previous section for parameter passing to USR routines. The following assembly language routine will perform the bitwise AND between two 16-bit integers:

address	assembly language	code	comment
23296	LD IX,(23563)	221,42,11,92	get parameter address
23300	LD A,(IX+4)	221,126,4	1st byte 1st parameter
23303	AND (IX+12)	221,166,12	AND with 1st byte of 2nd paramete
23306	LD C,A	79	store result
23307	LD A, (IX+5)	221,126,5	2nd byte 1st parameter
23310	AND (IX+13)	221,166,13	AND with 2nd byte of 2nd paramete
23313	LD B,A	71	store result
23314	RET	201	return to BASIC

If this AND routine is compared with the 16-bit addition routine given earlier, you will see that the only difference is that the ADD instructions have been changed to AND. In the same way, a bitwise OR routine can be produced by changing the two AND instructions to

```
OR (IX+12)    221,182,12
```

and

```
OR (IX+13)    221,182,13
```

To complete the set, a single parameter 16-bit NOT routine is provided in the following way:

Address	assembly	language code	comment
23296	LD IX, (23563)	221,42,11,92	get parameter address
23300	LD A, (IX+4)	221,126,4	load 1st byte

23303	CPL	47	complement (NOT) A
23304	LD C,A	79	store result
23305	LD A, (IX,5)	221,126,5	load 2nd byte
23308	CPL	47	complement (NOT) A
23309	LD B,A	71	store result
23310	RET	201	return to BASIC

Although these three routines have been described as if they were each intended to be loaded at the start of the printer buffer, they are in fact position independent, and can be loaded anywhere in memory. The following BASIC program loads the machine code for all three routines into the printer buffer, and defines the three functions:

FNa(x,y) which performs the bitwise AND of x and y
FNo(x,y) which performs the bitwise OR of x and y and
FNn(x) which performs the bitwise NOT of x

```
 10 DATA 221,42,11,92,221,126,4,221,166,
    12,79,221,126,5,221,166,13,71,201
 20 DATA 221,42,11,92,221,126,4,221,182,
    12,79,221,126,5,221,182,13,71,201
 30 DATA 221,42,11,92,221,126,4,47,79,
    221,126,5,47,71,201

 40 FOR A=23296 TO 23348
 50 READ D
 60 POKE A,D
 70 NEXT A

100 DEF FNa(X,Y)=USR 23296
110 DEF FNo(X,Y)=USR 23315
120 DEF FNn(X)=USR 23334

130 INPUT A,B
140 PRINT FNa(A,B),FNo(A,B),FNn(A)
150 GOTO 130
```

As an example of how the AND, OR and NOT functions can be used to simplify things, consider the problem of separating out the information supplied by the ATTR function. In Chapter 6 this problem was solved by bit manipulation techniques based on multiplying and dividing by powers of two. Multiplying by two is equivalent to shifting the pattern of bits that represents a value one

place to the left, and adding a zero to the right. This is equivalent to what happens to the pattern of digits when multiplied by 10. Similarly, dividing by 2 and taking the INTeger part is equivalent to shifting the bit pattern to the right and losing the old value of b0. Using these shift operations it is possible to isolate groups of bits within a byte, and it is even possible to set individual bits to 0 and 1, but it is usually very involved. Using the bitwise logical functions makes the isolation of parts of a byte very easy. For example, to isolate the ink colour (b2,b1,b0) from ATTR is now simple:

```
ink=FNa(BIN 111,ATTR(line,col))
```

To isolate the paper colour (5,b4,b3) is just as simple:

```
paper=INT(FNa(BIN 111000,ATTR(line,col))/8)
```

Finally, bright and flash are given by

```
bright=INT(FNa(BIN 1000000,ATTR(line,col))/64)
```

and

```
flash=INT(FNa(BIN 10000000,ATTR(line,col))/128
```

User-defined channels and Interface 1

The subject of adding user-defined channels to the basic Spectrum has already been covered in Chapter 5. However, the addition of Interface 1 and the new 8K ROM introduces an extended format for additional channel descriptors. With Interface 1 connected, the smallest channel descriptor corresponds to the 11 bytes that describe an RS232 channel. Of course, there is nothing wrong with changing the address of the I/O handler in the channel descriptor, and the first example given in the section **Creating your own channels** in Chapter 5 will work with Interface 1 connected. However, if you are going to create an entire channel descriptor, it is better to make it fit in with the extended formats introduced by the 8K ROM.

The channel descriptor for your new channel should have the following format:

---

byte	
0	address of output routine
2	address of input routine
4	one letter channel name

```
5    40 address of 8K ROM error routine
7    40 address of 8K ROM error routine
8    11 length of channel descriptor
```

The only difference between this and the RS232 channel descriptor is the use of the first four bytes to hold the addresses of the I/O handlers, and bytes 5 to 7 to hold the address of the error handler in the 8K ROM. The reason for this is that any I/O handlers that you write are going to be stored in RAM and not in the new 8K ROM.

Apart from this change in format, the channel descriptor also has to be stored in the channels area of memory rather than in the printer buffer, as in Chapter 5. To accomplish this, space of 11 bytes must be made in the channels area using the 16K ROM routine MAKESP (5717). This will produce an area of RAM for any purpose by shifting all of the used areas of RAM up by the desired amount, and correcting all the system variables that are affected by this change. The amount of space to be created is passed in the BC register pair, and the address of the first location of the free area is passed in the HL register pair. Thus

```
LD BC,100
LD HL,23700
CALL 5717
```

will create a free area 100 bytes long starting at 23700. When a new channel descriptor is added to the channels area, the extra space required is positioned at the end of all the existing channel descriptors. Thus the area for a user-defined channel should be created starting at one less than the address stored in the system variable PROG. The following routine will create the required 11 bytes of space and insert a new channel descriptor:

```
address  assembly language     code          comment
                    make 11 bytes of room
23296  LD HL, (23635)  42,83,92      23635 is PROG
23299  DEC HL          43            HL=end of chan. area
23300  PUSH HL         229           save HL
23301  LD BC,11        1,11,0        amount of space needed
23304  CALL 5717       205,85,22     make room
```

```
                    move the channel descriptor
                      into the channels area
23307  LD HL,23338    33,42,91      move the channel
23310  POP DE         209           descriptor given
23311  PUSH DE        213           at the end of this
23312  LD BC,11       1,11,0        routine into the
23315  LDIR           237,176       11 bytes of free space

                    calculate offset for
                        stream table
23317  POP HL         225           calculate 'offset'
23318  LD BC,(23631)  237,75,79,92  value for
23322  AND A          167           stream table
23323  SBC HL,BC      237,66
23325  INC HL         35

                    store in stream table
23326  LD (23582),HL  34,30,92      store in stream 4's
23329  RET            201           entry

                    output driver
23330  OUT LD BC,245  1,254,0       output routine
23333  OUT (C),A      237,121
23335  RET            201

                    input driver
23336  IN RST 8       207           input routine
23337  DEFB 18        18            invalid device error

                    channel descriptor
23338  DEFB 34        34            address of OUT routine
23339  DEFB 91        91
23340  DEFB 40        40            address of IN routine
23341  DEFB 91        91
23342  DEFB "E"       69            channel identifier
23343  DEFB 40        40            error handler
23344  DEFB  0         0
23345  DEFB 40        40            error handler
23346  DEFB  0         0
23347  DEFB 11        11            length of channel
23348  DEFB  0         0
```

The output and input routines that the channel descriptor uses are the same as used in the examples in Chapter 5, and simply send data to the border port. Although this routine will OPEN stream 4 by default, this can be changed by storing the address of a different stream table entry in locations 23327 and 23328.

The ZX BASIC program given below loads the machine code routine and provides an example of its use.

```
 10 DATA 42,83,92,43,229,1,11,0,205,85,22
 20 DATA 33,42,91,209,213,1,11,0,237,176
 30 DATA 225,237,75,79,92,167,237,66,35
 40 DATA 34,30,92,201
 50 DATA 1,254,0,237,121,201
 60 DATA 207,18
 70 DATA 34,91,40,91,69,40,0,40,0,11,0
 80 FOR A=23296 TO 23348
 90 READ D
100 POKE A,D
110 NEXT A

120 LET S=4:GOSUB 1000
130 PRINT #4;0;
140 PRINT #4;7;
150 GOTO 130

1000 LET A=23574+2*S
1010 POKE 23327,A-INT(A/256)*256
1020 POKE 23328,INT(A/256)
1030 LET A=USR 23296
1040 RETURN
```

Lines 10 to 110 form the usual machine code loader. Subroutine 1000 will OPEN stream S to the new channel descriptor, and lines 130 to 150 PRINT 0 and 7 to the border channel, so making it flash black and white. Notice that this method of adding a user-defined channel will work both with and without Interface 1 connected.

Adding commands to ZX BASIC

With Interface 1 attached, creating new ZX BASIC commands is fairly easy as long as you are a good Z80 assembly language programmer. The key to adding your own commands is the way that errors are handled when Interface 1 is connected. When an error

occurs, a RST 8 command is used to call the standard error handler. However, as already described, the error call is intercepted by Interface 1 and the new 8K ROM is paged in. This examines the nature of the error and checks to see if the command that has caused it can be correctly handled by it – that is, if it is one of the new commands implemented by the 8K ROM. If it is one of its commands, the appropriate machine code routine is called, and then control is returned to the standard 16K ROM. If the command is not recognised by the 8K ROM, control is returned to the 16K ROM at the address given by the new system variable VECTOR (23735). Normally this contains the address of a final error handling routine, but this address can be changed to transfer control to a user-supplied routine that makes a final attempt to recognise and implement whatever command has been rejected by both the 16K and the 8K ROM.

Changing the address in VECTOR effectively intercepts the normal processing of faulty ZX BASIC and extended ZX BASIC statements. This implies that any command added to BASIC in this way must normally cause an error. For example you could add commands such as

```
*T
ASN
PAUSE*
```

each of which causes an error because it is not recognised by either ROM. This guarantees that its processing will be passed on to the routine that VECTOR 'points to'.

In practice, adding commands is quite involved, and a good knowledge of the layout of the 16K ROM is essential. If you are going to extend ZX BASIC then there is no way you can avoid using many of its routines. However, when control is passed to your routine via VECTOR, the new 8K ROM is still paged in. To call routines in the 16K ROM you should use

```
RST 16
DEFW address
```

where 'address' is the address of the 16K routine that you want to use. All the registers are returned as the 16K ROM routine leaves them. While the 8K ROM is paged in, all of the RST addresses are different from what you would expect with the 16K ROM in action. The most important are:

RST 32 report an 8K ROM error,
the error code follows the RST 32
RST 40 report a 16K ROM error,
the error code is stored in ERRNO
RST 48 create new system variables

Routines to implement new commands always have the same overall form:

(1) a syntax checker
This checks to see if the new command has the correct form. If it hasn't, then an error should be reported by jumping to location 496. The syntax check should scan the line to its end and leave CH_ADD pointing to the end of the line. The end of the statement should be tested for by calling subroutine 1463 in the 8K ROM. If the syntax is only being checked, control will not return from this subroutine, but if the program is being RUN then control passes on to the second half of the routine.

(2) a RUN time module
This part of the routine actually does the work required to implement the new command. When the RUN time module is finished, it should return control to ZX BASIC by jumping to 1473 in the 8K ROM.

Two 16K ROM routines that are indispensible in writing new commands are

address	function
24	get current character in BASIC line in A register
32	get next character in BASIC line in the A register. Successive calls to this routine will advance the current character, so scanning the line

The 'next character' routine will automatically skip over spaces and control codes, so it should always return the next 'useful' character.

As a simple example of adding a command, the following routine implements the command

PAUSE *

which will halt processing until a key is pressed.

address	assembly language	code	comment
		syntax check	
23296	RST 16	215	get command code
23297	24	24,0	
23299	CP 242	254,242	PAUSE ?
23301	JP NZ,ERR	194,240,1	error
23304	RST 16	215	get next char.
23305	32	32,0	
23307	CP 42	254,42	is it '*'
23309	JP NZ,ERR	194,240,1	error
23312	RST 16	215	move to end of statement
23313	32	32,0	
23315	CALL CKEND	205,183,5	Check for end of statement
		run time	
23318	LOOP XOR A	175	zero A
23319	IN A, (254)	219,254	scan keyboard
23321	AND 31	230,31	keep only lower 5 bits
23323	SUB 31	214,31	A=31 if no key pressed
23325	JP Z,LOOP	202,22,91	loop until key pressed
23328	JP COMEND	195,193,5	return to 16K ROM

The syntax check part of the routine tests for the keyword PAUSE, and then the character "*". As long as it finds them, control is passed to CKEND which only returns control to the routine if the BASIC program is being RUN. The RUN time part of the routine simply loops until a key is pressed, and then returns via COMEND which pages in the 16K ROM and allows the BASIC program to continue. The following BASIC program loads the machine code and POKEs the new value to VECTOR.

```
10 DATA 215,24,0,254,242,194,240,
   1,215,32,0,254,42,194,240,
   1,215,32,0,205,183,5
20 DATA 175,219,254,230,31,214,31,
   202,22,91,195,193,5
30 FOR A=23296 TO 23330
40 READ D
50 POKE A,D
60 NEXT A
```

```
70 POKE 23735,0
80 POKE 23736,91
```

After running this program the command PAUSE * will be accepted as part of a program, and will cause the program to wait until a key is pressed.

Routines to add other new BASIC commands take the same form – a syntax checker and a RUN time module – but normally the RUN time module would be a lot more complicated than the one given in the example.

A stats program

The last few examples have made a great deal of use of Z80 assembly language. To illustrate the way that knowledge of the internal workings of the Spectrum can prove useful, even in apparently straightforward ZX BASIC programs, the following example presents a statistics program that will edit data, calculate statistics, plot histograms and save and load data on tape.

The first problem is how to store the data to be analysed. The most obvious method is to use a one-dimensional numeric array. This can easily be SAVEd and LOADed, and allows as much data as can be held in RAM to be analysed and edited rapidly. However, using an array is not without its problems. The first is that when an array is LOADed using

```
LOAD "filename" DATA D()
```

the number of elements in the array is not immediately accessible. When data is created by the program, it is not difficult to keep track of the number of elements in a variable – N, say. The problem is how to set the value of N when an array is read in from tape. One answer would be to store N in one of the array elements before it was written out, but this is an unnecessary complication to the data storage scheme. Using the information about the format of array storage given in Chapter 4 (see Fig. 4.1), it is possible to write a few lines of ZX BASIC that will PEEK the dimension of the array. The question is how to find the position in memory of the start of the array. One way would be to write an assembly language routine that searched the variables area for the array, but there is a much simpler way. The system variable DEST (23629) holds the address of the destination variable during an assignment. Thus if we want to find the address of

the first element of the array D, all that is necessary is

```
LET T=D(1)
LET D(1)=PEEK 23629+256*PEEK 23630
```

Following this

```
LET N=PEEK(D(1)-1)+256*PEEK(D(1))
LET D(1)=T
```

will store the dimension of the array in N and restore the value in D(1).

The only other real problem is how to add data to an existing array. If the array holds N values, and the user wishes to add M values, then the array has to be extended to DIM D(N+M) without losing any of the original data. This could also be achieved using an assembly language routine, but once again ZX BASIC is enough. To extend the array D to N+M, first dimension an array DIM E(N) and copy all of the existing data from D to E. Then re-dimension D to DIM D(N+M) and copy all the data back to D leaving M elements free, ready for the new data. Finally re-dimension the array E to DIM E(1) to release the space it occupied. Not the fastest method, but very simple!

Now that these two problems have been solved the resulting stats program is:

```
 10 REM stats program

500 CLS
510 PRINT TAB 5;"S t a t i s t i c s"
520 PRINT AT 6,0
530 PRINT "(1) Enter new data"
540 PRINT "(2) Generate random data"
550 PRINT "(3) Edit data"
560 PRINT "(4) Save/Load data"
570 PRINT "(5) Calculate Statistics"
580 PRINT "(6) Plot histogram"
590 PRINT "(7) Quit"
600 PRINT AT 21,0;"Type required number"
610 INPUT sel
620 IF sel=1 THEN  GOSUB 3000
630 IF sel=2 THEN  GOSUB 1000
640 IF sel=3 THEN  GOSUB 4000
650 IF sel=4 THEN  GOSUB 1500
660 IF sel=5 THEN  GOSUB 5500
670 IF sel=6 THEN  GOSUB 6000
```

```
 680 IF sel=7 THEN  STOP
 690 GOTO 500

1000 CLS
1010 PRINT "Random data"
1020 PRINT "How many values";
1030 INPUT n
1040 PRINT n
1050 DIM d(n)
1060 PRINT AT 3,0;"Fractional or integer
     data f/i";
1070 INPUT a$
1080 IF a$<>"f" AND a$<>"i" THEN  GOTO 1060
1090 PRINT a$
1100 LET t=0
1110 IF a$="i" THEN  LET t=1
1120 PRINT AT 4,0;"lowest value ";
1130 INPUT l
1140 PRINT l
1150 PRINT "highest value ";
1160 INPUT h
1170 PRINT h
1180 IF h>l THEN  GOTO 1210
1190 PRINT "highest<lowest!"
1200 GOTO 1120
1210 FOR i=1 TO n
1220 LET d(i)=RND *(h-l+t)+l
1230 IF t=1 THEN  LET d(i)=INT d(i)
1240 PRINT "data value ";i;" = ";d(i)
1250 NEXT i
1270 GOTO 8900

1500 CLS
1510 PRINT "Save or load data s/l"
1520 INPUT a$
1530 IF a$<>"l" AND a$<>"s" THEN  GOTO 1500
1540 IF a$="l" THEN  GOTO 1600
1550 PRINT "Filename "
1560 INPUT f$
1570 SAVE f$ DATA d()
1580 GOTO 8900
1600 PRINT "Are you sure"'" you want to load
     data"
1610 INPUT a$
1620 IF a$="n" THEN  GOTO 8900
1630 IF a$<>"y" THEN  GOTO 1600
```

```
1640 PRINT "Filename "
1650 INPUT f$
1660 LOAD f$ DATA d()
1670 LET t=d(1)
1680 LET d(1)=PEEK 23629+256*PEEK 23630
1690 LET n=PEEK (d(1)-1)+256*PEEK d(1)
1700 LET d(1)=t
1710 RETURN

2000 LET m=0
2010 LET s=0
2020 LET l=d(1)
2030 LET h=l
2040 FOR i=1 TO n
2050 LET m=m+d(i)
2060 IF l>d(i) THEN  LET l=d(i)
2070 IF h<d(i) THEN  LET h=d(i)
2080 NEXT i
2090 LET m=m/n
2100 FOR i=1 TO n
2110 LET s=s+(d(i)-m)*(d(i)-m)
2120 NEXT i
2130 LET s=s/(n-1)
2140 RETURN

2500 CLS
2510 PRINT "number of values= ";n
2520 PRINT "maximum= ";h
2530 PRINT "minimum= ";l
2540 PRINT "range= ";h-l
2550 PRINT "mean= ";m
2560 PRINT "variance= ";s
2570 PRINT "standard dev.= ";SQR (s)
2580 GOTO 8900

3000 CLS
3010 PRINT "Data input"
3020 PRINT "how many values ?";
3030 INPUT n
3040 PRINT n
3050 DIM d(n)
3060 FOR i=1 TO n
3070 PRINT AT 21,0;"value";i;" = ";
3080 INPUT d(i)
3090 PRINT d(i): PRINT
```

```
3100 NEXT i
3110 PRINT "data input complete"
3120 GOTO 8900

4000 CLS
4010 PRINT TAB 5;"Edit Data"
4020 PRINT AT 5,0
4030 PRINT "(1) list data"
4040 PRINT "(2) alter data"
4050 PRINT "(3) delete data"
4060 PRINT "(4) add data"
4070 PRINT "(5) return to main menu"
4080 PRINT AT 21,0;"Type required number"
4090 INPUT ed
4100 IF ed=1 THEN  GOSUB 4200
4110 IF ed=2 THEN  GOSUB 4500
4120 IF ed=3 THEN  GOSUB 4600
4130 IF ed=4 THEN  GOSUB 4800
4140 IF ed=5 THEN  RETURN
4150 GOTO 4000

4200 CLS
4210 PRINT "list starting at ?";
4220 INPUT l
4230 PRINT l
4240 PRINT "list ending at (-1 will list
     to end)? ";
4250 INPUT h
4260 PRINT h: PRINT
4270 IF h<0 THEN  LET h=n
4280 IF l>h THEN  GOTO 4200
4290 IF l>n OR h>n OR l<1 OR h<1 THEN
     GOTO 4200
4300 FOR i=l TO h
4310 PRINT "data value ";i;" = ";d(i)
4320 NEXT i
4330 GOTO 8900

4500 CLS
4510 PRINT "alter which value? ";
4520 INPUT i
4530 IF i<1 OR i>n THEN  GOTO 4500
4540 PRINT i
4550 PRINT "current value = ";d(i)
4560 PRINT "new value = ";
```

```
4570 INPUT d(i)
4580 PRINT d(i)
4590 GOTO 8900

4600 CLS
4610 PRINT "delete starting from ";
4620 INPUT l
4630 PRINT l
4640 PRINT "ending at ";
4650 INPUT h
4660 PRINT h
4670 IF h<l THEN  GOTO 4600
4680 IF h>n OR h<1 OR l>n OR l<1 THEN
     GOTO 4600
4690 PRINT
4700 PRINT "delete from ";l;" to ";h
4710 PRINT "is this ok ?";
4720 INPUT a$
4730 PRINT a$
4740 IF a$(1)<>"y" THEN  RETURN
4750 FOR i=h+1 TO n
4760 LET d(l+i-h-1)=d(i)
4770 NEXT i
4780 LET n=n-h+l-1
4790 RETURN

4800 CLS
4810 PRINT "how many extra values ?";
4820 INPUT m
4830 PRINT m
4840 DIM e(n)
4850 PRINT "making space"
4860 FOR i=1 TO n
4870 LET e(i)=d(i)
4880 NEXT i
4890 PRINT "nearly ready"
4900 DIM d(n+m)
4910 FOR i=1 TO n
4920 LET d(i)=e(i)
4930 NEXT i
4940 PRINT "ready"
4950 DIM e(1)
4960 FOR i=n+1 TO n+m
4970 PRINT "data value ";i;" = ";
4980 INPUT d(i)
```

```
4990 PRINT d(i)
5000 NEXT i
5010 LET n=n+m
5020 GOTO 8900

5500 CLS
5510 PRINT "calculating"
5520 GOSUB 2000
5530 GOTO 2500

6000 CLS
6010 PRINT "how many bins ? ";
6020 INPUT m
6030 PRINT m
6040 PRINT "maximum value= ";
6050 INPUT h
6060 PRINT h
6070 PRINT "minimum value= ";
6080 INPUT l
6090 PRINT l
6100 IF h<l THEN  GOTO 6000
6110 LET d=(h-l)/m
6120 GOSUB 7000
6130 FOR i=1 TO m
6140 PRINT AT 21,0;INT (l*100)/100;TAB 6;
6150 IF h(i)=0 THEN  GOTO 6190
6160 FOR j=1 TO h(i)/f*25
6170 PRINT CHR$ 143;
6180 NEXT j
6190 PRINT : PRINT
6200 LET l=l+d
6210 NEXT i
6220 PRINT : PRINT
6230 GOTO 8900

7000 DIM h(m)
7010 FOR i=1 TO n
7020 LET j=(d(i)-l)/(h-l)*m+1
7030 LET j=INT j
7040 IF j<1 OR j>m THEN  GOTO 7060
7050 LET h(j)=h(j)+1
7060 NEXT i
7070 LET f=0
7080 FOR i=1 TO m
7090 IF f<h(i) THEN  LET f=h(i)
```

```
7100 NEXT i
7110 RETURN

8900 PRINT
8910 PRINT AT 21,0;"press any key to
     continue"
8920 IF INKEY$="" THEN  GOTO 8910
8930 RETURN
```

This is the longest program in this book, and as such it contains many techniques and ideas that only become important when programs become large! In particular, notice the use of the menus to allow users to select the action that they want, and the extensive INPUT checking that attempts to stop invalid data getting into the program and crashing it. The extensive use of subroutines should make the program easier to understand, extend and maintain. The following subroutine table should enable you to find your way around:

line number	description
500– 690	main menu
1000–1270	generate random data
1500–1710	SAVE and LOAD data
2000–2140	calculate statistics
2500–2580	print results
3000–3120	data input
4000–4150	editing menu
4200–4330	list data
4500–4590	alter data
4600–4790	delete data
4800–5020	add data
5500–6230	plot histogram
7000–7110	construct frequency count
8900–8930	press any key to continue

The modifications to allow this program to work with data stored on Microdrives are simple, if you do not want to take advantage of the increased storage they offer. The direct way of changing the program is to change the SAVE...DATA and LOAD...DATA statements into SAVE*...DATA and LOAD*...DATA statements. Apart from this, the only other change would be to the form of the filenames used. However, a better way of organising data on the

Microdrives is to use PRINT files. Instead of storing all of the data in an array, a PRINT file could be used, and only a single item of data would be read in and processed at a time. This increases the amount of data that can be processed to the capacity of the Microdrive, rather than the amount of memory that is left over for variable storage. The cost of this scheme is time. Each time the data is required the entire file has to be read, and operations such as appending new data are even more time-consuming.

Using Interface 2

Interface 2 is a very simple piece of hardware that allows standard joysticks and ROM software cartridges to be used with the Spectrum. The most important thing about Interface 2 as far as the programmer is concerned is that it introduces a standard for the use of the keyboard in dynamic games. The joysticks are connected as a 'duplicate' set of top row keys, thus:

Direction	joystick 1	joystick 2
	key	key
Left	6	1
Right	7	2
Down	8	3
Up	9	4
Fire	10	5

Although these keys can be read using the standard INKEY$ function, there is an advantage in using IN 61438 to read joystick 1 and IN 63486 to read joystick 2. The user-defined logic functions can be used to test which keys are pressed. For example

```
500 LET A=IN 61438
510 IF FNa(a,BIN 1)=0 THEN PRINT "fire";
520 IF FNa(a,BIN 10)=0 THEN PRINT "up";
530 IF FNa(a,BIN 100)=0 THEN PRINT "down";
540 IF FNa(a,BIN 1000)=0 THEN PRINT "right";
550 IF FNa(a,BIN 10000)=0 THEN PRINT "left";
560 PRINT
570 GOTO 500
```

will PRINT the appropriate words when any key or group of keys is pressed. Notice that this program has to be added to the definition of the logic functions given earlier in this chapter.

Conclusion

There is no end to ways in which a knowledge of the inner workings of the Spectrum can be put to good use. The most important advice I can give concerning your own programming projects is to take them seriously! It is all too easy to start a project without any clear objectives, and give up when the going gets tough. If you start with an accurate idea of what you want the software to do, and set the specifications high enough, overcoming the difficulties is a satisfying challenge. Don't give up – try to isolate your problems and write routines to investigate what is happening. A finished program that does what you planned is a sufficient reward for any amount of effort.

Appendix

Further Reading

There are a huge number of books and magazines published concerning the Spectrum, and the range they present is bewildering. If you are looking for some further reading, you might find the following suggestions helpful.

If you are a self-taught BASIC programmer then *The Spectrum Programmer* by S. M. Gee (Granada, 1982) will help you to clear up any gaps in your knowledge. It starts at a fairly elementary level but very quickly gets to the more interesting features of ZX BASIC, graphics and sound. Even if you are an accomplished programmer you will find something to learn from the early chapters about the natural structure of a BASIC program.

A more advanced book about good programming style is my own *The Complete Programmer: A guide to better programming in BASIC*, (Granada, 1983). This is not a book about the Spectrum in particular but about BASIC in general, and more sophisticated programming techniques. In this sense it covers the machine-independent side of advanced programming.

At a slightly lower level than this book, but with many examples illustrating fundamental programming ideas, is *The Art of Programming the Spectrum* by Mike James (Bernard Babani Publishing, 1983).

The collection of games in *The Spectrum Book of Games* by Mike James, S.M. Gee and Kay Ewbank (Granada, 1982) also illustrates many of the techniques described in this book. In particular 'Laser Attack' uses functional characters, and 'Fruit Machine' uses the technique of internal animation described in Chapter 7.

If you are interested in developing Z80 assembly language programming then my own favourite is *The Z80 Microcomputer Handbook* by William Barden (Sams, 1978) but it does include a lot of hardware information as well as software details. For an introduction to Spectrum machine code see *Introducing Spectrum*

Machine Code by Ian Sinclair (Granada, 1982), and if you are already a proficient assembly language programmer then *The Complete Spectrum ROM Disassembly* by Dr Ian Logan and Dr Frank O'Hara (Melbourne House, 1983) is an essential companion. It contains everything you could ever want to know about the 16K ROM, but you often have to work quite hard to find it and understand it!

Finally it is worth mentioning that if you are interested in the more advanced software and hardware aspects of the Spectrum, and computing in general, then *Electronics and Computing Monthly* provides a welcome relief from the diet of games programs found in some other magazines, and invariably has some stimulating information relevant to Spectrum applications.

Index

THE SINCLAIR SPECTRUM . . . FOR CONNOISSEURS!

The continued success of the Sinclair Spectrum has been phenomenal. This book is a practical introduction to the advanced features of the Spectrum covering both hardware and software. It is aimed at the Spectrum user who seeks a deeper understanding of the Spectrum and its capabilities, starting with an inside view of the micro, followed by a connoisseur's guide to ZX BASIC and an introduction to the machine operating system. The ZX video is covered in detail and chapters are devoted to the tape system, the RS232 interface, the microdrive and advanced programming techniques. Complete program listings and projects are provided throughout for readers to explore the more sophisticated capabilities of the machine.

The Author

Mike James is the author of several very successful books on programming and has been a regular contributor to *Electronics and Computing Monthly* and other popular magazines.

Other books on the Spectrum

THE ZX SPECTRUM
and how to get the most from it
Ian Sinclair
0 00 383071 5

THE SPECTRUM BOOK OF GAMES
M. James, S. M. Gee and K. Ewbank
0 246 12047 9

THE SPECTRUM PROGRAMMER
S M Gee
0 246 12025 8

INTRODUCING SPECTRUM MACHINE CODE
Ian Sinclair
0 246 12082 7

SPECTRUM GRAPHICS AND SOUND
Steve Money
0 00 383136 1

COLLINS
Printed in Great Britain
0 00 383134 5

£6.95 net