

THE SPECTRUM PROGRAMMER



S.M.GEE

The Spectrum Programmer

The Spectrum Programmer

S. M. Gee

Editorial Adviser: Henry Budgett

GRANADA

London Toronto Sydney New York

Granada Publishing Limited – Technical Books Division
Frogmore, St Albans, Herts AL2 2NF
and
36 Golden Square, London W1R 4AH
515 Madison Avenue, New York, NY 10022, USA
117 York Street, Sydney, NSW 2000, Australia
100 Skyway Avenue, Rexdale, Ontario, Canada M9W 3A6
61 Beach Road, Auckland, New Zealand

Copyright © 1983 by S. M. Gee

British Library Cataloguing in Publication Data
Gee, S.M.

The Spectrum programmer.

1. Sinclair ZX Spectrum (Computers)—Programming

I. Title

001.64'2 QA76.2.S62/

ISBN 0-246-12025-8

First published in Great Britain 1983 by Granada Publishing Ltd
Reprinted 1983 (twice)

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

All rights reserved. No part of this publication may be reproduced,
stored in a retrieval system, or transmitted in any form or by any
means, electronic, mechanical, photocopying, recording or otherwise,
without the prior permission of the publishers.

Granada®
Granada Publishing®

Contents

<i>Preface</i>	ix
1 Before You Switch On	1
What is a computer?	1
Programs and programming	4
The history of the ZX Spectrum	5
2 Getting to Know Your Spectrum	7
A new acquaintance	7
Setting up the Spectrum	8
Using the keyboard	10
Entering a program	12
More about the keyboard	14
Using a tape recorder	17
Editing and our second program	19
As you go along	22
3 First Steps – Variables, PRINT, LET and INPUT	23
Variables	23
Storing things in variables – LET	24
Finding out what's in a variable – PRINT	25
Arithmetic	26
Understanding expressions – the order of evaluation	27
Variables and constants – the full expression	28
A short program	29
Another way of altering variables – INPUT	29
Variables and constants as expressions	30

Describing BASIC	31
INPUT prompting	32
Mixed PRINT	32
Some sample programs	33
4 Looping and Choice – the Flow of Control	38
The flow of control	38
Looping – the GOTO	38
Choices and conditions – the IF statement	41
Using IF	44
The FOR statement	49
Using the FOR loop	52
IF . . . THEN and the colon	52
A final example	54
The flow of control summarised	55
5 Handling Text and Numbers	57
Strings	57
String expressions	58
Arrays	63
A word game	67
Initialising variables – DATA and RESTORE	68
Saving data on tape	69
6 Functions and Subroutines	71
The idea of a function	71
The Spectrum's functions	73
User-defined functions DEF FN and FN	80
Subroutines GOSUB and RETURN	83
Using subroutines	84
7 Graphics	86
Controlling PRINT	86
PRINT functions – TAB and AT	88
A full screen – CLS and scrolling	90
Controlling INPUT	91

The graphics characters	93
User-defined graphics characters	96
Changing the way characters look – INVERSE and OVER	99
Character attributes – FLASH and BRIGHT	101
Colour – BORDER, INK and PAPER	102
Display commands in colour	104
Temporary colours	105
Using graphics in games	106
8 Sound and Games	107
Simple sounds – BEEP	107
Programming tunes	109
Resting – PAUSE	112
Some sound effects	113
Attack the saucer – the SCREEN\$ function	114
9 High-resolution Graphics	118
The high-resolution screen	118
The graphics commands – PLOT, DRAW and CIRCLE	119
High-resolution colours	124
Un-plotting – OVER and INVERSE	125
Finding out what's on the screen – POINT	126
Using hi-res graphics	127
10 Logic and Other Topics	130
Logic and the conditional expressions	130
Inside the Spectrum – BIN, PEEK, POKE, IN, OUT, USR and CLEAR	133
Finding out the colour – ATTR	136
REMark and good programming	136
Where next?	137
<i>Further Reading</i>	138
<i>Index</i>	139

Preface

This book has been written for all Spectrum users who want to learn to write their own programs. Programming your Spectrum can be great fun and very rewarding and learning BASIC is by no means a daunting undertaking. This book is intended to take you from very fundamental routines to proficient programming by clear and easy-to-follow stages.

Chapter One introduces some ideas that are relevant to all computers and programming in general and looks at the Spectrum's family tree. Chapter Two deals with getting ready to learn programming on your Spectrum and it includes a program that provides your Spectrum with its very own test card for tuning the TV set. We start to learn programming in Chapter Three which introduces variables and some of the essential keywords, such as PRINT, INPUT and LET that are used to handle them. Chapter Four introduces the idea of the flow of control which is central to computer programming. Chapter Five shows how your Spectrum handles both numbers and words and how you can combine both in your programs. The uses of the Spectrum's functions and the related idea of subroutines are the main topics in Chapter Six and this is where the very important commands RND, RANDOMISE and INKEY\$ are discussed. The book pays a great deal of attention to the Spectrum's graphics and colour starting in Chapter Seven. Sound is added in Chapter Eight which also presents a games program which uses colour graphics and sound with exciting effects. Chapter Nine explains the Spectrum's high-resolution graphics capabilities and includes lots of short programs that show off its features in this respect. The final chapter covers the topic of logic and also introduces some commands that allow the programmer to control the Spectrum's inner workings in a direct way.

As you will already realise, this book covers a lot of ground. However, it is not meant to be hard going – it is meant to be fun.

More importantly, it is not meant as a reading book – it is meant as a try-it-yourself book. There are lots of programs and program snippets included and they are all there for you to use. So stand by your Spectrum and enjoy learning BASIC.

I've actually had a good deal of fun writing this book and have also learned some interesting things both about the Spectrum, which I think is a remarkable micro, and about BASIC, which has long been the best computer language as far as I'm concerned. I would like to express my thanks to Jane Patience who helped with the not-so-enjoyable stages of presenting my publishers with a readable manuscript.

Chapter One

Before You Switch On

When you first set eyes on the Spectrum you may be surprised by just how small it is. You may be even more amazed if you know that only twenty years ago a computer with a similar memory capacity would have filled the whole of a family house. You may wonder whether the circuitry inside so slim and compact a case can really do all the things the adverts claim for it. Well it *can* do all those things – it can tackle complicated calculations, produce colour graphics and emit a variety of sounds – and come up with many more extras besides, and any limitations it has are not really to do with its small exterior although they are governed by its price. However, the fact that modern micro-technology has enabled so much to be squeezed into so small a space is a fact that the Spectrum user can cheerfully ignore, for at the same time as computer design is becoming more sophisticated, computer use is becoming accessible to everyone who wants to join in.

What is a computer?

This question is one that can be answered at many levels. Whole chapters, even whole books, can be devoted to the subject. In order to use your Spectrum, however, you really do not even need to ask this question. After all, we all watch TV but few of us ever question exactly what a television is. If you want to know every detail of the way the Spectrum works then there is no choice but to learn about electronics. However it is not difficult to gain an understanding of what a computer does and roughly how it does it without knowing anything about electronics or the ever present *chip*! The point is that a computer is something that would exist even if electronics had never been invented. Indeed the first computers were built using cogs and gears and it took one hundred years before a valve (an early

2 *The Spectrum Programmer*

electronic component) found its way into such a machine. Although in practical terms the computer seems to be a product of microprocessor technology, the idea that lies behind a computer doesn't depend on the materials that you choose to build it from.

Every computer is composed of a number of parts that each perform a well-defined function. Any computer has to have some way of communicating with the outside world. In the case of the Spectrum this need is met by a keyboard, on which you type, and the TV screen, which the Spectrum can use to show you what you have typed and anything else it needs to tell you. The keyboard is an example of an *input* device and the TV screen is an *output* device. These are not the only input/output or I/O devices that can be used with a computer. You can buy a small printer that can be used with the Spectrum for example and you can direct the Spectrum to produce its output on this printer instead of, or as well as, the screen.

A machine that could only receive information and pass it on unchanged wouldn't really be worth calling a computer. Rather it might be classed as a telephone or a telex! Inside every computer there has to be some mechanism that can change or *process* information before it is printed out. This mechanism usually takes the form (these days at least) of complex electronics hidden inside the computer. What we are talking about is often referred to as the *Central Processing Unit* or CPU but it also has a traditional English name that betrays the fact that computers were once made of cogs and gears – the *mill*.

In the Spectrum the CPU is contained in a single chip known as a Z80 and this is of course the origin of the Z in the Spectrum's full name – the ZX Spectrum. *What* exactly the Z80 does isn't of too much importance from the point of view of programming in BASIC and the *way* that it does it certainly isn't! In general, however, what the Z80 does is to perform arithmetic and other operations on information input from the keyboard and stored within the machine. What operations it does are controlled by a list of instructions called a program. This aspect of a computer is so important that you could almost say that a computer *is* a machine that will obey a list of instructions – but any sort of definition of a machine as complicated as a computer is dangerous! What sort of instructions the Spectrum can obey will occupy the rest of this book, so for the moment the subject will be set aside.

If a computer is going to obey a list of instructions concerning what to do with various pieces of information it must obviously have somewhere to store not only the information but also the list of

instructions. This part of a computer is known as *memory* but the slightly less general term RAM (standing for *Random Access Memory*) is almost universally used instead. You can think of RAM as a sort of note pad where the CPU can record its list of instructions and any data that it needs. Obviously every memory has a limited capacity and this is an important measure of how powerful a computer is. The larger the memory the larger the list of instructions that can be stored. The most convenient unit of measurement to apply to computer memory is the *byte*. Roughly speaking a memory that can store one byte, can store one character. (Here the term character means a letter, a digit or any punctuation that you might find in a normal text – such as this book!) So a 400 byte memory could store enough characters to hold about a quarter of a page of this book. The only trouble with this convenient unit of measure is that it is a little too small. Computers normally have memories that can store thousands of characters and so it makes good sense to think in terms of thousands of characters. The unit used for this is the *kilobyte*, which is often shortened to kbyte or even just K. For various reasons, however, 1 kbyte isn't 1000 bytes as its name suggests, but 1024 bytes. (You may notice that this strange number is the nearest power of two to 1000 and, as you might already know, computers work in binary which is based on *two* states.) There are two versions of the Spectrum that differ only in the amount of memory that they can use – either 16K or 48K. So even the *smallest* Spectrum can store roughly 16000 characters which is enough for a wide range of interesting applications. (Early computers that were used by the military to calculate missile trajectories, etc., often had less than 16K!)

This combination of I/O devices, CPU and memory is all that there is to a computer. The I/O communicates with the outside world, the CPU calculates and generally processes information and the memory holds the list of instructions that the machine obeys and the data that the CPU acts on. In practice, there is one addition that we must make to this list. When you switch your Spectrum off it *forgets* everything stored in its memory. To keep information stored accurately, most computer memories need a constant supply of electricity. If you switch off the supply, the information is lost. This sort of memory is often known as *volatile* memory. This loss of memory is something of a problem because it implies that we have to type in the list of instructions every time that the Spectrum has been switched off. To overcome this difficulty most computers have a second form of memory that is – *non-volatile*. In the case of the

4 *The Spectrum Programmer*

Spectrum, this takes the form of a standard cassette tape recorder that can be used to *save* programs and data in a form that exists even when the power has been switched off. A second advantage with this type of memory is that it is *removable*. You can record a program on a cassette and then take it out of the recorder (and even send it to someone else). The Spectrum is then ready for you to start on a new program or go back to an old one, which you can do by *loading* it from an earlier recorded tape. The Spectrum can also use a second, but less common form of removable storage – the *Microdrive*. This can be purchased as an addition to the basic Spectrum and it works very much like a cassette recorder except a lot faster! If you are serious about computing, or become serious by the end of this book, then a Microdrive is a must.

Programs and programming

As mentioned earlier, a computer obeys a list of instructions stored in its memory. This list of instructions is known as a program and writing such lists of instructions is known as *programming*. It is often thought that programming is an activity that started with the modern digital computers, but people have been writing lists of instructions for other people to obey since writing was first invented. In this sense, programming is nothing new and can be seen in the form of recipes and knitting patterns in almost every home. Perhaps one of the best examples of *traditional* programming is written music. You can think of sheet music as being a program that will instruct a musician to play a specific tune. In fact written music is very like a computer program in that it relies on using a special language that is much more precise than ordinary language. Just one note out of place and you have a different tune! A computer program is written using a special and equally precise language. In the case of the Spectrum this language is BASIC, the most popular programming language in the world. Just as with written music, slight changes in a BASIC program can alter its meaning completely, so it is important to realise as you learn BASIC that you must pay attention to the fine details right from the very beginning. Unlike learning English, where you can first learn words and sentences and then add punctuation, you have to take notice of every comma in a line of BASIC for it to make any sense at all!

If all this talk of strict rules is worrying you it is worth saying that the rules are usually very simple and very regular. Unlike English

there are rarely any exceptions to spelling and punctuation rules in BASIC! In addition, there are some powerful underlying ideas behind BASIC. Once you have recognised these they make it easy to understand why the rules are there at all. As you progress through this book there are therefore two types of thing that you will learn – the fine detail concerning the exact form of each BASIC statement and the general features that all programming languages share. The final detail is important to actually getting a program working but understanding the general details makes the act of programming a sensible occupation.

The history of the ZX Spectrum

Before moving on to a discussion of the Spectrum it might be of interest to take a brief look at its family tree. In 1980 Sinclair Research launched a small plastic-cased computer, the ZX80, that brought computing within the reach of nearly everyone. The trouble was that it was very limited. It was a revolution, but in many senses it was just a little before its time. It could be used to run small programs written in BASIC but the sort of arithmetic that it could do was restricted to whole numbers. It could display information on a TV screen but only while it wasn't processing data. If it was doing anything at all useful the TV screen flickered disturbingly. This meant that a lot of people who bought the ZX80 for various reasons were disappointed to discover that it just couldn't live up to their expectations of a computer.

In 1981 Sinclair launched the successor to the ZX80, the ZX81. The ZX81 is the first really useful computer to figure in this brief history. It had very little memory (1K) but at least its screen didn't flicker and for the first time it was possible to use good graphics – both static and moving! The ZX81 is also notable because it introduced Sinclair- or ZX-BASIC which is used in a slightly extended form in the Spectrum. ZX-BASIC is a fully developed version of BASIC that has many advantages over the BASIC on other machines. To make up for the small memory size Sinclair produced a 16K add-on RAM pack and to extend the machine's range of use a small low cost printer was offered. The 16K RAM pack can only be used with the ZX81 but the printer is still available and works very well with the Spectrum. (The program outputs in this book were produced using just such a printer.)

In 1982 Sinclair announced the Spectrum to complement the

ZX81. The Spectrum added sound, colour and high-resolution graphics to the capabilities of the ZX81 and generally offered an improved performance for a slightly higher price. In addition to the Spectrum, Sinclair also introduced a very low cost storage device, the Microdrive, and a communications interface (neither of which can be used with the ZX81). The availability of these two extras guarantees the Spectrum a place in both popular and serious applications for some time to come.

Seen as a steady development of the Sinclair range, the Spectrum is a logical and well-designed machine that builds on the experience gained both from the ZX80 and the ZX81. This ensures it a place as an important and popular computer. ZX-BASIC has also progressed steadily and has taken its place as an important and popular programming language. Learning this BASIC is, therefore, likely to stand you in good stead for now and for the future.

Chapter Two

Getting to Know Your Spectrum

There are two problems in using a computer. The first is simply getting it set up and getting used to its idiosyncrasies. The second is writing working programs. This chapter deals with both these problems so that we can get them out of the way before we get down to the main task of learning BASIC.

A new acquaintance

Getting to know a computer is a problem that exists even if you're an expert. For although there is a lot in common between different computers, there are always enough little differences to mean that there has to be a period of *adjustment* when moving from one machine to the next. For example, nearly every computer uses a standard typewriter (or QWERTY) keyboard but most place *extra*, but very important, keys in slightly different places and this can make even the most expert look silly at first! Now, if you're an expert, then you know that this early phase soon passes but if you're a beginner you may panic and think that computing is always going to be this tricky! The trouble is that not being 'at home' with your computer can make easy programming ideas seem difficult.

There is no way to avoid this early barrier to programming because being on friendly terms with your computer is simply a matter of time and a matter of using it. You'll come through this rather frustrating period more easily if you bear in mind the following advice:

- (1) Separate in your mind any difficulty that you encounter in using your Spectrum from any difficulties that you have with programming.
- (2) Don't immediately assume that any *strange* behaviour on the part of your Spectrum is its fault – at first the chances are that the mistake is yours!

- (3) Don't immediately assume that any unexpected behaviour of a program means that your Spectrum is illogical – computers are ruthlessly logical. Try not to confuse typing errors with programming errors.

To try and help you identify this initial difficulty and to help you overcome it, this chapter includes two short programs that you should try to get running on your Spectrum before moving on to the rest of the book. They are presented as complete and working programs for you to use to find out about the *non-programming* problems of using the Spectrum. At this stage you are not expected to be able to understand how they work and you might like to return to them as you read through later chapters to see how you're progressing.

Setting up the Spectrum

The Spectrum is one of the world's easiest computers to get going. All you have to do is to plug the power supply (the small black box marked ZX POWER SUPPLY) into the mains and then insert the small cylindrical plug into the socket on the back of the Spectrum marked 9V DC. At this point you should be able to hear a high pitched whine, sounding rather like a persistent mosquito, coming from your Spectrum! If you can't hear anything then check that the mains power is switched on. If it is and you still cannot hear anything then press the key marked 'A' on the left of the keyboard. If you keep this key pressed you should hear a low clicking noise coming from your Spectrum. This test should not fail with any working Spectrum.

Connecting the mains to your Spectrum is the only thing that is required to make it start working. However, if you want to see what it is doing you will have to connect it to a working UHF TV set! This operation is a little more difficult than connecting the mains to the Spectrum because it involves a piece of equipment that will vary from house to house – the TV set. When connecting the Spectrum to the TV set, it will help to think of it as being an extra channel. Yes, the Spectrum is not only a computer, it is a small television station! When your TV set was first delivered it had to be tuned-in to receive the channels that are used in your area. In the same way, when you first use your Spectrum you have to tune the TV set to receive it. If your TV set has push button tuning then you will have to decide which button is going to be the *Spectrum channel* and find out how to tune it in from the instruction manual that came with the

set. If your set has dial tuning then finding the Spectrum channel is exactly the same as finding any other channel. However, before you connect the Spectrum to the TV set, either tune the channel button that you are going to use or set the dial to receive BBC 2. (Non-UK readers must consult the tuning information that comes with the Spectrum to discover their equivalent of BBC 2.) The reason for this is that (in most regions) the Spectrum channel is just a little *higher* than BBC 2 and rather than start the search from anywhere it is easier to start from BBC 2 and then tune the set away from the other BBC and ITV channels toward channel 36 – which is the one Spectrum uses.

With the set tuned to BBC 2, take the lead that came with your Spectrum, with the TV aerial plug on one end, and plug it into the TV's aerial socket (removing the original aerial's plug first!). Then plug the other end into the socket at the back of the Spectrum marked TV. Now with both the Spectrum and the TV switched on (you should turn the TV's sound right down to avoid unpleasant noises) start tuning the TV set. As you get close to the Spectrum channel you will begin to see a fuzzy picture. Keep going until you can see the message:

© 1982 Sinclair Research Ltd.

clearly on the bottom of the screen. Just as with any TV channel, if you haven't tuned-in exactly the picture quality will be poor, so take care and be patient with the fine tuning until you have a nice sharp image.

If you are using a colour set then you may have to adjust the brightness, contrast and colour controls to get the best possible picture but don't do this until you have finished tuning for the sharpest picture or things will get hopelessly confused. If you're using a black and white set then you won't be able to see the colours that the Spectrum produces but you still might have to adjust the brightness and contrast on your set to produce a satisfactory picture. A black and white set will work perfectly well with the Spectrum but instead of seeing its different colours you will see eight different shades of grey ranging from black to white.

If you want to, you can leave any final adjustments until after you have entered the program given later in this chapter, because it will produce the Spectrum's very own test card pattern which will help to check the fine tuning.

Using the keyboard

After getting your Spectrum going the next thing to do is to start to learn your way around the keyboard. Perhaps the most off-putting feature of the Spectrum is the number of letters and words that are written on and around the keys of the keyboard. The idea of having each key on a keyboard perform more than one job is not a new one. Most typewriters use a single key to print lower case and capital letters (upper case) and no-one seems upset by the idea of selecting between the two by pressing an extra key – the *shift key*. Notice that the shift key doesn't actually print anything if you press it all on its own so it's not the same as the other keys on the keyboard. As it controls what the other keys produce, it is known as a *control key*.

The Spectrum actually has a number of control keys which are used in different combinations to give access to all its characters, commands and features. The diagram (Fig. 2.1) shows their location on the keyboard. The function of all these control keys will be explained in this chapter, starting with the ones needed most frequently.

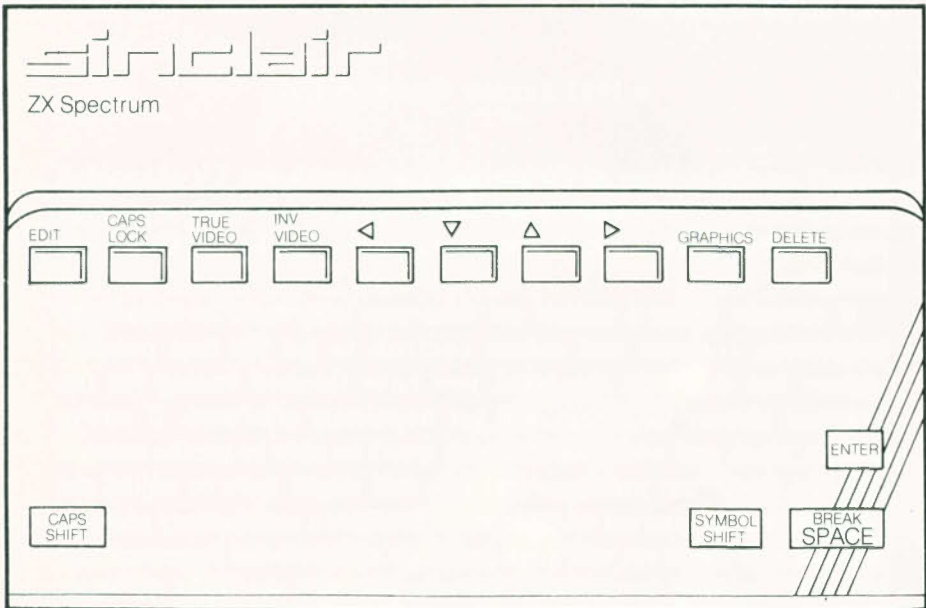


Fig. 2.1. The position of the control keys

The Spectrum uses two control keys to select between three of the sets of characters printed on each key. The CAPS SHIFT key works in the same way as the shift key on a typewriter. If you press a key on

its own you will get the lower case version of the letter written on it. However, if you press the key while holding the CAPS SHIFT key down you will get the upper case letter printed on the key.

The second control key is the SYMBOL SHIFT at the far right of the keyboard. Pressing any key while holding this down produces the red word or symbol written on the key to the right of the main letter or digit.

The use of the two shift keys seems easy enough but if you have just switched on your Spectrum and press the 'A' key you will be surprised to see the word NEW appear on the screen – something has clearly gone wrong! By the above rules pressing just the 'A' key should print a lower case 'a'. The reason for this anomaly is that every line of BASIC begins with one of a small set of words, the *keywords*. To make life easier, the Spectrum interprets the very first letter key that you press as a keyword. The keyword produced by each key is written in white at the bottom. The keyword on the 'A' key is NEW and this is why the word NEW appears on the screen when the 'A' key is pressed for the first time. If you press the 'A' key a second time when the letter 'a' is printed on the screen exactly as predicted. To let you know what any key is going to produce when you press it the Spectrum displays a different letter in the flashing square. This is known as the *cursor*. Its position on the screen indicates where the next printing position is and the letter displayed gives information about which set of keyboard characters will be used. At the start of every line the cursor is a flashing **K** until a keyword is entered and then it changes to a flashing **L**. (**L** stands for keyword and **L** stands for letter.) While the cursor is a flashing **L** the two shift keys work as described above.

To summarise, let's look at one particular key, the one shown below:



The white keywords printed on the bottom of each key are produced when the cursor is a flashing **K**. In the case of our example key the word RUN is produced.

Upper and lower case letters are produced when the cursor is a flashing **L**, lower case when the CAPS SHIFT isn't pressed (that

12 *The Spectrum Programmer*

is 'r' in this example) and upper case (that is 'R') when the CAPS SHIFT is pressed.

The red symbols/words printed on the top right hand corner of the keys are produced by pressing the key at the same time as the SYMBOL SHIFT key. In this case < (the less than symbol) is produced. This works with the cursor showing either \boxed{K} or \boxed{L} .

We now know how four of the symbols or words that surround each key can be produced and this is enough to enter a short program. There is more to using the keyboard but we will return to this later.

Entering a program

As explained in Chapter One, a program is a list of instructions that a computer can obey. In BASIC this list of instructions is built up by typing in lines of commands, each one beginning with a number – the *line number*. The easiest way to understand this is to try out a short program. Don't worry if you don't understand how the program works, just concentrate on entering it correctly. Before you begin, switch your Spectrum off and then, after waiting a moment, on again. This will produce the familiar copyright message on the screen and ensure that anything you may have typed in while experimenting is cleared out of the machine. Now type the following line taking care not to make a mistake. (If you do make a mistake then switch off and start again. This is only a temporary way to overcome mistakes. More satisfactory methods will be explained shortly.)

```
10 LET a=0
```

Notice that the word LET is a keyword and is produced by pressing 'L' while the cursor shows a flashing $\boxed{}$. Also notice that the second and last characters are zeros, not a letter 'O' (the zero (0) is on the far right of the top row of keys). The '=' sign is in red on the 'L' key and so to enter this you have to press SYMBOL SHIFT and 'L'.

After you have typed this line you have to press the key marked ENTER on the far right of the keyboard. The purpose of this is to tell the Spectrum that you have finished typing the line and that it can try to incorporate it into any program that you may have already typed. The word 'try' is used because, even though you may think that what you have typed is correct, the Spectrum checks it and, if it

finds that you have typed nonsense, it will refuse to accept it. When the Spectrum accepts the line it disappears from the bottom of the screen and appears at the top. Now enter the following line:

```
2Ø PRINT a
```

Once again the word PRINT is a keyword and is entered by pressing one key, in this case 'P'. Press ENTER, and the second line will appear, again in the upper part of the screen, just below the first line. Next, enter the following line:

```
3Ø LET a=a+1
```

This time, before you press ENTER, let's see how you could correct any errors that you might have made. Suppose, for example, that by mistake you had typed 'b=a+1'. You might then be pleased to know that the Spectrum offers you a *backspace* facility and enables you to change the line you have typed very easily. If you press the *left arrow* key (the 5 on the top row) while you press CAPS SHIFT you will see the cursor move over the letters that you have entered. If you press the DELETE key (the Ø on the top row) while pressing the CAPS SHIFT, the letter or even a whole keyword to the left of the cursor will vanish (i.e. will be deleted). You can insert a new keyword or letters by simply typing them in. You can move the cursor to the right by using the *right arrow* key (the 8 on the top row) and any new characters that you type will be inserted to the immediate left of the cursor. (This is all a lot easier to see happening than it is to explain so don't be afraid to experiment – you can't hurt your Spectrum.) Finally, when you have finished entering the line, press ENTER. Next, type:

```
4Ø GOTO 2Ø
```

Notice that when you type the key marked GOTO the two words GO TO appear on the screen.

You should now have the following program in the top part of the screen:

```
1Ø LET a=Ø
2Ø PRINT a
3Ø LET a=a+1
4Ø GOTO 2Ø
```



This is a list of instructions that you can make the Spectrum obey by entering the keyword RUN and then pressing ENTER. But before trying this it is worth noticing that there are two ways that the

Spectrum can obey commands. If you type a command without a line number and press ENTER then the Spectrum will obey the command at once – this is called *immediate mode*. However, if you precede the command with a line number and press ENTER then the command is added to whatever program already exists ready to be obeyed at some later time – this is called *deferred mode*. When you type RUN there is no line number so the Spectrum obeys the command immediately.

When you do type RUN you will find that the numbers from 0 to 21 are printed on the screen and then the Spectrum prints what looks like a question, “Scroll ? y/n”. If you press the ‘Y’ key you will discover that the screen is shifted up by one line to make room for more information to be printed. After this has happened 22 times the question is asked again. The reason for this is that the Spectrum tries to give you the opportunity to view what is on the screen before moving it off the top.

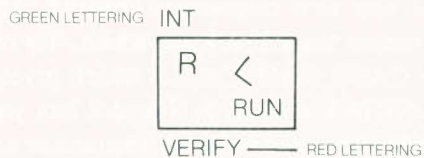
You can keep on saying ‘y(es)’ to this “Scroll?” question until you get tired of seeing numbers. When this happens answer ‘n’ (no) to the next question and then press ENTER. You will see a listing of the program appear on the screen once again. If at any time while a program is running you want to stop it, then press the key marked BREAK while pressing CAPS SHIFT. As another example of a command in immediate mode type LIST (the key marked ‘K’) and press return. This will cause the program you have typed in to be displayed on the screen.



More about the keyboard


If you want to enter a program that is in any way complicated, there are still some symbols and words that we need to know how to produce. In particular, we don’t as yet know how to produce any of the words or symbols written on the Spectrum’s case, as opposed to on the keys. In fact all of these new symbols and words are entered with the cursor in a different mode. In the same way that the keywords are produced when the cursor is showing a flashing , the red and green symbols and words are entered in the so-called *extended mode* with the cursor showing a flashing . You can put the Spectrum into extended mode by pressing both shift keys together. The green words above each key are produced by pressing the key while in extended mode and the red words below each key are produced by pressing the key along with the SYMBOL SHIFT


key. This is easy to remember because to produce any symbol or word in *red* you have to press the SYMBOL SHIFT key which is also lettered in red. So to enter RND (above the 'T' key) you first have to enter extended mode by pressing CAPS SHIFT and SYMBOL SHIFT and then press 'T'. If you want to enter BEEP (below the 'Z' key) first enter extended mode and then press SYMBOL SHIFT and 'Z'. You automatically leave extended mode as soon as you type anything but in case you want to leave it without typing anything you can press the two shift keys again.

To summarise this new information, let's extend the example we looked at before. The illustration below shows the 'R' key and its surrounding section of case.








The words in green above each key are produced when the cursor is a flashing , after both CAPS SHIFT and SYMBOL SHIFT have been pressed. Once  has appeared typing the 'R' key will result in INT being displayed.

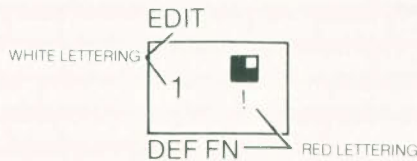
The words in red below each key are produced when the cursor is a flashing  and the SYMBOL SHIFT key (or the CAPS SHIFT key) is depressed. Such action would result in VERIFY in our example.



Now you know how to enter nearly all of the symbols and words on the keyboard. The only ones left that cause any trouble are those written in white on the top row of keys. These have to be entered in yet another mode – the *graphics mode*. While the subject of graphics is treated in full in Chapter Seven it is worth saying how to get into graphics mode here. To enter graphics mode all you have to do is to press GRAPHICS (over the 9 in the top row) and CAPS SHIFT at the same time. This changes the cursor to a flashing  and if you press any of the keys on the top row apart from 9 and 0 you will find that the shapes printed on the key in white are produced on the screen. If you press any of the keys on the top row while holding down CAPS SHIFT you will find that the shapes are produced inverted, i.e. white becomes black and vice versa.

You may be wondering what happens if you press any of the

number keys on the top row with the CAPS SHIFT held down in the normal  mode. After all there is no such thing as an upper case number! The answer is that no new characters are produced. Instead, you gain access to the rest of the control functions shown in Fig. 2.1. Pressing the 1 key and CAPS SHIFT moves the line that has the flashing cursor in it into the input area of the screen where you can *edit* it. More about this later in the chapter. Pressing the 2 key and CAPS SHIFT activates the CAPS LOCK. The first time you press it you will see the cursor as a  where you would expect an  and all the letters you type on the screen will be capitals. To get back to lower case again you have to press CAPS SHIFT and 2 again. You'll see INV. VIDEO printed above the 4 key. Again this is accessed with the CAPS SHIFT and has the effect of reversing the screen display of any new characters entered. They then show as white on black rather than black on white. To regain the normal display you type CAPS SHIFT and 3 which gives TRUE VIDEO. Pressing CAPS SHIFT with 5, 6, 7 and 8 lets you use the *cursor control keys* which allow you to move the cursor around the screen. These are used in games as well as in editing text. We have already met the left (5) and right (8) arrow keys and the other two will be introduced soon. As mentioned above, pressing CAPS SHIFT and 9 allows you to enter graphics mode, signalled by a flashing  cursor. If you are already in this mode, pressing this combination will restore you to the ordinary  mode. CAPS SHIFT and 0 allows you to delete the character to the immediate left of the cursor while entering or editing a line.


The number keys have a rather different appearance from the others on the Spectrum so it is worth examining one in detail before leaving the topic of the keyboard for a while. The diagram below shows the key that is at the top left hand corner – the 1 key – and the following is a summary of the ways of producing all the words and symbols on and around it and the other number keys.




When the  or  cursor is flashing, pressing a number key on its own will produce the number on it, in this case 1.

Pressing it with the SYMBOL SHIFT key will produce the red symbol on the bottom right of each key, in this case an exclamation mark.

Pressing it with CAPS SHIFT has the effect of calling the control function indicated in white above it. The control function associated with each number key has been detailed above. In this case it will cause a line of the current program to be displayed at the bottom of the screen, preparatory to editing.

When the  cursor is flashing and the SYMBOL SHIFT key is held down the commands printed in red on the Spectrum's case below each key are executed. In the case of the 1 key the command allows a function to be defined.

The graphics symbol on the top left of each key is produced when the key is pressed in graphics mode. In the case of the 1 key this is a solid square with the top right hand quadrant missing.

One other facility of the keyboard is worth mentioning before leaving this section. It is the ability to repeat any key by holding the key down for a prolonged period. If you try keeping your finger on a key when the  cursor is flashing you will see the line filling with the character and hear a clicking sound from your Spectrum.

At this point you might be thinking that the Spectrum's keyboard is the most complicated thing that you have ever come across. And indeed it is complicated, but as you get used to it you will find that it is very logical.

Using a tape recorder

The next demonstration program is considerably longer than the first and as it is likely that you will be loath to simply switch the machine off and so lose it altogether once you have finished entering and running it, now is the time to learn how to use a cassette recorder to save and load programs. The Spectrum can use almost any standard tape recorder to store programs, but it is true that the better the tape recorder, the more reliable the result. So if you are thinking about buying a cassette recorder for use with your new Spectrum, invest at least £20 or so and try to buy a model that uses miniature jack sockets for earphone and microphone connections. If the tape recorder that you plan to use doesn't use miniature jack sockets then you will have to buy an adaptor from your local hi-fi

shop because miniature jack plugs are all that the Spectrum comes equipped with.

To connect the Spectrum to your tape recorder simply plug the two jack plugs on the twin audio lead that came with your machine (you can easily recognise it because it's the only lead not already in use by this point) into the two sockets marked MIC and EAR on the back of your Spectrum. It doesn't matter at this stage which plug goes into which socket. Next plug the same colour jack plug that is in the MIC socket into the microphone socket on your tape recorder. You are now ready to record your first program.

To give the Spectrum something to save type in

```
1Ø REM this is a test
2Ø REM this is the second line
3Ø REM this is the third line
```

and then type

```
SAVE "test"
```

and press ENTER. The message "start tape and press any key" will appear. At this point your Spectrum is all set to record a program on tape but it is waiting for you to tell it that the tape recorder is running. Insert a blank tape, set the record level to about half way (this is unnecessary if your recorder has an automatic volume control) and set it recording. When you press any key you will see a pattern of horizontal lines appear around the edge of the screen – this means that the Spectrum is recording a program. When the message "OK" appears on the screen the program is saved on tape. However, it is still possible that although the program is recorded on tape it may not be good enough to be read back into the Spectrum for one of a number of reasons. For example, you may have the record level set too high or there may be a fault in the tape.

To check that the program has been successfully saved rewind the tape, unplug the jack plug from the microphone socket on the tape recorder and plug the *other* jack plug into the earphone socket. Now type:

```
VERIFY "test"
```

and press ENTER. Set the tape recorder running and if everything has gone to plan the next message that you should see on the screen will be another "OK" meaning that the program recorded on tape not only can be read but it is the same as the one still in the computer's memory. If everything hasn't gone well you might get an

“R Tape loading error” or simply nothing. This means that for some reason the Spectrum has misread the information on the tape or cannot even find it. In this case the best thing to do is to wind the tape back and listen to it (taking the jack plug out of the earphone socket on the recorder in order to hear what is happening). At the start of the tape you will hear a steady tone. This should be loud but not distorted. If this sounds all right then wind the tape on and listen to an unrecorded patch at the same volume. You should hear a very soft hiss. If it sounds like a rain storm then reduce the volume and perhaps alter the tone control and try again. If an unrecorded patch gives absolute silence then you should try again with the play-back volume increased.

After using VERIFY to check that the program has been saved correctly you can try loading the program for real. First switch your Spectrum off and on again to convince yourself that the test program has been lost and then type:

```
LOAD "test"
```

or

```
LOAD ""
```

and press ENTER. The first version of LOAD will search the tape until a program with the name “test” is found, but the second version will load the first program on the tape irrespective of its name.

Once you have the tape system working don’t alter the volume controls or tone controls unless it is necessary and remember to change the jack plugs each time. Plug in the microphone socket for SAVE and plug in the earphone socket for LOAD. This is very important as the tape system won’t work reliably otherwise.

Editing and our second program

At this stage you should feel confident enough to tackle entering a longer program. The program given below will produce the *test card* pattern shown in Fig. 2.2. Apart from being quite an interesting pattern you could use it to adjust the TV set that you are using with your Spectrum to produce the sharpest image with the best colour. Before you start typing in the program the following notes might help. First, type NEW to clear out any previous program.

Every word, apart from “SPECTRUM TV” in line 130 and “Channel 36” in line 140, is entered using a single key, so search the

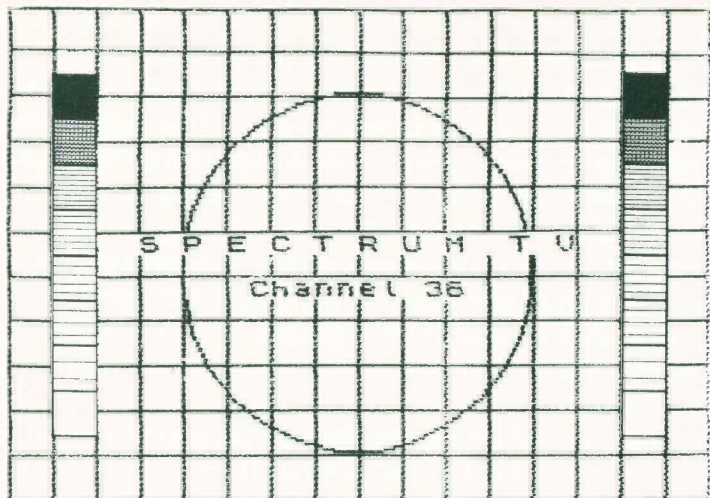


Fig. 2.2. Spectrum test card

keyboard until you find the word! Also, notice in line 110 the minus sign which precedes 175 is obtained by pressing the 'J' key at the same time as the SYMBOL SHIFT key. Because of the difficulty in printing the characters which are entered in graphics mode, a special notation is used. Wherever you see a character in square brackets this indicates that you should enter the character in graphics mode. If the character is preceded by an up-arrow "↑" then press CAPS SHIFT at the same time. So the "[↑8]" in lines 170, 180, 190 and 200 means, "enter graphics mode and press 8 while holding CAPS SHIFT". The result should be a solid black block.

```

10 FOR i=0 TO 240 STEP 16
20 PLOT i,0
30 DRAW 0,175
40 NEXT i
50 FOR i=0 TO 175 STEP 16
60 PLOT 0,i
70 DRAW 255,0
80 NEXT i
90 PLOT 0,175
100 DRAW 255,0
110 DRAW 0,-175
120 CIRCLE 127,81,64
130 PRINT AT 10,6; "S P E C T R U M T V"
140 PRINT AT 12,11;"Channel 36"
150 FOR c=0 TO 7

```

works

```

16Ø INK c
17Ø PRINT AT 3+2*c,2;“[†8]”
18Ø PRINT AT 4+2*c,2;“[†8]”
19Ø PRINT AT 3+2*c,28;“[†8]”
20Ø PRINT AT 4+2*c,28;“[†8]”
21Ø NEXT c
22Ø INK Ø

```

When you have finished entering the program simply press RUN and you should see the test card pattern appear on the screen. If it doesn't look like Fig. 2.2 then it is likely you have made a typing error so check the whole program very carefully against the listing. Press the LIST key so that the program is displayed on the screen. You may have omitted a line. This is easy to remedy. To insert a line just type it, with its line number, and it will appear at the bottom of the screen. Press ENTER and it will automatically assume its correct position in the program listing. Numbering in computer programs is normally done in jumps of ten so that you can easily insert extra lines if you need to. If you want to delete an entire line from a program just type its number and press ENTER and the line will automatically disappear – so do be careful not to delete the program lines you want to keep. If you find an error in a line you do not have to retype the entire line. Instead you can *edit* it. With the program listed on the screen, if you use the up-arrow and down-arrow (7 plus CAPS SHIFT and 6 plus CAPS SHIFT respectively) you will find that you can move the cursor up and down the screen to any line that you desire. If you move the cursor to the line with the error in it and then press EDIT (1 plus CAPS SHIFT) you will find that it appears back at the bottom of the screen (the *input area*) where you first entered it. You can now use the right and left arrow keys (on 5 and 7) and the DELETE key (Ø plus CAPS SHIFT) to edit it and hopefully correct any mistakes. When you have finished editing the line simply press ENTER and the corrected line will take its rightful place back in the program.

You can sometimes save time when entering a program by making use of the editing facility. For example, lines 17Ø to 20ØØ are very similar. So instead of entering each one in turn, enter line 17Ø and then (with the cursor over it) press EDIT. This will copy the line back into the input area where you can change the line number to read 18Ø and the 3 following the AT to read 4. If you then press ENTER you will find that line 18Ø appears in the program without you having had to type it all in.

As you go along

As you learn BASIC and the special features of the Spectrum from the rest of this book you are bound to improve, both in your understanding and your use of the keyboard until you cannot understand what all the fuss was about. However, until then it is wise to recall the advice given at the start of this chapter and try not to let the frustration produced by typing errors interfere with your understanding of computing in general and BASIC in particular.

Chapter Three

First Steps - Variables, PRINT, LET and INPUT

A program is a list of instructions that your computer can carry out. The question that this poses is what sort of instructions can you use in a computer program? It is clear that instructions like 'go and make a cup of tea' are too vague for anything other than a human to cope with! Instructions used in a computer program must be precise. They have to specify exactly what must be done and, perhaps less obviously, they have to specify what it has to be done to. In other words a computer instruction tells the computer what to do and what to do it to. In this chapter we will look at the simplest *objects* in BASIC and some very simple things that you can do with them.

Variables

The idea of a *variable* is the most important single idea in programming. A variable is an area of computer memory that is used to store information. This sounds like an easy idea but it has one or two subtle points. If you are going to store information in an area of memory you are going to need some way of referring to it. You're going to have to give it a name! This is not such an unusual idea if you think about other, more traditional, ways of storing data. For example, each file in a filing cabinet is normally given a name that identifies it and it alone. Just think of the confusion of asking for a file if two files had the same name! It is just the same with BASIC. An area of memory that is used to store information, a variable, must be given a unique name that can be used to refer to it. The only additional difficulty with a BASIC variable is that you must also define what sort of objects you are going to store in the memory area. One reason for this (we will meet others later) is that the amount of memory set aside to store the information depends on its type. For the time being the only sort of information that we will

24 *The Spectrum Programmer*

store in memory will be numbers of any type. A variable that is used to store a number is called a *numeric variable* or a *simple variable*.

You cannot give a variable any name that takes your fancy because this would lead to confusion on the computer's part. For example, suppose you gave the name '1' to a variable. How would the Spectrum know the difference between the variable 1 and the number 1? In the case of the Spectrum you can give a simple variable a name of any length as long as it starts with a letter and thereafter uses only letters and digits. You can use both upper and lower case letters but the Spectrum will not distinguish between them (i.e. the variable name 'A' is treated as being the same as 'a'). You can also insert spaces anywhere in the name to make it more readable but the Spectrum ignores them. Here are some examples of simple variable names that are allowed:

```
sum, SUM, Sum
Totalscore, TOTAL SCORE, TOTALSCORE
This is the longest name that anyone would ever want to use
Total1
total2
day2month3year80
```

Notice that all the versions of 'sum' are treated as the same name and so are all the versions of 'total score'. It is often difficult to think up names for variables that suggest the nature of the information to be stored in them but it is well worth doing. If you come back to read a program after a long time, clear and obvious variable names can make it a lot easier to re-understand your own program! Even so you should try to avoid very long variable names – they can be very boring to type out over and over again in a program. Some examples of names that the Spectrum would not allow are:

<i>name</i>	<i>reason for rejection</i>
1day	starts with a number
*date	starts with an * which is not a letter
date*	contains * which is not a letter and is not a digit
answer?	? is not a letter or a digit
over-time	- is not a letter or a digit

Storing things in variables - LET

Now that we know about variables and how to give them names it is

time to discover how to store information in them. This can be done using the BASIC command LET. For example:

```
10 LET total=56
```

will store the number 56 in an area of memory called 'total'. If you recall, lines of a program are entered with *line numbers* that control their order in the list of instructions that make up the program.

This example is in fact our first program! If you enter it exactly as written nothing will happen until you enter RUN when the Spectrum will start obeying the list of commands. In this case there is only one command and this is very easy to obey – the number 56 is stored in an area of memory called 'total'. If you think about it, just a little more than this has happened. Before the program was run there was no area of memory called 'total' to store 56 in! When the Spectrum comes across the name of a variable that you wish to use to store something in, it checks to see if it already exists and if it doesn't it sets aside an area of memory of the right size and remembers its new name. So this innocent single line program has two effects – it creates the variable called 'total' and then it stores the number 56 in it.

Finding out what's in a variable – PRINT

The one-line example in the previous section is a little disappointing because we have to take on trust that the Spectrum has actually stored the number 56 in a variable called 'total'. What we need is a command that will make the Spectrum find the variable and print its contents on the TV screen. The BASIC command with this effect is, most reasonably, called PRINT. If you add a new line, numbered line 20, to the previous example you will have the following two line program:

```
10 LET total=56  
20 PRINT total
```

If you RUN this program you will be pleased to find that the number 56 is printed in the top left hand corner of your TV screen.

It is important that at this point you understand exactly what is happening as a result of this two line program. Later on when you have absorbed BASIC almost as a second language you will understand what is going on without even thinking about it, but for now it is all too easy to read this two line program and think you

understand it because it *sounds* all right! So, to recap what we have already learned, the first line creates a variable called 'total' and stores 56 in it. The second line finds the area of memory with the name 'total' and prints what is stored in it on the screen. (Notice that it is easy for a beginner to think that PRINT 'total' would print the word 'total' on the screen. So, if you already understand why this interpretation is incorrect you are no longer a beginner!)

For the PRINT statement to work it has to be possible for the Spectrum to find the variable that it refers to. If for some reason you try to print a variable that hasn't been created then the Spectrum will, quite rightly, give you an error message – to see this, delete line 10 from the previous program (simply by typing 10 and ENTER) and RUN it. You should be able to understand why you get a "variable not found" error message at the bottom of the screen. This is your first bug!

Arithmetic

Our programs are slowly becoming more interesting but they are still a long way from being useful. We can now store numbers in variables and print out what is stored in any variable but so what! To be of any use we have to be able to change what is stored in a variable and print out something that we regard as an answer. The key to doing this lies in the idea of an *arithmetic expression*. An arithmetic expression is nothing more than a piece of arithmetic that you haven't yet worked out. For example, $3+6$ is an arithmetic expression that works out or *evaluates* to 9. You can write an arithmetic expression on the right hand side of the equals sign in a LET statement with the effect that the Spectrum will evaluate the expression and store the result in the variable. For example try:

```
10 LET total=3+6  
20 PRINT total
```

You will see 9 printed in the top left hand corner of the screen. As promised the Spectrum has evaluated the expression and stored the result in 'total'.

As with most things to do with computers, there are rules governing what makes a correct expression. You can use the four operations that you should be familiar with from simple arithmetic. Addition and subtraction are indicated by the usual symbols, + and -, but multiplication and division use the symbols, *, and /. The

reason for using * to mean multiply instead of a cross is that the traditional symbol is too easy to confuse with the letter 'x'. Some examples of correct arithmetic expressions are:

<i>expression</i>	<i>evaluates to</i>
3+2	5
3*2	6
6/2	3
3+2-4	1
2.1+3.3	5.4

Apart from the four usual operations of arithmetic there are two others that can be used on the Spectrum – the *unary minus* and *raise to a power*. The unary minus sounds rather grand but it is simply the normal subtraction sign used in front of a single number. For example, the '-' in $3-2$ is the normal subtraction sign but the '-' used in -3 is the unary minus. Although the same sign is used in both cases, as we will see later, they are treated slightly differently. The raise to a power sign is \uparrow . For example, $2\uparrow 2$ is read as two raised to the power of two, i.e. two squared, or four. The raise to a power sign is not used very often and it is mentioned here more for completeness than for its importance.

Understanding expressions – the order of evaluation

Although the idea of an arithmetic expression seems straightforward, there is a hidden complication. For example, if you write the innocent-looking expression $3+2*4$ does it mean three plus two (i.e. five) times four (answer twenty) or does it mean three plus the answer to two times four (i.e. three plus eight, answer eleven). It may seem strange to you that there are two possible ways to work out this expression because you may feel that one of the two methods is obviously correct and the other is equally obviously incorrect. However, even in arithmetic, there are no absolute answers! The correct interpretation is a matter of convention and isn't something that is handed down from on high. The question of whether we do the + or the * first in an expression like $3+2*4$ is settled by a general agreement that multiplication is more important than addition and so it should be done first, making the correct answer eleven. This agreement that multiplication is more important than addition can be formalised in terms of assigning *priorities* to each operation and

carrying out the operation with the highest priority first. The assignment of priorities can be extended to every operation that can be used in an expression (even some that we haven't met as yet). The priorities that the Spectrum uses to sort out the order in which arithmetic should be carried out are:

<i>operation</i>	<i>priority</i>
↑	10 - highest
unary -	9
*, /	8
+, -	6 - lowest

The reason why the priorities start at 10 and finish at 6 is to allow other operators that we have yet to meet to be assigned priorities. To evaluate an expression you should always work out the operators with the highest priority first. If two operators in an expression have the same priority then you should do the one furthest to the left first (i.e. in the absence of any other preference, you work from left to right).

All this may seem over-complicated just to carry out a little arithmetic but it is necessary if you want to write unambiguous expressions. However there is another way of specifying the order of evaluation that can be used to override the usual priorities - brackets (). It is a longstanding convention that any parts of an expression enclosed in brackets are carried out first. For example, although $3+2*4$ is 11, $(3+2)*4$ is 20. If you're ever in any doubt about how the Spectrum will evaluate an expression then put brackets around the parts that you want worked out first. Brackets sometimes waste time and effort but they can never cause trouble!

Variables and constants - the full expression

So far we have looked at arithmetic expressions involving only numbers but there is no reason why we cannot use variables in expressions. If you write an expression such as 'total+3' the Spectrum will find the variable called 'total' and retrieve the number stored in it. It will then add three to this number. For example, if 'total' had 32 stored in it, the expression 'total+3' would evaluate to 35. Notice that there is no suggestion that what is stored in the variable 'total' is in any way altered. Its contents are simply used in the evaluation of an expression. A number

such as 32 is known as a *constant* (because its value never changes) and now we can see that an expression can be made up of variables and constants with the arithmetic operators, +, -, /, *, ↑. An expression always evaluates to a constant and it is this constant that is stored in a variable by a LET statement.

A short program

Using all that we have found out so far about constants, variables and expressions we can now write a short program that adds two numbers together:

```
10 LET number1=23.34
20 LET number2=44.32
30 LET answer=number1+number2
40 PRINT answer
```

If you enter and RUN this program you will see that the sum of the two numbers in lines 10 and 20 are printed by line 40. Although this is a simple example it demonstrates a wide variety of programming ideas. In lines 10 and 20 the by now familiar LET statement is used to store two constants in two variables. In line 30 the arithmetic expression 'number1+number2' is evaluated and the result is stored in a third variable 'answer'. Line 40 prints the contents of 'answer' on the screen. If you think this is easy, so far so good! Try changing lines 10 and 20 to add different numbers together and change line 30 to give you different arithmetic expressions.

Another way of altering variables - INPUT

In the previous example the two variables 'number1' and 'number2' had numbers stored in them by use of the LET statement. This is convenient unless we want to use the program many times with different values. As suggested, the only way that it is possible to change the values stored in the variables is to edit each line before running the program. Obviously what we need is a statement that will allow us to enter any value into the variable *while the program is running*. This is what the BASIC statement INPUT is for. For example try the following program:

```
10 LET number1=5
```

```
20 INPUT number2
30 LET answer=number1+number2
40 PRINT answer
```

When you run this you might be surprised to find that nothing happens! *Don't panic!* What has happened is that line 10 was carried out and 5 was stored in the variable 'number1'. Then the Spectrum moved on to line 20 where it obeyed the command INPUT by waiting for you to type a number and this is why nothing is happening. The Spectrum is waiting for you to type a number and then press ENTER to signal that you have finished typing/correcting the number. It then stores the number that you have typed in the variable 'number2' and proceeds to the next instruction. So if you haven't already done so, run the program and type in a number of your choice. You will be pleased to see your number with five added to it printed in the usual place.

We now have two ways of storing numbers in a variable – LET and INPUT. It is important to understand the difference between the way LET and INPUT work. As in the case of LET, if a variable doesn't exist before its use in an INPUT statement the Spectrum will create it. If you always want to store the same value or the result of an expression in a variable then use a LET statement. If you want to store a different value in a variable each time the program is run, then use an INPUT statement.

Variables and constants as expressions

One of the most powerful features of BASIC is the way that almost anywhere you can use a constant or a variable, you can use an expression as well. For example, in the PRINT statement you can write:

```
30 PRINT number1+number2
```

and the Spectrum will evaluate the expression and print the result.

It is also true that the simplest forms of an expression are the constant and the variable. For example, the number 3 can be thought of as either a constant or an extremely simple expression. Similarly, the variable 'total' can also be thought of as an expression. So not only can you use an expression wherever you might use a variable or a constant, you can use a variable or a constant anywhere that you can use an expression! For example:

```
LET number1=number2
```

and

```
PRINT 3
```

are both valid BASIC statements.

Describing BASIC

It is difficult to describe any language and BASIC is no different. The trouble is that, while it's easy to give an example of what is correct, it is difficult to explain all the possible correct variations. For example, at the start of this chapter the LET statement was introduced by:

```
LET total=56
```

but this gave no hint that you could write things like:

```
LET total=number
```

or

```
LET total=number1+number2
```

To try to overcome this difficulty it is usual to give a definition involving the general types of things that a statement allows. For example, the general form of the LET statement can be written as:

```
LET 'simple variable' = 'arithmetic expression'
```

where the things between the single quotes are not to be taken literally but replaced by an example of the stated type. So in a real LET statement 'simple variable' would be replaced by a variable name such as total, sum, number, etc.

Throughout the rest of this book, BASIC statements will be introduced by examples and then defined in the same way as the LET statement above. As we learn more about a statement it may prove necessary to redefine it to include a wider range of features. So far the two other BASIC statements that we have introduced, PRINT and INPUT can be defined as follows:

```
PRINT 'arithmetic expression'
```

and

```
INPUT 'simple variable'
```

but as we shall see later these are not the final definitions!

INPUT prompting

Although we can now use INPUT to store information in variables, the way the Spectrum just stops and waits for someone to type a number in is a little unsatisfactory. What is required is the ability to print a message on the screen saying something like “type in a number now” or “what is your number”. Such a message is often called an *input prompt* and BASIC provides two similar ways to print such messages. Try the following short program:

```
10 PRINT "this is a prompt"
20 INPUT "what is your number ? ";number1
```

Line 10 will display “this is a prompt” in the top left hand corner of the screen and line 20 displays “what is your number ?” at the bottom of the screen and then waits for you to type a number. In both cases the characters printed on the screen are the ones inside the double quotation marks. A set of characters in double quotes is known as a *literal string* or simply as a *string*. You can use either PRINT or INPUT to produce prompt messages on the screen depending on which is more convenient. As an example of the use of both, consider the following version of the number addition program given earlier:

```
10 PRINT "what is your first number ?"
20 INPUT number1
30 INPUT "what is your second number ? ";number2
40 PRINT number1+number2
```

Mixed PRINT

The ability to print messages on the screen is clearly a very useful facility for other things than just printing prompts. For example, in the last program it would have been better to print a message saying that the number about to be printed was the sum of the two numbers. You can in fact use a single PRINT statement to print more than one thing at a time. For example, change line 40 in the previous program to:

```
40 PRINT number1;"+";number2;"=";number1+number2
```

and you will see that the contents of ‘number1’ are printed, then a space and a plus sign, followed by another space, then the contents

of 'number2' followed by an equals sign with a space on either side of it and the answer. You can consider this PRINT statement as a list of items to be printed, each item in the list being separated by a semicolon and printed in the next free printing position. Our general definition of PRINT can now be updated to read:

PRINT 'print list'

where 'print list' is a list of items separated by semicolons. The items can be either expressions or strings. Each PRINT statement starts printing on a new line. In Chapter Seven we will return to the definition of the 'print list' and expand it to include ways of formatting the information on the screen but this simple version of the 'print list' will satisfy all our requirements until then.

Some sample programs

Even with so little BASIC it is possible to write some useful, if simple, programs. For example, if you want regularly to work out how many dollars you would buy for a given number of pounds then a currency conversion program would be useful. The overall outline of this program is easy to explain as follows:

ask the user for the conversion rate
ask for the number of pounds to be used to buy dollars
print number of pounds times conversion rate

By now you should realise that this program is a simple one for your Spectrum, so before you look at the version given below, try to write your own version. Remember to include input prompting and explanatory messages. There is no one perfect way to write any program, so don't worry if your version is different from the one shown. It could well turn out to be better!

```
10 PRINT "Pounds To Dollar Conversion"  
20 INPUT "What is the conversion rate ? ";rate  
30 INPUT "How many pounds do you want to spend ?";amount  
40 PRINT  
50 PRINT "For ";amount;" pounds "  
60 PRINT "you can buy ";rate*amount;" dollars"
```

Line 10 simply prints a title for the program. Lines 20 and 30 prompt and accept the necessary input. Notice how line 40 is used to print a

blank line to space the output into an input and a results section. Lines 50 and 60 print the answer with some explanation.

As another example, consider the problem of calculating the stopping distance of a car travelling at any speed in miles per hour. The main problem in writing this program is knowing how to calculate the stopping distance. It is important to realise at this stage that no computer can calculate something unless you can explain to it *how* to do the calculation. Looking at the highway code reveals that the stopping distance is made up from two components – a thinking distance and a braking distance. The thinking distance in feet is roughly the same as the speed in mph. The braking distance is a little more complicated and is given by the square of the speed divided by 20. However both of these quantities are very easy for the Spectrum to calculate, so to add to the usefulness of this short program it is reasonable to print out the thinking distance, the braking distance, the overall distance and how many car lengths it takes to come to a stop. This last calculation is based on the fact that the average (UK) car is 14 feet long. The program is now easy to write:

```

10 PRINT "Stopping Distance"
20 PRINT
30 INPUT "Speed in mph=";speed
40 PRINT "At ";speed;"mph"
50 PRINT
60 PRINT "Thinking distance=";speed;" feet"
70 LET brakdist=speed*speed/20
80 PRINT "Braking distance=";brakdist;" feet"
90 PRINT "Overall distance=";speed+brakdist;" feet"
100 PRINT
110 PRINT "Which is ";(speed+brakdist)/14;" car lengths"

```

Line 10 simply prints a title for the program. Line 30 prompts for the only information needed by this program – the speed in mph. This is stored in the variable 'speed'. Lines 40 and 50 start giving the answer to the user by printing the speed that the calculation is for and leaving some space. Line 60 prints the thinking distance, which requires no calculation as it is numerically the same as the speed in mph. The braking distance is calculated by line 70 and printed in line 80. The overall distance is calculated and printed by line 90. Notice that this is an example of using an expression in a PRINT statement. Line 110 calculates and prints the overall stopping distance in terms of car lengths. Notice that you have to put brackets around the

addition to make sure that it is carried out first.

The final example in this chapter is a sizeable and really useful program if you are interested in DIY. A very common problem is that of estimating how much sand, aggregate and cement you should buy to cast a slab of concrete. To help with this difficult task the following program is a “Concrete Calculator”. Once again our first problem is knowing how to do the calculation before we begin writing the program. Looking up the relevant information in a book on building reveals the following – one cubic metre of concrete made up from one part cement, x parts of sand and y parts of aggregate (i.e. a 1: x : y mix) needs

$$\frac{1}{0.025*(1+x+y)}$$

bags of cement and

$$\frac{x*1.5}{(1+x+y)}$$

cubic metres of sand and

$$\frac{y*1.5}{(1+x+y)}$$

cubic metres of aggregate. (It’s not too important to understand why these equations work, it is often the case that a program is written from information that the programmer doesn’t fully understand – and why should it be otherwise!) Converting this information into a program is, once again, relatively easy. The first part of the program should ask for the dimensions of the concrete slab and then calculate its volume. The program should then ask for the mixture ratio and, using the equations given above, work out the number of bags of cement and the volume of sand and aggregate required. Finally, the results should be printed out in a form that the user will find acceptable. The details of the program are:

```

10 PRINT "Concrete Calculator"
20 PRINT
30 INPUT "What is the thickness in mm ? ";thick
40 INPUT "What is the length in m ? ";len
50 INPUT "What is the width in m ? ";width
60 PRINT
70 LET vol=thick*.001*len*width
80 PRINT "Total volume = ";vol;" cubic m"
90 PRINT
    
```

```

100 INPUT "How many parts of sand to one of cement ?
";partsand
110 INPUT "How many parts of aggregate to one of cement ?
";partagg
120 LET total=1+partsand+partagg
130 LET cement=vol/total/.025
140 LET sand=vol*partsand*1.5/total
150 LET agg=vol*partagg*1.5/total
160 PRINT "Using a 1:";partsand;";";partagg;" mix"
170 PRINT "you need "
180 PRINT cement;" bags of cement"
190 PRINT sand;" cubic m of sand"
200 PRINT "and ";partagg;" cubic m of aggregate"

```

Lines 30–80 ask for the dimensions of the slab and both calculate and print the total volume. Lines 100 and 110 ask for the ratio of the mix and line 120 calculates $1+x+y$ used in all of the following calculations. The most interesting lines of the whole program are 130–200. The first few lines (130–150) carry out the main calculations and the last section (160–200) prints the results. If you look carefully at the calculations you will see that an arithmetic expression in BASIC doesn't always look like the equation that it comes from. For example, you might fall into the trap of writing:

$$\frac{1}{0.025*(1+x+y)} \text{ as } 1/0.025*(1+x+y)$$

but the result of this BASIC arithmetic expression is given by dividing 1 by 0.025 and then multiplying the answer by $(1+x+y)$ i.e. it works out:

$$\frac{1}{0.025}*(1+x+y)$$

rather than the equation that we are interested in. The correct expression is:

$$1/0.025/(1+x+y)$$

or if you want to use extra brackets to clarify matters:

$$1/(0.025*(1+x+y))$$

In the section that prints the results, notice the way that each answer is *embedded* in the printed message. Remember that you do not have to write programs that produce results in a standard way - experiment with what looks good and seems natural.

Chapter Four

Looping and Choice - the Flow of Control

So far we have written programs that are a list of instructions that are carried out one at a time from the top to the bottom. Although it is possible to write useful programs using nothing more, programming really only becomes interesting when you can change the order in which instructions are carried out.

The flow of control

If you look at any of the example programs at the end of Chapter Three it should be possible for you to follow through with your finger the order that the instructions would be obeyed by the Spectrum. You can think of this as tracing the *flow of control* through the program. Each instruction has its turn at governing or *controlling* what the Spectrum is doing and it then *passes control* to the next instruction. In the absence of any other information the next instruction is taken to be the next line down. So the *default* flow of control is a line starting at the top of the program and finishing at the bottom. The following is a very simple program that demonstrates this default condition.

```
10 INPUT a
20 PRINT a
30 INPUT b
40 PRINT b
```



Fig. 4.1.

Looping - the GOTO

BASIC provides a single statement for changing the default flow of

control – the GOTO statement. Try the following program:

```

10 LET test=0
20 PRINT test
30 GOTO 20
    
```

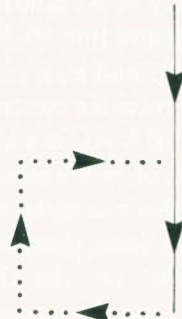


Fig. 4.2.

If you run this program you will see the screen fill with zeros (one to each line) and then the Spectrum will stop and ask “Scroll? y/n”. If you answer y to this question the screen once again fills with zeros and so on. This program is our first example of a loop. Tracing the flow of control through the program soon shows why the word *loop* is appropriate. First line 10 stores zero in the variable ‘test’, then line 20 prints the contents of ‘test’. Line 30 is new in that it uses GOTO but its meaning should be clear from just reading it. The statement GOTO 20 causes the Spectrum to obey line 20 as the next instruction. So after line 30 control is transferred to line 20 and the contents of variable ‘test’ are printed on the screen for a second time. After this line 30 is again carried out. This transfers control back to line 20 and so on ... until the screen is full of zeros and the Spectrum asks for permission to *move the screen on* to make room to print even more zeros! The repetition of lines 20 and 30 will continue forever and tracing the flow of control shows it to take the form of a loop. Not a very useful loop however and, because the only way to stop it is to press the BREAK key or answer n to the “Scroll? y/n” question, it is usually called an *infinite loop*.

A GOTO statement can be used to force the Spectrum to carry out any instruction next. Its general form is:

```
GOTO 'arithmetic expression'
```

so you can write things like:

```
GOTO 2*10
```


(meaning the same thing as GOTO 20) but this sort of thing is not used very often and it is better to think of GOTO as:

GOTO 'line number'

You may be wondering what happens if you write something like GOTO 50 and line 50 doesn't exist. Most versions of BASIC would simply stop and give you an error message to the effect that you are trying to transfer control to a non-existent line but the Spectrum's version of BASIC is a little different. If the line number doesn't exist then control is transferred to the line with the next highest line number. For example, if you use GOTO 50 and line 50 doesn't exist but line 55 does, then control will pass to line 55. If there is no 'next highest line' i.e. the GOTO tries to transfer control out of the program then the program stops without an error message. This means that it is virtually impossible to get an error by using a GOTO! You may think that this is an advantage but take great care because if you have made a typing error in a GOTO the Spectrum won't tell you that you are transferring control to a line that doesn't exist. This might not cause any problem at first because the next highest line number might just be the line you wanted to GOTO anyway, but if you insert any lines of program later on trouble might appear from nowhere!

Although the example of the infinite loop serves to introduce the idea of transferring control to a different point in the program, it doesn't really indicate the sort of thing that a loop is used for. An important idea in programming is the repetition of series of operations. This is often called *iteration*. For example, the instruction LET count=count+1 simply adds 1 to the contents of the variable 'count' and stores the answer back in 'count'. In other words, it increases the number stored in 'count' by 1. If you repeat this operation by using a GOTO you have something more than adding 1 to a variable - you have a program that counts! Try the following:

```
10 LET count=0
20 LET count=count+1
30 PRINT count
40 GOTO 20
```

You will see the screen fill with the numbers 1 to 21 and then the usual "Scroll? y/n" question. By answering y to this you can keep the numbers coming for a very long time.

Notice that by adding a GOTO to an instruction that adds one to a variable we actually seem to produce a program that does a bit more than just count. In fact, we generate a *sequence* of numbers. This is

much more the *flavour* of real programming than the one-after-the-other programs in Chapter Three. To see looping doing something a little more useful try:

```

1Ø LET count=Ø
2Ø PRINT "x= ";count;" x squared= ";count*count
3Ø LET count=count+1
4Ø GOTO 2Ø
    
```

This program will print out two lists of numbers, the second being the square of the first.

Even if you are very familiar with the idea of a loop you can be confused about what the current value of a variable is at any point in the loop. For example, in the case of the counting loop program, the first value of 'count' that was printed was one but in the squares program the first value was zero. This difference is simply due to *where* in the program the line that adds one to 'count' is placed and *also* what value 'count' is set to before the loop starts. If you change line 1Ø in the squares program to read 1Ø LET count=1 then the first value to be printed will be one.

Understanding what goes on in a loop gets easier with practise but it's all a matter of following through the action of the program clearly and without rushing.

Choice and conditions – the IF statement

Although the GOTO is a very useful statement the only thing that you can do with it is to form infinite loops. This is fine for simple things like printing tables of values but it is a bit too crude for many applications. What we are lacking is a statement that will stop the loop when a *condition* is satisfied. For example, suppose we want to write a program that will add ten numbers together and then print out the answer. At the moment the best that we can do is to type in each number in turn and add it to a *running total* which we print out each time, as in the following example:

```

1Ø LET total=Ø
2Ø LET count=Ø
3Ø INPUT "number = ";number
4Ø LET total=total+number
5Ø LET count=count + 1
6Ø PRINT count;" total=";total
7Ø GOTO 3Ø
    
```

To stop this program type STOP when it asks for the next number.

If you run this program you will find that it produces rather a lot of output that you don't need. What we really want to do is read in the ten numbers, keep a 'running' sum and only print out the answer at the end. This can be achieved using the IF statement:

```

10 LET total=0
20 LET count=0
30 INPUT "number = ";number
40 LET total=total+number
50 LET count=count+1
60 IF count=10 THEN GOTO 80
70 GOTO 30
80 PRINT count;" total =";total

```

The only difference between this and the first program to add ten numbers together is the use of the IF statement in line 60. Each time through the loop formed by the lines from 30 to 70 the IF statement is obeyed. This takes the form of comparing the contents of the variable 'count' to 10. If it isn't equal to 10, control passes to the next statement, i.e. line 70, and the GOTO following the THEN has no effect. However if 'count' is equal to 10 the GOTO following the THEN is carried out and control passes to line 80 printing the answer and ending the loop.

There are many ways of using the IF ... THEN GOTO statement to alter the flow of control depending on whether or not a *condition* is true. Before we can go on to investigate the sort of thing that can be done with IF we have to find out what types of conditions we can use.

All of the conditions that you can use in an IF statement take a very simple form:

'arithmetic expression1' 'relation' 'arithmetic expression2'

We already know what an *arithmetic expression* is so the only new element is the *relation*. In BASIC there are six relations:

<i>relation</i>	<i>meaning</i>
=	equals
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Notice that on the Spectrum each of these symbols is entered by a

single keypress. You will get an error if you enter a < and a > to make up a single <>! The meaning in BASIC of each of these relations is the same as their normal meaning.

Conditions are very often called *conditional expressions* and a condition is like an expression in that it evaluates to a value, but in its case there are only two possible results, true or false. It is important to read conditions in the right way to avoid confusion. For example, the condition 'count=3' is *not* an instruction to make count equal to 3, it is a *question* about what is stored in 'count'. If the value stored in 'count' is 3 then 'count=3' is true, but if the value is anything other than 3 then 'count=3' is false. Thus the Spectrum uses the sign = in two different ways – as an instruction to store a value in a variable and as a relation in a condition. The only way to tell which is the correct meaning in any case is to look at the rest of the instruction. Some examples of conditions are:

count<>4	false if count is 4, otherwise true
count*2>10	true if count*2 is greater than 10, otherwise false
3>1	ALWAYS true
1>6	ALWAYS false

To reinforce the idea that a condition is an expression that evaluates to one of two values (true or false) it is worth saying that the Spectrum represents both values as numbers. True is represented by 1 and false is represented as 0. In other words, if a condition is true it evaluates to 1 and if it is false it evaluates to 0. To prove that this is the case try the following program:

```

10 INPUT "first number";a
20 INPUT "second number";b
30 PRINT "the result of ";a;">";b;" is ";a>b
40 GOTO 10

```

You will find that either a 1 or a 0 will be printed depending on whether 'a>b' is true or false in the case of the numbers you typed. You might be surprised that you can write a condition in a PRINT statement where you would normally write an arithmetic expression. This is simply another reflection of the fact that a condition is an expression just like an arithmetic expression and it can be used anywhere that an arithmetic expression can. Indeed you can mix conditional and arithmetic expressions with no problems as long as you pay attention to the order in which they are evaluated. (Relations have priority 4 which means that any arithmetic is done before they are evaluated.) For example:

44 *The Spectrum Programmer*

```
PRINT (1>2)+(3>1)+(3=2+1)
```

will print 2 on the screen because (1>2) is false and evaluates to 0, (3>1) is true and evaluates to 1 and (3=2+1) is also true and evaluates to 1, which gives 0+1+1 i.e. 2. (Notice that the conditions all have to be enclosed in brackets to stop the Spectrum trying to do the arithmetic first!) This sort of thing is fairly advanced BASIC so don't worry if you don't understand it fully – what is important is that you understand that something like 2=3 is an expression and evaluates to true or false rather than an instruction to do some operation.

We have taken a slight detour around the subject of the IF statement to examine the idea of a conditional expression but armed with this new information the remainder of this chapter should seem easier. The general form of the IF ... THEN GOTO statement is:

```
IF 'conditional expression' THEN GOTO 'line number'
```

If the conditional expression evaluates to 1, or true, then the GOTO following the THEN is obeyed. If the conditional expression evaluates to 0, or false, then the statement following the IF is obeyed.

There are so many ways of using the IF ... THEN GOTO apart from breaking out of loops, that it is difficult to give examples of everything. But if you understand the ideas of flow of control and the way that IF and GOTO can be used to change it then you should have no trouble in understanding the examples in the rest of this book.

Using IF

The only way to find out how useful IF can be is to write your own programs that use it. In this way you'll slowly pick up all the standard ways that you can change the flow of control depending on the result of a conditional expression. However, to speed up this process and avoid clumsy ways of using the IF statement, it might help to give examples of some of the most common ways that it is used.

An IF statement can be used to *skip* a section of program

```
1Ø INPUT a
2Ø IF a>Ø THEN GOTO 4Ø
3Ø LET a=-a
4Ø PRINT a
```

This short program refuses to let you enter a negative! The IF statement in line 20 checks to see if the number in 'a' is greater than zero and if so control passes to line 40 – effectively skipping line 30. If 'a>0' is false line 30 changes the sign of the contents of 'a'. In another program a different list of statements might be skipped according to some other condition. The *shape* of the skip is depicted in Fig. 4.3 which illustrates how the result of the condition decides whether or not the list of instructions is carried out or skipped.

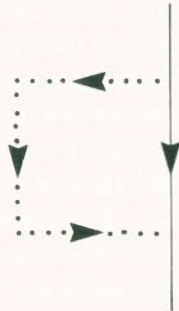


Fig. 4.3.

An extension of the idea of skipping some lines of BASIC is to choose between two different lists of commands. In this case it is easier to understand the idea after looking at the shape of the flow of control (see Fig. 4.4).

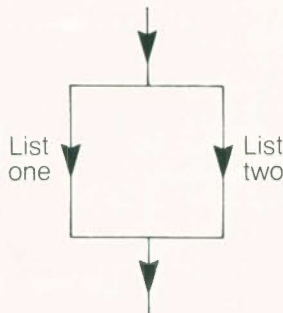


Fig. 4.4.

Which list is carried out depends on the condition used in the IF statement. The reason why it is useful to look at Fig. 4.4 before looking at an example of an IF statement to *select* between two sets of instructions is that it is a little difficult to see the simple division into two in the BASIC. Consider the following program:

```

10 INPUT a
20 IF a<0 THEN GOTO 60
30 PRINT "a is positive"
40 LET b=a
50 GOTO 80
60 PRINT "a is negative"
70 LET b=-a
80 PRINT a,b
    
```

According to the value of 'a' either lines 30, 40 and 50 are obeyed or lines 60 and 70. The *division* point in the diagram corresponds to the IF statement itself (line 20). The *join up* point is line 80 because this is the first statement that will be carried out no matter which of the two lists of instructions is selected. If you try to superimpose Fig. 4.4 on the program you will see that, although it represents what happens, it is difficult to fit it. The reason for this is that it is impossible in BASIC to write the two alternative lists next to each other, so the flow of control diagram is more like the one in Fig. 4.5.

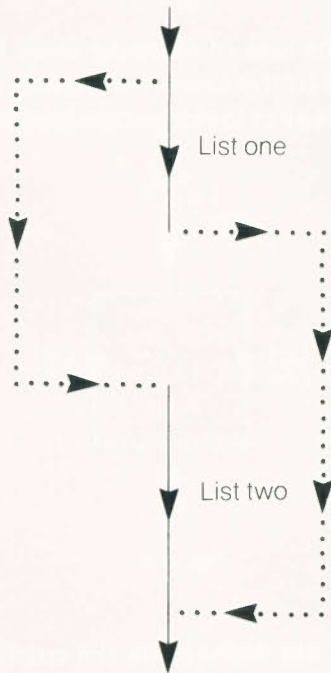


Fig. 4.5.

If you look at Fig. 4.5 you should be able to see that it is a 'mangled' form of Fig. 4.4 Using the IF to select between two

alternatives is easier to understand from Fig. 4.4 but Fig. 4.5 corresponds more closely to reality.

We have already met the only other important use of the IF statement, that of breaking out of a loop. The flow of control diagram for that circumstance can be visualised as shown in Fig. 4.6.

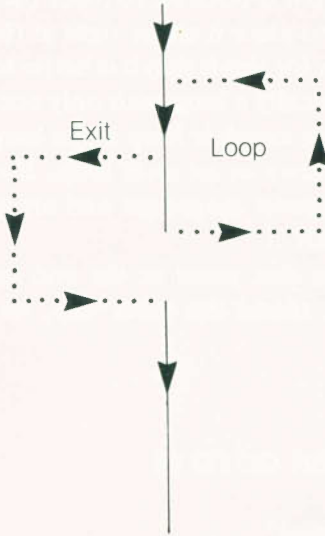


Fig. 4.6.

You should be able to see the familiar shape of an infinite loop. The path that leads out of the loop and back to the normal flow of control corresponds to a GOTO taken when the condition in the IF statement is true. Notice that the point at which the IF statement breaks out of the loop can be placed anywhere. In other words, you can break out of a loop anywhere from the first statement to the last.

For example:

```

10 LET a=0
20 LET a=a+1
30 IF a=20 THEN GOTO 60
40 PRINT a
50 GOTO 20
60 PRINT "finished"
    
```

has its exit point in the middle and


```

10 LET a=0
20 LET a=a+1
30 PRINT a
40 IF a=20 THEN GOTO 60
50 GOTO 20
60 PRINT "finished"

```

has its exit point at the end of the loop.

In general, although it is possible to place the exit point of a loop anywhere, it is better to place it either right at the beginning or right at the end. The reason for this is that it is better to avoid carrying out *part* of a loop. (Technically a loop with only one exit point placed at the beginning is known as *while* loop and a loop with only one exit point at the end is known as an *until* loop. However, these names come from other computer languages and are not important when programming in BASIC.)

A loop that has its exit point at the end can be simplified by turning the condition round the other way, for example:

```

10 LET a=0
20 LET a=a+1
30 PRINT a
40 IF a=20 THEN GOTO 60
50 GOTO 20
60 PRINT "finished"

```

can be turned into

```

10 LET a=0
20 LET a=a+1
30 PRINT a
40 IF a<>20 THEN GOTO 20
50 PRINT "finished"

```

The loop in the first example comes to an end when 'a' is equal to 20 and line 50 contains the GOTO that otherwise makes the loop continue. In the second example the GOTO has been eliminated by turning the condition around to make the loop continue when 'a' is *not* equal to 20. After seeing this sort of thing a few times you will adopt the shorter, neater form without thinking, but it is worth occasionally remembering where it derives from.

As stated at the beginning of this section, there are many ways of using an IF statement to alter the flow of control of a program and you shouldn't feel restricted to just those that have been introduced in this section. However, it is wise not to experiment too much

because the more ways that you use the IF statement the more difficult it will be to understand your program. In essence the rule is to make the flow of control diagram as simple as possible so that your programs will be easy to read, understand and debug.

The FOR statement

Apart from the position of the exit point, loops differ in two other ways. All loops continue until a condition is satisfied but in many cases this is equivalent to carrying out the loop a fixed number of times. For example, if you wanted to print the word “Hello” on the screen five times you could do it using:

```
1Ø LET a=1
2Ø PRINT “Hello”
3Ø LET a=a+1
4Ø IF a<=5 THEN GOTO 2Ø
```

However, this is such a common situation that BASIC provides two extra statements – FOR and NEXT – to make repeating lists of statements easier. Using FOR and NEXT the program that prints “Hello” on the screen five times can be written:

```
1Ø FOR a=1 TO 5
2Ø PRINT “Hello”
3Ø NEXT a
```

The meaning of the FOR and NEXT should be clear from the program. The variable ‘a’ is used to count the number of times that the loop has been obeyed in the same way as in the earlier example. The difference is that everything is done automatically. The FOR statement first sets ‘a’ to one. Each time the NEXT statement is carried out one is added to ‘a’ and, as long as its value hasn’t exceeded 5, control is transferred back to line 2Ø (i.e. there is an implied GOTO 2Ø). The result is that the PRINT statement at line 2Ø is executed five times before control passes on to the statement following the NEXT.

The general form of the FOR ... NEXT loop is:

```
FOR ‘index variable’=‘start value’ TO ‘end value’
. . . . .
NEXT ‘index variable’
```

The 'index variable' is initially set to the 'start value'. Each time NEXT is reached the 'index variable' is increased by one and as long as the value hasn't exceeded the 'end value' control is transferred to the statement just after the FOR. The only restriction is that the name of the 'index variable' can only be a single letter – so 'index variable' can be any one of 'a' through to 'z'. To make sure that you understand exactly what a FOR loop can do try the following examples:

```

10 FOR i=1 TO 10
20 PRINT i
30 NEXT i

10 FOR z=100 TO 110
20 PRINT z
30 NEXT z

```

In general both the 'start value' and the 'end value' can be full arithmetic expressions but it is important to realise that these are only evaluated *once* at the start of the loop. This becomes clear if you think of the FOR statement as only being carried out once at the start of the loop. The value of the 'index variable' can be used in arithmetic expressions during the loop but *its value must not be changed*. In other words, you can use the 'index variable' in a LET statement on the right hand side of an = sign but not on the left. For example, the following short program will print a multiplication table

```

10 INPUT "Which table (e.g. enter 2 for two times table) ?";t
20 INPUT "Starting at ?";s
30 INPUT "Ending at ?";e
40 FOR i=s TO e
50 PRINT i;" x ";t;" = ";i*t
60 NEXT i

```

Notice that both the start and end values of the FOR loop are arithmetic expressions, simple variables in fact! Also note the use of the 'index variable' 'i' in line 50.

The simple FOR loop serves for most purposes, however there is a slightly more advanced form that is occasionally useful and is certainly worth knowing about. In the simple FOR loop each time through the loop one was added to the value of the index variable. This is sensible if you are using the index variable to count the number of times that the loop has been carried out. However, it is

sometimes the case that a calculation carried out inside the loop needs a value that changes by something other than one each time through the loop. This is catered for by the addition of the BASIC statement STEP, the general form of which is:

```
FOR 'index variable'='start value' TO 'end value' STEP
    'increment'
```

The 'increment' specified following STEP is the amount that is added to the index variable each time through the loop. So the simple FOR loop is equivalent to STEP 1. The only thing that you have to be careful of when using FOR STEP is to make sure you know when the loop will come to an end. The rule is that the loop terminates when the value of the index *exceeds* the 'finish' value. So the value of an index variable in a FOR loop can never become larger than the 'finish value'. To see this in action try the following examples:

```
1Ø FOR a=.4 TO 1Ø STEP .Ø1
2Ø PRINT a
3Ø NEXT a
```

or

```
1Ø FOR a=1 TO 1ØØ STEP 25
2Ø PRINT a
3Ø NEXT a
```

The value of the *increment* following STEP can be negative, in which case it is better called a *decrement*. For example:

```
1Ø FOR a=1ØØ TO Ø STEP -15
2Ø PRINT a
3Ø NEXT a
```

You may have some slight difficulty in working out when this and similar loops end. However, the rule is almost the same as for a positive increment. Each time through the loop the value of the index variable decreases by the increment and the loop ends when its value first drops below the 'end' value.

As well as having a negative increment, it is possible for either the 'start' or the 'end' value to be negative and this is where things can be confusing. Consider this short program:

```
1Ø FOR a=-1ØØ TO 5Ø STEP 1Ø
2Ø PRINT a
3Ø NEXT a
```

At first sight it may not seem to make sense. However, if you keep a cool head then you should be able to see that the same rules apply. The value of the increment is added to the index variable each time through the loop until its value exceeds the 'end' value if the increment is positive, or is less than the 'end' value, if the increment is negative.

Using the FOR loop

There are one or two rules that govern the use of FOR loops. As you can use any valid BASIC statement within a FOR loop it is possible to use a GOTO or an IF to leave a FOR loop before it is finished (i.e. before the index variable has reached the 'end' value). This is quite permissible in Spectrum BASIC but many other versions of BASIC will complain if you leave FOR loops in an unfinished state. Because of this it is better not to fall into the habit of using *sloppy* FOR loops.

It is often the case that FOR loops are used in combination. This is easy to understand as long as you think logically about the way each loop works. For example:

```
10 FOR a=1 TO 10
20 FOR b=1 TO 10
30 PRINT a,b
40 NEXT b
50 NEXT a
```

is correct because the FOR loop formed by lines 20, 30 and 40 is completely contained within the FOR loop starting at line 10 and ending at line 50. This means that the *inner* loop is carried out each time through the *outer* loop. It is all too easy to make a slight mistake and end up with:

```
10 FOR a=1 TO 10
20 FOR b=1 TO 10
30 PRINT a, b
40 NEXT a
50 NEXT b
```

which will give you an error message.

IF ... THEN and the colon

There is an even more general version of the IF statement than the one we have examined so far. As well as being able to follow the

THEN by the BASIC command GOTO, you can in fact use any valid BASIC command. For example:

```
10 INPUT a
20 IF a<0 THEN PRINT "a is negative"
30 IF a=0 THEN PRINT "a is zero"
40 IF a>0 THEN PRINT "a is positive"
50 GOTO 10
```

The best way to think of this is as a sort of easier version of the IF to skip an instruction. Only in this case the instruction is only carried out if the condition is true. There isn't very much to add to this description except to emphasise the fact that you can follow THEN by any valid BASIC statement – including another IF!

This extended version of the IF statement seems very useful at first but you quickly come to the conclusion that it's not often that you want to execute a *single* statement as a result of some condition. Clearly what is needed is some way of writing a list of BASIC statements following the THEN. Spectrum BASIC does provide a way of doing this although it is important to realise that not all versions of BASIC do. You can group together a number of BASIC statements on a single line, separating each one by a colon and they will be obeyed in turn from left to right. For example:

```
10 PRINT "first";LET a=1;PRINT "second";a
```

will be carried out as a single line of BASIC working from left to right and is equivalent to:

```
10 PRINT "first"
20 LET a=1
30 PRINT "second";a
```

This use of colons to group a number of BASIC statements together effectively extends the rule for the default flow of control. Now instead of just obeying instructions in order of increasing line number (i.e. effectively moving down the screen) instructions that are grouped together on a line are obeyed from left to right. This is exactly the same way that a human would read a list of instructions from a sheet of paper (i.e. left to right and then downwards). Although this extension can make life much easier it should be remembered that, as other versions of BASIC do not have this facility, over-using it may make you very dependent on the Spectrum's version of BASIC.

As mentioned earlier, the place where it is really useful to have the

ability to put more than one BASIC instruction on to a line is following a THEN. For example, at the end of most games programs it is usual to ask if the player would like another game. In the following example of a routine that could be added to the end of a program the answer is expected in the form of 1, to mean yes, or 0, to mean no. Any other value as a response will be taken to be a mistake and a message to that effect will then be printed and the question asked again.

```

110 INPUT "do you want another game (yes=1,no=0)";a
120 IF a=1 THEN GOTO xxx
130 IF a<>0 THEN PRINT"you must answer 1 or
    0":GOTO 110
140 PRINT "bye!!"

```

Notice the use of the colon to group two BASIC commands together in line 130. The xxx in line 120 should of course be replaced by the line number of the start of the program.

There is a simple BASIC statement that we have not discussed so far that is very useful when used in conjunction with the IF statement. Suppose that as a result of some condition you should want the program to stop, then currently the only way that we know of achieving this is to GOTO a line number that is at the end of the program. The BASIC statement STOP can be used anywhere in a program to return control to the user. For example:

```

10 PRINT "Do you want to continue Y=1,N=0"
20 INPUT a
30 IF a=0 THEN PRINT"bye bye":STOP
40 IF a<>1 THEN PRINT "I don't understand - you must
    enter either 1 or 0":GOTO 10
50 PRINT "I continue - master"
60 GOTO 10

```

Notice the use of the colon to group commands together in lines 30 and 40 and the use of STOP in line 30.

A final example

As a further example of both the IF and FOR statements consider the problem of turning the stopping distance program given in Chapter Three into a sort of quiz. For a range of speeds the player is asked for the thinking, braking and total stopping distance in feet.

```

10 PRINT "Stopping distance"
20 LET mark=0
30 FOR s=10 TO 80 STEP 10
40 PRINT "What is the thinking distance at ";s;" mph ?"
50 INPUT t
60 IF t=s THEN LET mark=mark+1
70 PRINT "What is the braking distance at ";s;" mph ?"
80 INPUT b
90 IF b=s*s/20 THEN LET mark=mark+1
100 PRINT "What is the total stopping distance ?"
110 INPUT d
120 IF d=s*s/20+s THEN LET mark=mark+1
130 NEXT s
140 PRINT
150 PRINT "You scored ";mark
160 PRINT "out of a possible 24"
    
```

The flow of control summarised

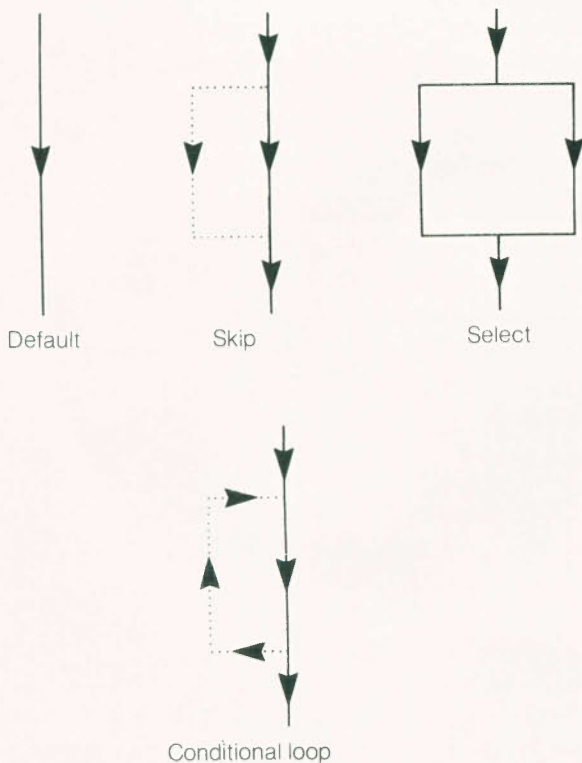


Fig. 4.7. Flow of control diagrams

It is difficult to say exactly what goes on in the mind of an experienced programmer. Whatever it is, it is a process that marks the difference between a beginner and an expert. One thing that does seem certain is that, in addition to the catalogue of programs that have been seen before and a collection of handy tricks, the standard forms of the flow of control diagrams that we have been studying in this chapter are ever-present. It is worth mentioning at this point that it has been proved that you can write any program using only the default, select and conditional loop. This is so important that it is worth gathering together the flow of control diagrams so that you too can store them away in your personal memory.

Chapter Five

Handling Text and Numbers

So far the only programs that we have written have used numbers. If this was all that computers could do they would be little different from pocket calculators! In this chapter we will look at how the Spectrum can handle characters and text just as easily as digits and numbers. In the last part of this chapter some other ways of extending the things we can do with data are introduced – arrays and tape storage.

Strings

In Chapter Three the idea of a *string* – a collection of letters within double quotes – was introduced as a way of printing messages and prompts. In fact what we have been calling a string is really a *string constant*. The use of the word constant might alert you to the fact that there are such things as *string variables*. A string variable is similar to a simple variable in that it is a named area of memory that can be used to store information. In this case, the information is a collection of characters instead of a number. The rules for naming string variables are different from simple variables. A string variable's name consists of a letter followed by a dollar sign. The dollar sign is used to distinguish between a simple variable and a string variable (e.g. 'b' is a simple variable but 'b\$' is a string variable and therefore different). The LET statement can be used to store a string constant in a string variable and the PRINT statement can be used to print its contents.

```
10 LET a$="this is a string"  
20 PRINT a$
```

In fact, a string variable can be used anywhere that a simple variable can as long as it makes sense. For example, you can use INPUT a\$ to

store a string typed in from the keyboard while a program is running but `LET total=3+a$` is obviously nonsense (you cannot add a string to a number!). Notice in particular the difference between:

```
10 LET a$="1"
```

and

```
20 LET a$=1
```

Line 10 is fine because the 1 is enclosed in double quotes and is therefore a string but line 20 will give an error message because a\$ is a string variable and 1 is a number.

The introduction of string variables is exciting because it opens up the possibility of the Spectrum handling text and even dialogues. So far, however, the only sort of program that we can write is:

```
10 INPUT "What is your name?";n$
20 PRINT "Hello ";n$," I am your Spectrum computer"
```

which is all right for a start but it can result in unnatural dialogues like:

```
What is your name? Fred Bloggs
Hello Fred Bloggs, I am your Spectrum computer
```

The trouble is that although we can INPUT, PRINT and store strings, we have no way of changing them. This is rather like being able to INPUT, PRINT and store numbers but having no way of doing arithmetic – it's obvious that this would limit the programs that we could write! The answer lies in inventing an 'arithmetic' for strings so that as well as having arithmetic expressions we can use 'string expressions'.

String expressions

Before introducing the Spectrum's facilities for handling strings it is worth considering what sort of things you might like to do and then see if the Spectrum can actually fit the bill. If a program had someone's first name in f\$ and their last name in l\$ then it would be useful to be able to *join* them together to form one longer string. Such joining together of strings is known as *concatenation*. Another thing that would be useful is the ability to *extract* part of a string. For example, you could extract the last name from a string consisting of an initial and surname i.e. extract "BLOGGS" from

“F.BLOGGS”. A string that is part of another string is often called a *substring*. For example, BLOGGS is a substring of F.BLOGGS. Another useful facility would be to replace one substring with another. For example, if we were trying to keep F.BLOGGS a secret we might want to replace the surname with asterisks giving “F.*****”. Finally it would be a great advantage to be able to test for the presence of a particular substring within a string, for example, to see if the substring “BLOGGS” occurred in the name stored in n\$. To recap, the string operations that we would like to find are concatenation, substring extraction, substring replacement and substring searching.

Our first requirement, string concatenation, is immediately satisfied by the Spectrum’s concatenation sign +. If a\$ contains the string “abcd” and b\$ contains “efgh” then after:

```
LET c$=a$+b$
```

c\$ contains “abcdefgh”. Notice that we now have two uses for the symbol +, as the sign for addition and as the sign for concatenation. You can use the + sign more than once in a string expression:

```
LET c$=“Mr ”+f$+“ ”+l$
```

will join up the four strings involved and if f\$ contains a first name “Fred” and l\$ a last name “Bloggs” then c\$ will contain “Mr Fred Bloggs”. Notice the use of a single space between the two strings to avoid the result being “Mr FredBloggs”!

Extracting or changing a substring can be done by using a single new operation, *slicing*. Most other versions of BASIC use methods that are much more complicated than the Spectrum’s *string slicing*. Although the Spectrum may be on its own when it comes to handling strings, in fact it offers a distinct improvement over other BASICs. A string slicer specifies a substring by giving the position of the first and last letters. For example:

```
PRINT “12345678” (3 TO 6)
```

displays the substring “3456” i.e. starting at the third character and ending at the sixth. Remember that TO must be entered by a single keystroke (it is to be found on the ‘F’ keypad). Typing the letter ‘T’ and the letter ‘O’ just won’t do! The general form of a slicer is:

```
‘String (‘arithmetic TO ‘arithmetic )  
expression’ (expression1’ TO expression2’ )
```

and the substring that the slicer specifies starts at the character given

by 'arithmetic expression 1' and ends at the character given by 'arithmetic expression 2' in the string that 'string expression' evaluates to. This may seem like a complicated definition, and indeed it does go further than the most common forms of the slicing notation found in other BASICs. For most of the time the 'string expression' is either a constant or a single string variable and the arithmetic expressions are again simply constant numbers or single variables. For example:

```
"abcd"(2 TO 3)
```

or

```
"abcd"(count TO 3)
```

However, by now you may have realised that one of the most powerful principles in BASIC is that anywhere you can use a constant or a variable you are also allowed to use an expression. String slicing is no different in this respect and you can write things like:

```
("abcd"+"efgh")(start TO start+3)
```

which first concatenates the two strings "abcd" and "efgh" to form the single string "abcdefgh" and then extracts four letters, starting with the character at the position stored in 'start'. You shouldn't be frightened to write complicated string expressions any more than you would be over complicated arithmetic expressions.

There are a number of special cases of the slicing notation that are well worth knowing about. The start and end of a string are so often used in forming substrings by slicing that if you leave 'arithmetic expression 1' out the start of the string is assumed and if you leave 'arithmetic expression 2' out the end of the string is assumed. So:

```
"123456"( TO 5) means "123456"(1 TO 5) which is "12345"
```

```
"123456"(3 TO ) means "123456"(3 TO 6) which is "3456"
```

```
"123456"( TO ) means "123456"(1 TO 6) which is "123456"
```

There is also a very useful abbreviation that will extract a single character at any position in a string. Instead of having to write (n TO n) to extract the character at position n in the string you can simply write (n). So:

```
"123456"(3) means "123456"(3 TO 3) which is 3
```

It is possible to get things wrong when using slicing. For example, if you specify a character position that doesn't exist you will get an

error message saying “Subscript wrong” or “Integer out of range”. For example, “123456”(4 TO 7) or “123456”(−1 TO 3) will both produce error messages. However, if you use a starting position that is less than the final position, e.g. “123456”(3 TO 1) you might be surprised to discover that you do not get an error message. Instead the answer is a very special form of string – the *null string*. The null string has no characters in it and plays a very similar role in string expressions to that of zero in arithmetic expressions. The string constant that corresponds to the null string is written “” i.e. a pair of double quotes with nothing in between. Notice the difference between “” and “ ”. The first is the null string and has no characters in it, the second is a string consisting of one character – a blank or space. From the point of view of a computer a space is just as much a character as a letter of the alphabet, it takes up one position to print and needs just as much computer memory to store it. If you print a null string it has no effect whatsoever. The statements PRINT a;b and PRINT a;””;b produce the same result.

As an example of using slicing consider the problem of printing the name of a month given its number (i.e. you should print Dec for 12 and May for 5 etc.). Try the following short program:

```

1Ø LET y$="JanFebMarAprMayJunJulAugSepOctNovDec"
2Ø INPUT "Month number ?";mon
3Ø PRINT "month ";mon;" is ";y$(1+3*(mon-1) TO 3*mon)
4Ø GOTO 2Ø

```

If you enter in the range 1 to 12 the program will print the correct abbreviation for the month in question. The way that it works is by *slicing out* the three letter name from the long string y\$. The best way of understanding the slicer in line 3Ø is by working it out by hand for a few values of ‘mon’. if ‘mon’ is 4 then (1+3*(mon−1)) is 10 and 3*mon is 12 and (10 TO 12) specifies the three letters “Apr”.

Slicing can not only be used to extract a substring but also to change all the characters in the substring. You can use the standard slicing notation to define a substring to be changed on the left hand side of the = in a LET statement. For example:

```

1Ø LET a$="123456"
2Ø LET a$(3 TO 4)="ABCDEFG"
3Ø PRINT a$

```

will print ‘12AB56’ on the screen. This feature is a very logical extension of the normal use of LET to store a string in a string variable. The slicer simply restricts the range of the characters that

are altered by the LET. If the string on the right hand side of the = in the LET is bigger than the substring specified the extra characters are ignored and if it is shorter then it is *padded* with blanks. For example:

```
1Ø LET a$="123456"
2Ø LET a$(2 TO 5)="ABC"
3Ø PRINT a$
```

will print the string "1ABC 6" on the screen. (Notice the blank following the letter C.)

The only thing left from our initial list of string handling requirements is testing to see if a particular substring is present in another string. Spectrum BASIC doesn't provide a direct method of achieving this but it does extend the use of conditional expressions (see Chapter Four) to strings and this can be used to achieve the same ends. You can use all of the relations that were introduced in that chapter with strings. The meaning of = and <> are easy enough to understand. Namely, two strings are equal if they are of the same length and contain the same characters in the same order otherwise they are not equal. However, what do the relations <, >, <= and >= mean when applied to strings? The answer to this question varies from BASIC to BASIC. In the case of the Spectrum, however, they are defined so that a\$<b\$ is true if the string in a\$ would come before the string in b\$ in an alphabetically ordered list of strings. The trouble is that we are all so familiar with alphabetically ordered lists that we tend to forget how they work! If the two strings being compared are single letters, then a\$<b\$ if the letter in a\$ comes earlier in the alphabet than the letter in b\$. For example, "a"<"b" is true but "d"<"b" is false. What about comparing strings that contain single characters that are not necessarily letters, for example what do we make of "*"<"\$"? In the case of the letters, the alphabet provided us with a readymade order so what we need is to extend this order to include all the other symbols that the Spectrum can use. In other words we need a *super alphabet*! This is already available for the Spectrum. If you look at Appendix A in the *Spectrum Manual* you will see a listing of the complete set of Spectrum characters in a predefined order and it is this that is used as the super alphabet to decide if a\$<b\$. If you don't have the *Spectrum Manual* to hand, or if you are just interested, you can print all the characters in their proper order by using the following program:

```

1Ø FOR i=34 TO 255
2Ø PRINT "character ";i;" = ";CHR$(i)
3Ø NEXT i

```

Line 1Ø specifies 34 as the starting point as the characters before that are unprintable. Don't worry about the use of CHR\$ in line 2Ø this will be explained later. If you run this program you might be surprised to see words like PRINT and FOR appearing on the screen as characters! This is simply a reflection of the fact that the Spectrum treats everything that can be entered as a single keystroke as a single character even if it appears on the screen as a word! Coming back to the question of whether or not "*" < "\$" is true or false, "*" is character 42 and "\$" is character 36 so "\$" comes before "*" in the order and "*" < "\$" is false. You can decide the truth or otherwise of any relationship in the same way.

If the two strings contain more than one character they are compared one character at a time until the first pair of different characters is found. The relationship between the two strings is then decided on the basis of those two characters. For example, ABCD < AZCD is true because the first pair of letters that are different is B and Z and B < Z is true. If one of the strings is the same as the other apart from the addition of a few extra characters then the comparison is based on length, i.e. ABCD < ABCDEF, because there is no pair of letters that is different and ABCD is shorter than ABCDEF.

The idea of the length of a string is quite important and so the Spectrum provides a very simple way of finding the length of any string. Try the following program:

```

1Ø INPUT a$
2Ø PRINT LEN(a$)
3Ø GOTO 1Ø

```

you will see that it prints the number of characters that you type in response to the INPUT. In general:

```
LEN('string expression')
```

will give the number of characters in the 'string expression'. (The use of LEN will be discussed in more detail in Chapter Six.)

Arrays

Strings and numbers are the only two types of data that the

then you will get an error message for trying to use something that doesn't exist! It is tempting to think that it is better to define arrays larger than you need to try to avoid such error messages but be warned, arrays can quickly use up all the memory that your machine has to offer! There is a similar restriction on the names of arrays as on index variables i.e. they can only be one letter long! This means that you can only have 26 different arrays called 'a' to 'z', but this is usually more than enough. Using the DIM statement destroys any arrays that already exist with the same name and creates a brand new array initialised so that each element stores zero.

In addition to being able to define arrays that can be thought of as *rows* of variables (see Fig. 5.1) you can define arrays that correspond to organising variables into tables made up of rows and columns. For example:

```
DIM a(10,10)
```

defines a collection of variables organised into 10 rows and 10 columns. A particular element of this array can be referred to as $a(i,j)$ where the two indices select the row and column.

The idea of a two-dimensional array can be extended to three-, four- ... up to 255-dimensional arrays. It is difficult to think of an arrangement of variables that corresponds to the higher-dimensional arrays but they are defined and used in roughly the same way as one- and two-dimensional arrays. For example:

```
DIM a(10,20,5)
```

is a three-dimensional array and typical element is $a(2,1,4)$. There is not much use for arrays with dimensions greater than two. This is fortunate because they tend to use up memory very, very fast.

You can form arrays from string variables as well as numbers and these can be used to store and manipulate lists of words. There is one complication, however, in that in some senses a string is already a one-dimensional array of characters. If you define a one-dimensional string array, in fact what you get is a string of fixed length. For example:

```
DIM s$(10)
```

is a string variable that always stores 10 characters no matter what (even if some are blank). To see the difference between the string $r$$ and the string array $s$$ try the following program:

```

10 DIM s$(10)
20 LET s$="abc"
30 LET r$="abc"
40 PRINT s$;"x";r$;"x"

```

You should see that s\$ is ten characters long no matter what you store in it. Notice that for string arrays you can store values in more than one element at a time. You can also store single characters in elements, however. The following lines store z in s\$(5).

```

50 LET s$(5)="z"
60 PRINT s$

```

You might be able to see a similarity between this and the slicing notation introduced earlier.

You can handle lists of words by using two-dimensional string arrays. For example, the number reversing program can be used to reverse a list of words:

```

10 DIM a$(5,10)
20 FOR i=1 TO 5
30 INPUT a$(i)
40 NEXT i
50 FOR i=5 TO 1 STEP -1
60 PRINT a$(i)
70 NEXT i

```

Notice that if you leave out the last index in a two- (or more) dimensional string array this is taken to mean that you want to treat the array as a collection of strings. For example:

a\$(5,2) is a single character

but

a\$(5) is a string of 10 characters.

You can even use the slicing notation to pick out substrings. For example:

a\$(5,1 TO 2)

is a substring starting at character 1 and ending at character 2.

Sinclair BASIC allows very flexible uses of strings and arrays. These can be difficult to understand at first but once you have become accustomed to them you will find that they are very powerful.

A word game

As an example of using string arrays consider the problem of writing a program to play the game of hangman. Because the computer cannot 'think up' a list of words for you to guess, it is necessary to ask someone else to type in a list for you to guess. Once the list of words has been entered the player must try to guess each word in turn letter by letter. Each letter that is entered must be checked against each letter in the word. If it is present then the letter in the word must be replaced by a blank to make sure that the player cannot guess it a second time. When all the letters have been guessed the program moves on to the next word, or if there are none, comes to an end.

After this description you should be able to make a good attempt at your own hangman program before looking at the one below:

```

10 DIM w$(5,10)
20 FOR i=1 TO 5
30 INPUT "Word = ";w$(i)
40 NEXT i
50 FOR i=1 TO 5
60 LET g=0
70 LET t$=w$(i)
80 LET g=g+1
90 INPUT "guess=";a$
100 LET f=0
110 FOR j=1 TO 10
120 IF a$(1)=w$(i,j) THEN PRINT"YES! - ";a$(1):LET
    w$(i,j)=" "
130 IF w$(i,j)<>" " THEN LET f=1
140 NEXT j
150 IF f=1 THEN GOTO 80
160 PRINT "You got it in ";g;" !!"
170 PRINT "The word was ";t$
180 NEXT i
190 PRINT "game over"

```

Line 10 defines the array w\$ so that it can hold five words of up to ten letters each. Lines 20 to 40 are used to input the words. The FOR loop starting at line 50 and ending at line 180 repeats the guessing part of the program five times, once for each word. The variable t\$ is used in line 70 to store the word until the end of the game so that the array element can be modified by storing blanks in the place of letters that have been guessed. The guess is input in line 90. Each

letter in the word is checked against the current guess by the FOR loop starting at line 110 to line 140. The variable 'f' is used to check that there are still letters left to be guessed.

Initialising variables - DATA and RESTORE

It often happens that a standard set of values needs to be stored in a set of variables or an array for a program to work properly. For example, suppose we want to print the number of days in a given month we could set up an array of 12 elements and STORE the answer in each element - i.e. the number of days in Jan would be stored in the first element of the array, the number in Feb in the second and so on. This is a very simple and useful idea but how do you initialise each element of the array to the correct value? You could use a FOR loop and an INPUT statement to read in the answers or you could write 12 LET statements. To make life slightly easier, BASIC provides the facility to store data within a program so that it can be transferred to any variable of your choice. The data is stored in a DATA statement that is composed simply of the word DATA followed by a list of values separated by commas. For example, the days in each month could be written as:

```
10 DATA 31,28,31,30,31,30,31,31,30,31,30,31
```

To transfer these data values into variables the READ statement is used. A READ statement is simply the keyword READ followed by a list of variables separated by commas. Each time a READ statement is encountered data values are transferred into variables in the variable list - one data value per variable. The best way to think of this is to imagine a pointer initially set to the first datum value in the DATA statement. Each time a READ statement transfers a datum value into a variable, the pointer is moved on to the next datum value. Whenever a READ statement is obeyed data is transferred, starting from whatever datum value it has reached after previous READ statements. So the array holding the 12 data values in the example DATA statement given above can be produced using:

```
20 DIM m(12)
30 FOR i=1 TO 12
40 READ m(i)
50 NEXT i
```

You can have as many DATA statements in a program as you like, anywhere that you like, and they are treated as if all the data that they store was contained in one big DATA statement. So when the pointer moves past the end of a DATA statement it moves to the beginning of the next DATA statement. If there isn't one, however, you'll get an error message.

There is no restriction on the types of data that you can store in a DATA statement (i.e. you can use strings or numbers) but you must always be careful to READ data into variables of the same type, i.e. strings into string variables and numbers into numeric variables. Strings in DATA statements have to have double quotes around them otherwise it wouldn't be possible to tell where one datum item ended and another began. As an example of using strings in DATA statements consider the following:

```

1Ø DATA "jan","feb","mar","apr","may","jun","jul","aug",
    "sep","oct","nov","dec"
2Ø DIM m$(12)
3Ø FOR i=1 TO 12
4Ø READ m$(i)
5Ø NEXT i

```

and compare it to the example given earlier.

Sometimes, especially in games, it would make things easier if we could alter the position of the imaginary pointer in data statements. This can be done using the RESTORE command. If you use RESTORE followed by a line number, then the next READ statement will start taking its data from the beginning of the next DATA statement following the line number specified (if there isn't one you will get an error message). If you just RESTORE without a line number, then the pointer is moved to the beginning of the first DATA statement in the program. For example try:

```

1Ø DATA 1,2,3,4
2Ø READ a,b,c,d
3Ø PRINT a,b,c,d
4Ø RESTORE
5Ø GOTO 2Ø

```

Saving data on tape

We have already seen that it is possible to SAVE and LOAD programs on tape. It would seem to be logical to extend the facility

to include SAVEing and LOADing data stored in variables to and from tape. In fact the Spectrum only allows SAVEing and LOADing of arrays but this is no disadvantage as any simple variable that you might want to save can be first transferred to free array elements and then the array saved. All you have to do to save an array on tape is to proceed exactly as you would for saving a program but use:

```
SAVE 'filename' DATA 'array name' ()
```

Where 'filename' is a string constant or variable that is the name of the file of data and 'array name' is the name of the array, numeric or string to be saved.(Notice the empty brackets at the end of the definition.) Similarly, to load a previously saved array use:

```
LOAD 'filename' DATA 'array name' ()
```

This will search the tape until a previously saved array is found with the correct file name. Any existing arrays with the same name are deleted and the saved version of the array is read in. Notice that both SAVE and LOAD can be used from within a program. You can also use:

```
VERIFY 'filename' DATA 'array name' ()
```

to check that data has been saved without error in the same way as for programs. As an example of saving and loading data try:

```
10 DIM a(10)
20 FOR i=1 TO 10
30 LET a(i)=i
40 NEXT i
50 PRINT "rewind tape and"
60 SAVE "test" DATA a()
70 PRINT "rewind tape and press play"
80 LOAD "test" DATA a()
90 PRINT "data loaded OK"
```

Apart from remembering to press the correct controls on the tape recorder, it's as easy as that.

Chapter Six

Functions and Subroutines

At this point in learning BASIC you should be in a position to see that the expression is the main way that programs *change* data. Without expressions – arithmetic, conditional and string – BASIC would be reduced to moving values from one place to another. It is only by the use of expressions that values can be combined and compared to produce new results. To make expressions even more useful, BASIC provides a large range of operations that can be used to make expressions, in the form of *functions*. You can also extend the range of functions by creating your own, *user-defined* functions. This creation of new operations can be taken one stage further in BASIC by using the GOSUB and RETURN statements to group statements together into *functional units* or *subroutines*.

In the first part of this chapter we will look at the general idea of a function and then move on to examine some of the more common functions available to the Spectrum programmer. The sections that deal with particular functions can be read very quickly or even skipped until you need to use them or until they are used in an example. However, don't skip the section on *special functions* because these are particularly important.

The idea of a function

Before dealing with the way the Spectrum handles functions, it is worth looking at functions in general. You may already be familiar with the idea of a function from mathematics. For example, $\sin(x)$ is a function. However, the idea of a function isn't really anything to do with advanced mathematics. At its most simple, a function is an operation on data that produces a *single* value as its result. For example, finding the larger of two numbers is an operation on data that returns a single value – the maximum of 3 and 42 is 42, the

maximum of 2 and 2 is 2 – and *maximum* is therefore a function.

The Spectrum doesn't have a maximum function but it is nevertheless a useful one to consider as an example because it is easy to understand and, as we will see later, it is easy to remedy the deficiency and program a maximum function of our own.

The standard way of writing a function involves writing its name to the left of the values on which it is to operate that are enclosed in brackets. In the case of finding the maximum of two numbers a sensible name for this function is 'max' and so the previous two examples can be written:

```
max(3,42)
and
max(2,2)
```

Following the usual BASIC convention that anywhere that you can use a constant or a variable you can use an expression, the following is also allowed:

```
max(count+3,total*20)
```

The data values that follow the name of the function are called *parameters*. It is possible for functions to have any number of parameters but all of the functions supplied on the Spectrum only have one.

Functions can be used in expressions just as if they were variables or constants. For example:

```
LET result=max(3.3,4.2)
```

would evaluate our function 'max' and store the answer (4.2) in 'result'. (Remember that the Spectrum doesn't have a 'max' function so don't try this example.) You can now see why the condition that a function should give only one result is so important. If a function gave more than one result, which one would be stored in the variable 'result' or which one would be used to evaluate the rest of the expression? As we want to use functions in expressions they *can* only return one answer. Sometimes it is possible to change something that isn't a function into a function by simply choosing one of the possible answers. For example, the Spectrum has a function SQR which gives the square root of a number. Now if you ask for the square root of four the answer is obviously two i.e. two times two is four, but it is all too easy to forget that minus two is also the square root of four. A minus times a minus is a positive and so $-2 * -2$ is 4 (not -4). The objection that the square root operation

isn't a function can be overcome by simply deciding that SQR will be a function that returns the positive square root of a number.

The Spectrum's functions

The Spectrum has a very wide range of functions and some of them are so specialised that it is better to deal with them in detail in other chapters. However, there is a *central core* of functions that you would expect to find in any BASIC and these will be explained in this chapter. A full list, with brief explanations, of all the Spectrum's functions can be found in Appendix C of the *Spectrum Manual*.

The *core* functions can be divided into three groups: the arithmetic functions such as SQR and ABS; the trigonometrical functions such as SIN and COS; and the string functions such as LEN and CHR\$. In addition there are a number of unclassifiable but very important *one-off* functions such as RND. All of the functions are entered by a single keystroke and the use of brackets around the parameters is optional. The most important thing is to have some idea of what functions are available, so a brief reading of the description of each function listed below is recommended. However, it is difficult to appreciate, let alone remember, the subtler details of the use of a function until you actually *need* to use it! If you want to see the effect of any of the functions use the following program:

```
1Ø INPUT x
2Ø PRINT SIN x
3Ø GOTO 1Ø
```

but change the PRINT SIN x to the function that you are interested in.

Arithmetic Functions

ABS – ABSolute value of a number

The absolute value of a number is obtained by ignoring its sign and treating it as positive, i.e. ABS(-2) is 2 and ABS(2) is 2.

EXP – EXPOnential function

EXP(x), the exponential number, which has the value 2.718281, is calculated by raising 'e' to the power of 'x'. That is, EXP(x) is the same as e^x .

It is difficult to explain why this function is so important but it

crops up in just about every area of mathematics, (see also the function LN). When using EXP it is well worth being aware of the fact that EXP(x) gets very big for even small values of 'x'. The largest integer that EXP(x) can accommodate on the Spectrum is 88. Larger numbers will cause the message "Number too big" to be displayed.

INT – INTeger value of a number

The INT function is probably the simplest and most used of all the arithmetic functions. It will remove the fractional part of a number and turn it into a *whole* number or an *integer* by rounding it down. For positive numbers this corresponds to *chopping off* the fractional part, e.g. INT(3.21) is 3 but for negative numbers things are a little more complicated. Rounding a negative number down looks a little strange, e.g. INT(-4.7) is -5, but this is simply because $-4 > -5$ i.e. -5 is *smaller* than -4!

LN – Natural Logarithm of a number

The natural logarithm of a number is the power to which you have to raise 'e' to give the number. Most people are more familiar with a slightly different form of the logarithm. The logarithm that is given in most log tables is in fact the logarithm to the base 10. In other words it is the number to which 10 has to be raised to give the original number. Some versions of BASIC include the log to the base 10 in the form of the LOG function. The Spectrum lacks such a function but this is no great disadvantage because you can obtain the log to the base ten by using $\text{LN}(x)/\text{LN}(10)$. As LN(x) is the number to which you have to raise 'e' to get x you should be able to see that $\text{EXP}(\text{LN}(x))$ is x.

PI (π)

This is a very odd function in that it has no parameters and always returns the same result ($\pi=3.14159265$) but it is very useful. As everyone knows, the area of a circle of radius r is given by πr^2 and this translates to BASIC as:

```
10 LET area=PI*r*r
```

SGN – the SiGN of a number

The sign of a number is +1 if the number is positive and -1 if it is negative. For example, $\text{SGN}(-232)$ is -1 and $\text{SGN}(3239)$ is 1.

SQR – the SQuare Root of a number

The square root of a number is a result that when multiplied by itself gives the original number i.e.:

$$\text{SQR}(x) * \text{SQR}(x) \text{ equals } x$$

Notice that negative numbers do not have square roots because if you multiply any number, even a negative one, by itself you will get a positive number. If you try to take the square root of a negative number you will therefore get the very precise error message, "Invalid argument".

Trigonometrical functions

The best known examples of functions are probably the *trigonometrical* or *trig* functions. It is beyond the scope of this book to go into detail about the theory of trigonometry and anyway the need to use such functions always arises from a very specific problem. There is, however, one use for the trig functions that is important to nearly every computer user interested in graphics, namely drawing circles. If you want to know more about this topic then see Chapter 10 of the *Spectrum Manual*. Spectrum users are particularly lucky, however, in that Spectrum BASIC includes a command that will plot a circle. This is covered in Chapter Nine and its presence avoids the need to go into any detail about using SIN, COS and TAN to draw circles.

If you do need to use any of the trig functions, it is important to realise that the Spectrum doesn't measure angles in degrees but in radians. (Radians as a measure of angle is dealt with in Chapter Nine where it is explained with reference to drawing parts of circles on the screen.) If you want to convert an angle in degrees to radians then use:

$$\text{radians} = \text{degrees} * \text{PI} / 180$$

and to convert radians to degrees use:

$$\text{degrees} = \text{radians} * 180 / \text{PI}$$

The three trig functions available on the Spectrum are:

- SIN – SINE of an angle measured in radians
- COS – COSINE of an angle measured in radians
- TAN – TANGENT of an angle measured in radians

The Spectrum also has available the inverse function related to these three.

ASN – ArcSiNe

The arcsine of a number is the angle in radians whose SIN is equal to the number, i.e. $x = \text{SIN}(\text{ASN}(x))$. Because SIN gives a result between +1 and -1, ASN(x) can only be worked out for x in this range.

ACS – ArcCoSine

The arccosine of a number is the angle in radians whose COS is equal to the number, i.e. $x = \text{COS}(\text{ACS}(x))$. Because COS gives a result between +1 and -1, ASN(x) can only be worked out for x in this range.

ATN – ArcTaNgent

The arctangent of a number is the angle in radians whose TAN is equal to the number, i.e. $x = \text{TAN}(\text{ATN}(x))$.

String Functions

We have already met some of the string functions in Chapter Five.

CHR\$ – CHaRacter function

CHR\$ n will give the 'character' that is at the nth position in the list of all the Spectrum's characters. Notice that as keywords such as LET count, from the Spectrum's point of view, as a single 'character' the function CHR\$ can return a string of more than one letter. CHR\$ will return all the characters that the Spectrum can use even if they cannot be printed on the screen. So, if you type CHR\$(8), or any number up to and including 33, all you will see is a blank screen.

CODE – the code of a character

The CODE function does the opposite to the CHR\$ function in that it returns the position in the list of characters of any particular character. For example, CODE "A" is 65 and CHR\$(65) is "A". If CODE is applied to a string of more than one letter the code of the first character in the string is returned as the result. If the string is null, i.e. contains no characters, the code returned is zero.

LEN – the LENgth of a string

The LEN function returns the length of any string. For example, LEN "computer" is eight and LEN "" (the null string) is zero.

STR\$

The STR\$ is a function that is useful for advanced applications. It converts any number (or the result of an expression) to the string of

characters that would be displayed if the number (or the result of the expression) were printed. The STR\$ function provides a link between numbers and strings – for example, “July ”+STR\$(31) works out to the string “July 31”.

VAL – eVALuate arithmetic expression

The VAL function is the opposite of the STR\$ function in that it converts a string into a number. The string can be any correct arithmetic expression and the resulting number is the value of the expression. For example, VAL “34” is 34 and VAL “3+3*6” is 21.

Special functions

There are two functions, RND and INKEY\$, that are so generally useful that it seems worthwhile to treat them on their own and at some length. RND is a function that returns a number so it could have been treated in the section on arithmetic functions. INKEY\$ returns a character so it could have been treated as a string function.

RND

RND is a function with no parameters that returns a number in the range 0 to less than 1 which can be treated as if it were random. To say that a computer can give a number at random always sounds like a contradiction and indeed to some extent it is. The point of confusion comes from the use of the word *random*. If you are using the computer to play a game then all that you need is a sequence of numbers that are not predictable by anyone playing the game. In other words, for most purposes a list of numbers can be said to be random if there is no detectable pattern. If you run the following program:

```
1Ø PRINT RND
2Ø GOTO 1Ø
```

you should see a list of numbers that shows no obvious pattern. (In fact there is a pattern but it is so complicated it takes a Spectrum to follow it!) This sort of randomness is more correctly called *pseudo randomness* and the RND function is a *pseudo random number generator*. The numbers that it produces are *evenly spread* throughout the range 0 to less than 1, i.e. any number is just as likely to *come up* as any other and there should be no discernible pattern that would help someone predict the next number that RND will produce.

The main trouble with RND is that it's not often that we need a

random number in the range 0 to less than 1. We normally need the program to do one of a number of different things at random. The best way of doing this is to change the RND into a random whole number between 1 and n where n is the number of anything we want to select from, using the formula:

$$\text{INT}(\text{RND} * n) + 1$$

For example, if you want to program a six-sided dice then you would choose six as the value of n – and:

```
1Ø PRINT INT(RND*6)+1
2Ø GOTO 1Ø
```

will print numbers 1 to 6 with approximately the same frequency and in such a way that there should be no obvious pattern. The subject of how to use random numbers in programs is too vast to cover in this book but examples will crop up in later chapters.

The RND function is also special because it is associated with another BASIC command, RANDOMISE. The list of numbers that RND produces doesn't go on for ever. Eventually after 65 536 values it repeats itself. Every time you switch the Spectrum on the list starts from the same place. If you want to check this, first switch your Spectrum off and on again and then enter the program that prints a screen full of random numbers. No matter how often you repeat this procedure, remembering to switch off and on again each time, you will get the same numbers in the same sequence. Obviously this is not a good idea if you want to play games because you might eventually learn the sequence that is produced when the Spectrum is first switched on. On the other hand, you might actually want to generate the same sequence each time and if this meant switching the machine off between each run this would also create difficulties. To overcome both these problems, you can use the RANDOMISE command (entered by pressing the key marked RAND) to start the sequence off. The command RANDOMISE n will start the sequence off from the nth random number in the Spectrum's fixed list. For example, if you enter:

```
1Ø RANDOMISE 3Ø
2Ø PRINT RND
3Ø GOTO 2Ø
```

you will get the same sequence of numbers every time you run it without switching the machine off. However, if you use RAN-

DOMISE \emptyset or RANDOMISE without any starting number the Spectrum will use a number that is related to the time that the machine has been switched on to start the sequence. To see this in action try:

```
1 $\emptyset$  RANDOMISE  $\emptyset$ 
2 $\emptyset$  PRINT RND
3 $\emptyset$  GOTO 1 $\emptyset$ 
```

which prints the first number in the sequence produced by RANDOMISE \emptyset over and over again. You should see that the numbers printed slowly increase, counting the time that the Spectrum has been switched on. The most random sequence that you can produce using RND and RANDOMISE can be seen by running:

```
1 $\emptyset$  RANDOMISE  $\emptyset$ 
2 $\emptyset$  PRINT RND
3 $\emptyset$  GOTO 2 $\emptyset$ 
```

INKEY\$

The function INKEY\$ is closely related to INPUT in that it can be used to *read in* a single character from the keyboard. The difference is that INPUT A\$ waits for something to be typed on the keyboard until the ENTER key is pressed but INKEY\$ doesn't wait. If you try the following program:

```
1 $\emptyset$  INPUT A$
2 $\emptyset$  IF A$<>" " THEN PRINT A$
3 $\emptyset$  GOTO 1 $\emptyset$ 
```

you will have to press ENTER before you see anything on the screen. (To stop this program you will need to delete the left hand quote marks and then type in STOP.) However, if you change line 1 \emptyset to:

```
1 $\emptyset$  LET A$=INKEY$
```

the character corresponding to any key that you press appears on the screen at once. (While running this second version of the program try pressing more than one key at a time and try pressing SHIFT and the other keys.) Notice that another difference between INPUT and INKEY\$ is that INKEY\$ doesn't automatically print anything that you type on the bottom of the screen.

Whenever the Spectrum meets the INKEY\$ function it immediately examines the keyboard. If there is a key already pressed the

appropriate character is returned by the function. If no key is pressed the function returns the null string. No matter what has happened the INKEY\$ function does *not* wait for a key to be pressed.

The main use of INKEY\$ is in games where the *arrow* keys used for editing are used to control the movement of something on the screen. For example:

```

10 LET a$=INKEY$
20 IF a$="" THEN GOTO 10
30 IF a$="5" THEN PRINT "left"
40 IF a$="6" THEN PRINT "down"
50 IF a$="7" THEN PRINT "up"
60 IF a$="8" THEN PRINT "right"
70 GOTO 10

```

Line 10 gets the character corresponding to any key pressed on the keyboard, if any. Line 20 tests to see if a\$ is the null string, i.e. no key has been pressed, and if it is, sends control back to line 10. Thus the loop formed by line 10 and 20 only stops when a key is pressed. Then lines 20 to 50 test to find out which of the arrow keys are pressed and print an appropriate message. Line 70 repeats the whole program. Notice that if any key other than an arrow key is pressed the loop formed by lines 10 and 20 stops but nothing is printed on the screen. In Chapter Nine an example is given where the same sort of program is used to drive a dot around the screen.

User-defined functions - DEF FN and FN

In our general introduction to functions, the idea of a function to find the maximum value of two numbers was discussed but it was pointed out that, although the Spectrum has a wide range of functions, such a 'max' function isn't among them. Spectrum BASIC does allow the definition of new functions but there are some tricky ideas involved.

You can define a new function in terms of an expression. For example, the Spectrum lacks a *square* function, i.e. one that will work out the square of any number. The names of all user-defined functions are 'FN' followed by one letter, so you can define the new function 'FN s' to fill this gap by:

```
DEF FN s(x)=x*x
```

(DEF FN is entered by one keystroke. You'll find it at the top left

hand corner of the keyboard, on the I key.) The word DEF is used to indicate that this is a function definition. The meaning of this function should be clear – whenever the Spectrum sees the function called ‘FN s’ it squares the parameter within the brackets next to it. For example:

```
LET a=FN s(2)
```

would result in 4 being stored in the variable ‘a’. (The FN is again entered as one keystroke. It is on the 2 key.) This is all fairly straightforward but what about the following:

```
1Ø DEF FN s(x)=x*x
2Ø LET x=3
3Ø LET z=4
4Ø LET a=FN s(z)
5Ø PRINT “x= ”;x;“ z= ”;z;“ a= ”;a
```

before you run this program try to work out what the values of ‘x’, ‘z’ and ‘a’ will be. The correct answer is that ‘x’ is 3, ‘z’ is 4 and ‘a’ is 16. Any possible confusion comes from the fact that ‘x’ is used in the definition of the ‘FN s’ function and in the program. A point to note is that the names that you give to parameters in the definition of functions are nothing to do with any variables that you might use in the rest of the program. In this sense DEF FN s(a)=a*a, DEF FN s(z)=z*z, etc., all define the same function! In a function definition the parameters merely show what is to happen to the *real* parameters when the function is used – because of this they are often called *dummy parameters*.

If you’ve managed to cope with these ideas, you might like to try to work out what this program’s result is:

```
1Ø DEF FN t(i)=total+i
2Ø LET total=Ø
3Ø PRINT FN t(3Ø)
4Ø LET total=1Ø
5Ø PRINT FN t(3Ø)
```

In this case the variable ‘total’, used in the expression to define the function isn’t a dummy parameter and it is therefore taken to be a variable in the main program, i.e. ‘total’ in the function definition is the same as ‘total’ in the rest of the program. The idea of a dummy variable gets easier after you have had time to think about it.

To summarise, the rules for forming a user-defined function are:

- (1) Every function that you use must be defined using a DEF FN statement somewhere in the program (not necessarily before it is first used).
- (2) Both numeric and string functions are allowed. Numeric functions, i.e. functions that return a number as a result have names of the form 'FN' followed by a single letter. String functions, i.e. functions that return a string as a result have names of the form 'FN' followed by a single letter and a dollar sign.
- (3) A function definition can include any number of dummy parameters (or none at all) which can either stand in place of numeric parameters or string parameters. All dummy parameters have one-letter names and dummy parameters standing in place of string parameters must end with a dollar sign. Brackets must be used even if there are no parameters.
- (4) Any valid BASIC expression can be used in a function definition.

These four rules may seem like a lot to remember but they all accord with common sense. Some examples of functions may help to clarify matters:

```
DEF FN m$(a$,i,j)=a$(i TO i+j-1)
DEF FN l(x)=LN x/LN 10
DEF FN d()=INT(RND*6)+1
```

The first function 'm\$' is a string function with one dummy string parameter and two dummy parameters. It will extract 'j' characters starting from character 'i'. For example, m\$("Hello",1,4) gives "Hell". The second function 'l' is a numeric function with a single numeric dummy parameter that will calculate the log to the base ten of x (see the notes on the LN function earlier). Notice that 'any valid BASIC expression' in point (4) includes expressions that involve other BASIC functions! The third and final example has no parameters but notice that the brackets () are still necessary in the definition. It returns as a result a random number in the range 1 to 6 (see the details of the RND function above). You even need the '(' when you use the function 'd', e.g. PRINT d().

We can now use what we've learned about user-defined functions to create the 'max' function that was used earlier to introduce the idea of functions. If you think about it, it seems as though a user-defined function couldn't be used to produce a 'max' function. After all how can you write an expression that will choose between one of two numbers? If, however, you followed the discussion of

conditional expressions in Chapter Four, and remember how they evaluate to 1 if true and to 0 if false, the answer to this problem should be easy to understand:

```

1Ø DEF FN m(a,b)=a*(a>=b)+b*(a<b)
2Ø INPUT a
3Ø INPUT b
4Ø PRINT FN m(a,b)
5Ø GOTO 2Ø

```

If you run this program you will find that line 4Ø always prints the larger of the two numbers that you type in response to lines 2Ø and 3Ø. The function itself works because only one of the two conditions '(a>=b)' and '(a<b)' can be true and hence one will work out to 1 and the other to 0. As zero times any number is zero you should now be able to see how one of the two numbers is selected. This example once again emphasises the fact that *any* valid BASIC expression can be used in a function definition.

Subroutines GOSUB and RETURN

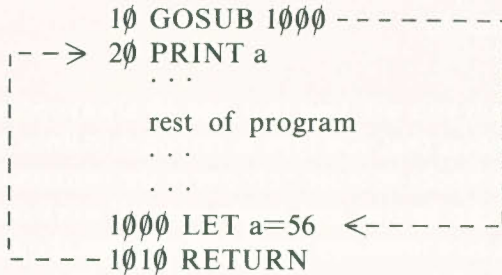
The idea of creating new operations by defining functions is a very powerful one but it is already possible to see its shortcomings. Sometimes it would be an advantage to give a name, not just to one line of BASIC in the form of a user-defined function, but to a whole collection of lines. This is the idea behind a *subroutine*. A subroutine is nothing more than a group of BASIC statements that can be used as often as required just by writing their *name*. (In the same way that a single line of BASIC can be used as often as required by using a user-defined function's name.) The trouble with BASIC subroutines is that they are very limited. You cannot even give a one-letter name to a subroutine. It has to be referred to by the line number of its first line and there is no provision for parameters of any sort. Even with these restrictions the BASIC subroutine is still well worth knowing about and using.

If the lines of BASIC that go to make up the subroutine start at line 'n' then you can use the subroutine by:

```
GOSUB n
```

Where GOSUB stands for GO to SUBroutine. Indeed, the action of a GOSUB is very like GOTO in that it transfers control to line 'n'. The difference between a GOTO and a GOSUB is that the GOSUB

command causes the Spectrum to store the line number of the GOSUB in a special area of memory set aside for the purpose. This stored line number is used by the RETURN statement to transfer control to the line following the GOSUB when the subroutine has finished. For example, the effect of a GOSUB and a RETURN on the flow of control is:



Line 10 transfers control to the subroutine that starts at 1000, which simply stores 56 in the variable 'a'. The RETURN statement at line 1010 ends the subroutine and automatically transfers control back to line 20.

You can use *any* BASIC statement that you can use elsewhere in a program within a subroutine. In particular, there is nothing to stop a subroutine from using GOSUB and transferring control to another subroutine. If you do this then the next RETURN will transfer control back to the statement after the most recent GOSUB. In other words, if one subroutine calls another, then RETURN behaves as you would expect it to, by transferring control back to the place that each subroutine was initiated, i.e. a RETURN never forgets where it came from!

This pair of instructions GOSUB and RETURN are all that there is to the BASIC subroutine. All the variables in a subroutine are the same as the variables in the rest of the program, there are no parameters of any kind. As suggested earlier this might lead you to believe that subroutines are not very useful – this is far from the truth.

Using subroutines

If you read the *Spectrum Manual* (Chapter 5) you are led to believe that the most useful way to use a subroutine is to replace any piece of program that is needed more than once. For example, if in a large program you need to print the same message over and over again,

then it is better to turn that line into a subroutine and GOSUB to it every time it is needed. Although this is an important use of subroutines it is often the case that it is a good idea to form parts of a program into subroutine even if each part is only used once! The reason for this is that programs that use subroutines are easier to understand, easier to find mistakes in and easier to modify. This is not the sort of statement that anyone can prove because what is easier in this context is clearly a matter of opinion. The use of subroutines in writing BASIC programs will be illustrated by the examples in the rest of the book. If you discover some other method of programming that you like better, then no one will be able to argue with you! All I can say is that many programmers agree that subroutines are a good thing!

Chapter Seven

Graphics

The main delight of programming the Spectrum is using its graphics and sound! This chapter starts by introducing the sort of graphics that can be produced using PRINT statements, *low-resolution graphics*. Chapter Eight adds sound and Chapter Nine deals with drawing pictures in finer detail, *high-resolution graphics*. You shouldn't be misled into thinking that high-resolution graphics are in some way more useful than low-resolution graphics. They are not more advanced, just different and it's amazing how often a program works better and is easier to write using low-resolution graphics.

Controlling PRINT

So far we have used the PRINT statement to print numbers and strings on the screen, either one to a line or next to each other on the same line. Although this is sufficient for most programs there is often a need to control exactly where something will be printed on the screen. In Chapter Three the PRINT statement was defined as:

```
PRINT 'print list'
```

where 'print list' was explained to be a list of items, each one separated by a semicolon. The need for the semicolons is simply to show where one item ends and another begins. For example, PRINT total sum will try to print a single variable called 'totalsum' (remember blanks are ignored in variable names), while PRINT total;sum will try to print two variables 'total' and 'sum' next to each other on a single line.

The Spectrum actually allows the use of three symbols to separate print items and each one has a different effect on the layout. The semicolon, ; , that we have been using since Chapter Three simply means print the next item without leaving any space. Using a comma

, as a separator means move to the next print *zone* before printing the next item. The Spectrum's screen is divided into two print zones – the first sixteen printing positions on a line and the last sixteen positions on a line. So PRINT “a”, “b” will print “a” in column one and “b” in column seventeen. The third and final separator is the apostrophe, ' , which the Spectrum interprets as an instruction to start a new line before printing the next item. So:

```
PRINT “a”“b”
```

has the same effect as:

```
PRINT “a”
PRINT “b”
```

You can replace more than one separator between any two print items without any ill effects, e.g. PRINT “a”,, “b” will cause the Spectrum to move on two print zones before printing “b”. If you think about it, this means that “a” will be printed at the beginning of a line and “b” will appear at the beginning of the next line. The most important case of using a separator more than once is the repeated use of apostrophes to leave a number of blank lines. There is one special case that is worth commenting on. If any of the three separators are placed at the end of a list of print items the automatic starting of a new line is suppressed. For example:

```
PRINT “a”,
PRINT “b”
```

is the same as PRINT “a”, “b”
and

```
PRINT “a”;  
PRINT “b”
```

is the same as PRINT “a”; “b”

Of course, ending a PRINT statement with an apostrophe suppresses the automatic starting of a new line but the apostrophe itself forces a new line so there is no noticeable difference between PRINT “a” and PRINT “a”'.

To summarise:

<i>separator</i>	<i>effect</i>
;	print next item without leaving any space


```

,           move to next print zone
'           start a new line

```

PRINT functions - TAB and AT

The use of different print list separators has certainly increased our control over how things are printed on the screen but we still cannot easily make a print item start at a particular column on a particular line. To achieve this we need something more than different separators. Two special functions, TAB and AT, are provided to control exactly the place where any item starts printing. (They are *special* in the sense that although TAB and AT are written like functions, they produce no value as a result of their use and they can be used only as part of a 'print list'.) TAB can be used to control the horizontal position on the current line and AT is an entirely general function that can produce output anywhere on the screen.

The general form of the TAB function is:

```
TAB 'arithmetic expression'
```

and its effect is to move the printing position on to the column given by the value of 'arithmetic expression'. For example:

```
PRINT "a";TAB 10;"b"
```

will print "a" at column one and "b" at column 10. You can have as many TABs in a PRINT statement as you need so:

```
PRINT "a";TAB 10;"b";TAB 20;"c"
```

will print "a" at column 1, "b" at column 10 and "c" at column 20.

There are two possible problems that can arise when using TAB. What happens if you have already gone beyond the column specified by a TAB and what happens if you specify a column that doesn't exist, like TAB 35 (there are only 32 columns)? The answer to the first question is that the Spectrum will move to the correct column *on the next line*. You can see this if you try the following:

```
10 PRINT "a";TAB 15;"b";TAB 10;"c"
```

which prints "a" at column 1, "b" at column 15 and "c" at column 10 on the next line. The answer to the second question is that the Spectrum subtracts 32 from the column number that you specify until it gets a number in the range 0 to 32. For example, TAB 46 has the same effect as TAB 14, because 46-32 is 14, and TAB 126 has the

same effect as TAB 30, because $126 - 32$ is 94, $94 - 32$ is 62 and $62 - 32$ is 30!

The most powerful print function is AT because it can position output both at a particular line and a particular column. The general form of AT is:

AT 'arithmetic expression 1', 'arithmetic expression 2'

The value of 'arithmetic expression 1' is the line number that the next item will be printed on and the value of 'arithmetic expression 2' is the column number. (If the line or column specified is off the screen then you will get an "out of screen" error message.) Unlike TAB, which numbers printing positions 1 to 32, with AT the columns are numbered starting with 0 and ending with column 31. Similarly, the line numbering goes from 0, for the top line, to 21, for the bottom line. For example:

```
PRINT AT 3,5;"a"
```

will print "a" on line 3 (i.e. the fourth line down) and column 5 (i.e. the sixth printing position). (This might seem a little difficult if you are used to x,y co-ordinates where x is horizontal distance and y is vertical distance, because to move to printing position x,y you have to use PRINT AT y,x.)

An interesting difference between TAB and AT is that if you use TAB to move to a printing position it *clears* all of the screen from the current printing position. For example, try:

```
10 PRINT "*****"  
20 PRINT AT 0,0;  
30 PRINT TAB(10);"a"
```

It doesn't matter exactly how many asterisks you use in line 10, they are only there to show the effect of TAB. There are two interesting points about this demonstration. Firstly, you can use AT to move the current printing position anywhere on the screen without actually printing anything and secondly, line 10 shows that the TAB to column 10 erases all the asterisks from the beginning of the line to column 10. If you want to see the effect of AT (it doesn't erase the asterisks but prints the "a" in place of the tenth one) substitute:

```
30 PRINT AT 0,10;"a"
```

for line 30.

You can have as many ATs in a 'print list' as you desire. Each one moves the printing position to the place indicated before going on to

the next item. Not only can you have more than one AT, you can mix AT, TAB and all the other print control symbols in a print list in any way that makes sense to you. For example:

```
10 PRINT AT 10,5;"a";TAB(5);"a";AT 3,10;"b""c"
```

is accepted without qualms by your Spectrum.

As an exercise in using AT try to write a program that draws a square made of asterisks on the screen. In case you have any problems, one of the many possible answers (there is always more than one way of writing a program) is:

```
10 FOR i=0 TO 10
20 PRINT AT 5,i+10;"*"
30 PRINT AT 15,i+10;"*"
40 PRINT AT i+5,10;"*"
50 PRINT AT i+5,20;"*"
60 NEXT i
```

```

* * * * *
*
*
*
*
*
*
*
*
*
* * * * *

```

Fig. 7.1. Square using asterisks

Lines 20 and 30 print the two horizontal rows of asterisks and lines 40 and 50 print the two vertical columns of asterisks.

A full screen - CLS and scrolling

The Spectrum's screen is 32 characters by 22 lines and sooner or later you are bound to use all the space and still want to print yet more information. If you have finished with everything already on the screen you can use the CLS statement (CLS stands for CLear the Screen), which simply wipes everything off the screen and sets the current printing position back to the top left hand corner of the screen.

If you are printing things on the screen working from top to bottom there will come a time when you reach the bottom line. If you try to print another line the Spectrum will print the message "Scroll ? y/n" and wait for your answer. If you press any key apart

from "n", SPACE or STOP, the whole screen is moved up one line, the top line is lost and the new information is printed on the newly created bottom line. This technique of moving the screen up by one line is known as *scrolling* and was introduced briefly in Chapter Two in connection with listing programs. After this first scroll new bottom lines are introduced, i.e. the screen is scrolled, every time a PRINT tries to take the printing position off the bottom of the screen. After 22 scrolls the line created by the first scroll is about to disappear off the top of the screen and the Spectrum once again asks if it's all right to scroll. In this way, the Spectrum makes sure that you always have a chance to see each screen full of numbers before they start vanishing off the top. To see this in operation try, for example:

```
1Ø PRINT RND
2Ø GOTO 1Ø
```

If for any reason you want to make the screen scroll before it would normally do so you can use:

```
PRINT AT 21,Ø''
```

which first moves the current printing position to line 21 and then tries to move it on one more line.

Controlling INPUT

It may come as something of a surprise to find that the INPUT statement can use all of the items that a PRINT statement can. This is a pleasant discovery because it greatly increases the range of input prompts that we can give. The general form of the INPUT statement is:

```
INPUT 'print list'
```

There are, however, some additional rules and important differences in the way that the 'print list' is interpreted when used in the INPUT statement. The first change that is necessary is to indicate which variables are to have their values changed by the INPUT and which are to be printed out as part of the prompt. The rule that the Spectrum uses is that any 'print items' that begin with a letter are treated as input variables. (Notice that TAB and AT don't begin with letters because they are single keystrokes.) For example, the INPUT statement:

```
10 INPUT "a=";a;"b=";b
```

will print a prompt, then wait while you type in the value for 'a', print a second prompt and wait while you type in a value for 'b'. You can short-circuit the first letter rule by enclosing any variables or expressions in brackets. The meaning of an expression isn't changed by wrapping it up in an unnecessary pair of brackets but then it no longer begins with a letter. For example:

```
10 LET a=10
20 INPUT "a=";(a);"b=";b
30 PRINT a,b
```

will print the value of 'a' on the bottom line and then wait for you to type in the value of 'b'.

By now you will have noticed the strange way that PRINT and INPUT work in separate parts of the screen. This division of the screen applies to the way AT works. If you use AT in a PRINT statement then you count lines starting at the top of the screen from zero. However, if you use AT in an INPUT statement then you have to start counting lines from the bottom of the screen starting from zero at the first line in the input area (see Fig. 7.2). For example try:

```
10 INPUT AT 0,5;"top line";AT 1,5;"next line";a
```

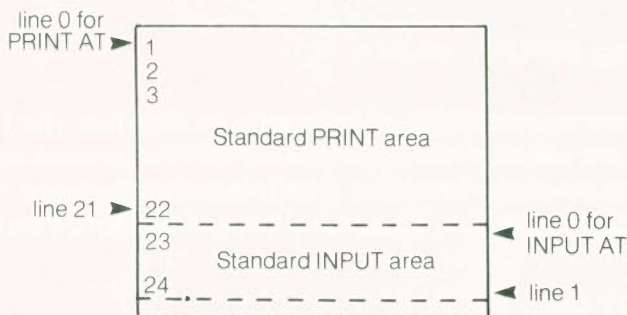


Fig. 7.2. Screen areas

This prints two prompts, one on each line of the input area, and then waits for you to type in the value of 'a'. At this point you might be wondering what happens if you try to print on lines in the input area other than lines 0 and 1. The answer is that the Spectrum will scroll the input area up to bring the line onto the screen. To see this in operation, try:

```
10 INPUT AT 10,0;"this line is normally off the screen";
   AT 0,0;"and this is the top line";a
```


If there is anything printed on the upper part of the screen and an INPUT wants to print something in the same area, then the print area of the screen is scrolled up to make some space for the input area. This is something that is a lot easier to understand once you have seen it happen:

```
1Ø PRINT AT 15,Ø;"This is printed on line 15"
2Ø PRINT AT 16,Ø;"This is printed on line 16"
3Ø INPUT AT 2Ø,Ø;"This is printed on line 2Ø of the input
  area";AT Ø,Ø;"this is the top line of the input area";a
```

You can write an INPUT statement that only prints on the screen but you are unlikely to see the message that it prints because *every* INPUT statement clears the input area when it is finished! This means that your message will be briefly flashed on the screen, which is then cleared and your Spectrum moves on to the next instruction. Not only does every INPUT statement clear the input area of the screen it also restores it to its original size, i.e. the bottom two lines of the screen.

The division of the screen into two different areas, the print and the input area, is sometimes said to be a problem with the Spectrum (and the ZX81) but if you understand how it works you should be able to turn it to advantage!

The graphics characters

The combination of PRINT and AT can obviously be used to place a character anywhere on the screen and, as we saw earlier in this chapter (in the program that prints a square), this can be used to produce limited graphics. This ability really only comes into its own when used with the Spectrum's range of graphics characters. These can be produced by pressing CAPS SHIFT and GRAPHICS (9) to change the cursor to  and then pressing another key. You can leave graphics mode by pressing GRAPHICS again. Although something is displayed on the screen for every key that you press in graphics mode, there are only 37 graphics characters. Eight of these are printed in white on the top row of keys alongside the numbers. You can get eight more by pressing any of the top row of keys together with CAPS SHIFT. These additional eight are simply the original eight reversed, i.e. black changed to white and vice versa but it is important to realise that they are distinct characters. This set of sixteen graphics characters can also be printed on the screen by

using their character codes in the CHR\$ function. To see this complete set of sixteen try the following program:

```
10 FOR i=128 TO 136
20 PRINT i,CHR$(i)
30 NEXT i
```

The other 21 graphics characters are produced by the keys 'A' to 'U' entered while in graphics mode and they correspond to the character codes 144 to 164. As will be explained later, these can be changed to produce any shape that is required, that is they are *user-defined* but initially they reproduce the upper case character set. So if you type a graphics 'A', an upper case 'A' appears on the screen. Apart from these 37 graphics characters, pressing a key while in graphics mode simply produces a character that can be entered by some other method. To be specific, while in graphics mode, pressing any key enters the character that is 96 further on in the list of characters. In other words if the key produces character CHR\$(n) it will produce CHR\$(n+96) in graphics mode.

Apart from having to be entered in a different mode, the graphics characters behave like any others. For example, they can be used in PRINT statements and strings. The program that printed a square of asterisks can be changed to print a better looking square using graphics characters:

```
10 FOR i=1 TO 9
20 PRINT AT 5,i+10; "[3]"
30 PRINT AT 15,i+10; "[3]"
40 PRINT AT i+5,10; "[5]"
50 PRINT AT i+5,20; "[5]"
60 NEXT i
```



Fig. 7.3. Square using graphics characters

(As explained in Chapter Two, because of the difficulties of printing graphics characters in a book, they are represented by the main character on the key enclosed in square brackets, e.g. [1] is the graphics character on the key marked '1'. If the character has to be

entered using CAPS SHIFT to produce the required graphics character then a ' will be written before it. So [↑1] is the graphics character produced by pressing '1' with the CAPS SHIFT held down.) The trouble with this square is that the corners are missing and putting them in is a matter of printing at the correct positions the L-shaped graphics symbols. This is left for you to remedy.

As you can imagine, drawing more complicated shapes using the graphics characters is very difficult. Fortunately, apart from drawing the occasional 'thick' horizontal or vertical line, the graphics characters on the top row of the keyboard are normally used in small numbers to print special shapes. For example, if you want to print the outline of a ship during a game you could use:

```
PRINT AT y,x;"[↑2][↑3][↑3]"
```

which will print a ship at line y and column x.



Fig. 7.4. Ship graphics

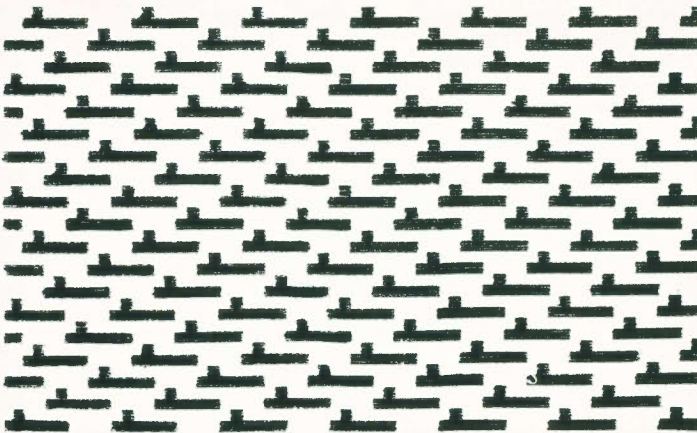


Fig. 7.5. Ship graphics

If you would like to see a ship move across the screen try

```
10 FOR x=0 TO 27
20 PRINT AT 5,x;" [↑2][↑3][↑3]"
30 NEXT x
```


Notice the space left before the first graphics character in line 20. If you want to know what the space is for try leaving it out!

User-defined graphics characters

The range of shapes that can be made up of combinations of the graphics characters on the Spectrum's top row of keys, discussed in the previous section, is fairly limited. For example, how would you make up the shape of a man? Fortunately the Spectrum has 21 graphics characters that can be altered to produce any shape that you could ever want.

Before we can go on to explain how to define new characters we first have to examine how characters are produced on the screen. Every character that the Spectrum can display on the screen is in fact produced from a grid of 64 dots arranged into a square grid, eight dots by eight dots. The pattern of any character depends on which dots in the grid are displayed as black and which are displayed as white. By analogy with printing or writing on paper, black dots are referred to as 'ink' dots and white dots are referred to as 'paper' dots. For example, the letter 'a' is produced by the pattern of dots shown in Fig. 7.6 (i stands for ink and p for paper).

```

p p p p p p p p
p p p p p p p p
p p i i i p p p
p p p p p i p p
p p i i i i p p
p i p p p i p p
p p i i i i p p
p p p p p p p p
    
```

Fig. 7.6.

You might find it difficult to see the pattern of the letter 'a' among the 'i's and 'p's but it becomes very clear if each 'i' is replaced by an asterisk and each 'p' is replaced by a blank as in Fig. 7.7.

```

  * * *
    *
 * * * *
*           *
  * * * *

```

Fig. 7.7.

Obviously, if we are going to define the shape that corresponds to a user-defined graphics character, there must be some way of specifying which dots in the 8 by 8 grid are ink and which are paper. The Spectrum provides a very simple way of doing this by use of the USR and the BIN functions. The user-defined character which replaces the existing definition for the character [n], where n is an upper case letter from 'A' to 'U', involves the function:

USR "n"

For example, if we are going to define a new character in the shape of a man and we want it to be produced when [M] (graphics M) is typed, the function:

USR "M"

is used. The definition of the new character has to be done a row at a time. Each row in the grid can be written as a sequence of eight digits by writing a 0 for every paper dot and a 1 for every ink dot. For example, the row 'ppiiipp' would be written as 00111000. Using this representation each row of the new character can be redefined by

POKE USR "M"+r,BIN 'dot pattern'

where 'r' is the row number to be defined (the first row is numbered 0) and 'dot pattern' is the sequence of 0s and 1s that correspond to the ink and paper pattern. The command POKE and function BIN will be described in Chapter Ten but exactly what they do is unimportant for defining characters – they are always used in the same way. To define a complete character you have to define eight rows and so use the above statement eight times but remember there is nothing to say that you *have* to change all eight rows from their existing definitions.

All this will be easier after an example. The little man character introduced earlier could be defined using the following dot pattern:

```

00011000
00011000
11111111
00111100
00111100
00100100
00100100
00100100

```

where 1 has been used to mean ink and 0 to mean paper. This can be transferred to the graphics [M] key row by row by:

```

10 POKE USR "M" ,BIN 00011000
20 POKE USR "M"+1,BIN 00011000
30 POKE USR "M"+2,BIN 11111111
40 POKE USR "M"+3,BIN 00111100
50 POKE USR "M"+4,BIN 00111100
60 POKE USR "M"+5,BIN 00100100
70 POKE USR "M"+6,BIN 00100100
80 POKE USR "M"+7,BIN 00100100
90 PRINT "[M]"

```

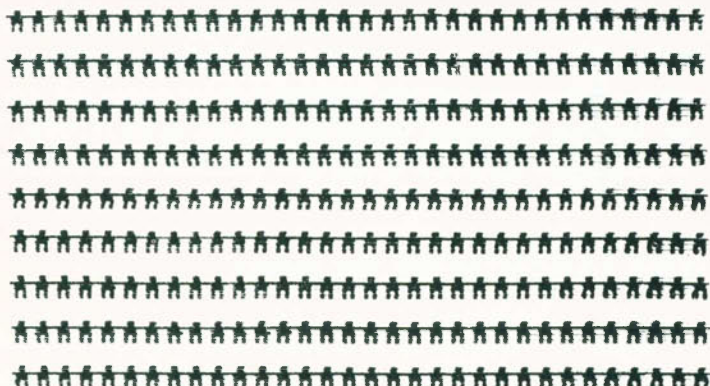


Fig. 7.8. Man figure

Notice that after you have run this program the [M] in line 90 displays as the little man shape. The reason for this is that once you have defined a graphics character the definition holds until you either redefine it or switch the machine off.

User-defined graphics characters are most useful for producing the special shapes that are so essential to any sort of games program. For example, a large dot character could be used as a ball. However,

there are some serious uses of user-defined graphics characters such as showing mathematical or chemical formulae on the screen.

Changing the way characters look - INVERSE and OVER

The two commands INVERSE and OVER do not create or give us access to any more new characters but they can be used to alter the way existing characters appear on the screen. The command INVERSE is perhaps the easier to understand. The effect of:

INVERSE 1

is to swap the ink and paper dots of every character subsequently printed, i.e. ink dots become paper dots and vice versa. This means that instead of the usual black characters on a white background you get white characters on a black background. To change back to the usual assignment of ink and paper the command:

INVERSE Ø

has to be used. You can think about the Ø and 1 following the word INVERSE as meaning 'off' and 'on'. For example:

```
1Ø INVERSE 1
2Ø PRINT "this is inverted"
3Ø INVERSE Ø
4Ø PRINT "this is back to normal"
```

Although INVERSE is not a difficult command to understand, there are two important points to notice - INVERSE does not create any new characters it merely changes its definition by changing ink dots to paper dots and vice versa. The effect of INVERSE 1 continues until the Spectrum obeys an INVERSE Ø or until it is switched off. The result of this persistence of the INVERSE command is that if you stop a program before turning the INVERSE off then all your listings, etc., will appear on the screen in inverse mode. The solution to this problem is to enter INVERSE Ø in immediate mode, i.e. without a line number.

The command OVER is very similar to INVERSE, in that it alters the way that ink and paper dots are displayed, but it is slightly more difficult to explain. After the command:

OVER 1

whether an ink or a paper dot is displayed depends on what is

already on the screen. If the new dot and the old dot are both the same then a paper dot is displayed. If the new dot and the old dot are different then ink is displayed. This may seem complicated but it can easily be summarised:

<i>new dot</i>	<i>old dot</i>	<i>displayed</i>
paper	paper	paper
paper	ink	ink
ink	paper	ink
ink	ink	paper

and can be remembered by 'two anythings make a paper and one of each makes an ink'. (This behaviour is known as an *exclusive OR* of the old dot with the new, see Chapter Ten.) To see what effect this has try the following:

```
1Ø OVER 1
2Ø PRINT AT Ø,Ø;"hi there !"  
3Ø PRINT AT Ø,Ø;"-----"
```

You should see the message produced by line 2Ø underlined. The way that this works is that the bottom row of dots of every character is a row of paper dots and the underline character is simply a bottom row of ink dots. By the rules given above, paper and ink make ink so the underlining appears as the bottom row of each character already on the screen. The reason why OVER is so named is because this effect is very similar to over-printing on a typewriter. However, it is important to notice that in many ways OVER is nothing like over-printing on a typewriter! For example try:

```
1Ø OVER 1
2Ø PRINT AT Ø,Ø;"O"  
3Ø PRINT AT Ø,Ø;"/"
```

On a typewriter this would have produced a letter O with a slash / through it but because two inks make a paper the dots where the slash and the 'O' are black appear as white paper dots. Although this might appear to be a nuisance it can be used to good effect. Try, for example,

```
1Ø OVER 1
2Ø PRINT AT Ø,Ø;"*"  
3Ø GOTO 2Ø
```

The first time the asterisk is printed it appears on the screen because

all the original dots are paper. The second time it disappears because all the ink dots are in the same place and so they *cancel out*. This is repeated each time the asterisk is printed and so we get the flashing effect observed.

It is important to notice that OVER is the same as INVERSE in that it doesn't create any new characters only alters the way they are displayed and its effects continue until it is cancelled by:

OVER Ø

or the machine is switched off.

Character attributes - FLASH and BRIGHT

So far we have concentrated on the position and shape of the characters printed on the screen. However there are other things that we can control about the way the Spectrum displays characters. The commands FLASH and BRIGHT can be used to alter the way that any given characters are displayed on the screen. The command:

FLASH 1

causes all subsequent characters to be printed *flashing* and:

BRIGHT 1

causes all subsequent characters to be printed brighter than normal. As for INVERSE and OVER you can cancel the effects of FLASH and BRIGHT by using Ø instead of 1 in the commands. These two commands are easier to illustrate than they are to explain. Try the following program:

```
1Ø FLASH 1
2Ø PRINT "Flashing characters"
3Ø FLASH Ø
4Ø BRIGHT 1
5Ø PRINT "Bright characters"
6Ø BRIGHT Ø
```

The message printed at line 2Ø will flash (black changing to white and white changing to black) and the message produced by line 5Ø will be brighter than the surrounding screen. As you might expect the effect of FLASH and BRIGHT persists until you either cancel it or switch the machine off.

Colour - BORDER, INK and PAPER

You may be surprised to find that it has taken so long to reach the exciting subject of colour! The reason for this delay is that the way the Spectrum's colour commands work is much easier to understand once the idea of characters being made up of ink and paper dots has been introduced. The Spectrum can display eight different colours and these are referred to by the numbers from 0 to 7:

- 0 - black
- 1 - blue
- 2 - red
- 3 - magenta (purple)
- 4 - green
- 5 - cyan (pale blue)
- 6 - yellow
- 7 - white

You do not, however, need to commit these numbers to memory as the colour names are printed (in appropriate hues) over the number keys. Notice how lower numbers correspond to darker colours. If you're not using a colour TV set with your Spectrum then the colours will appear as different shades of grey, the darker shades corresponding, as might be expected, to the lower numbers.

Perhaps the easiest way to see the Spectrum's range of colours is to use the **BORDER** command. The area of the TV screen that the Spectrum cannot **PRINT** on is known as the *border* and although you cannot **PRINT** on it you can determine its colour. (The border consists of the top and edges of the screen and from the bottom up to the top line of the input area.) The command:

BORDER c

will change the border to the colour corresponding to the number 'c'. (Note that, in general, 'c' can be an arithmetic expression.) Try the following program:

```
10 FOR i=0 TO 7
20 INPUT a$
30 BORDER i
40 NEXT i
```

which will change the colour of the border each time you press **ENTER**. To make the point that the input area is included in the border, replace line 20 by:

20 INPUT AT 20,0;a\$

In the same way that the colour of the border can be changed so can the colour of the rest of the screen. However unlike the border this involves specifying *two* colours, one for ink dots and one for paper dots. When the Spectrum is first switched on ink dots are set to black and paper dots are set to white but you can alter these settings using:

INK c

and

PAPER c

The command INK sets the colour of any ink dots subsequently printed and PAPER sets the colour of any paper dots subsequently printed. For example, try the following program:

```
10 INPUT "ink colour=";i
20 INPUT "paper colour=";p
30 INK i
40 PAPER p
50 PRINT "*";
60 GOTO 10
```

Using this program you can print asterisks with any given ink and paper colours. (Enter the colour codes in response to the questions printed by lines 10 and 20.) Notice that the Spectrum will even let you specify the ink and paper colour to be the same, in which case of course you don't see the asterisk! It is, however, important to realise that even though you cannot see it, the asterisk is still there – its pattern of ink and paper dots are present on the screen but it is invisible because both ink and paper are being displayed as the same colour.

This is almost all that there is to the colour commands on the Spectrum yet it is possible to create some excellent colour effects using them. However, it is important to be aware of the limitations built into this simple method of colour control. Each eight by eight character block can only display two different colours – the paper colour and the ink colour. This means that, although you can have eight colours showing on the screen at the same time, only two colours can meet in a single character position. This becomes more of a problem when we look at high resolution graphics in Chapter Nine.

Apart from the colours 0 to 7 there are also two *pseudo colours* corresponding to the numbers 8 and 9. If you use the number 8 in any of the commands INK, PAPER, BRIGHT or FLASH the effect is that the state existing at any character position is unchanged by any more printing. For example, if PAPER 8 is carried out, all future printing uses the paper colour already at the printing location. FLASH 8 will leave the character positions that were flashing, still flashing and those that were steady, remain steady. Because in some senses the commands when used with colour 8 allow the old colour or condition to *show through* it is sometimes referred to as *transparent*.

The colour 9 can only be used with INK and PAPER and simply instructs the Spectrum to use a colour that contrasts with the colour that is present at the printing position. In practice, black is used to contrast with all the light colours (4 to 7) and white is used to contrast with all the dark colours (0 to 3). To see this try:

```
10 INK 9
20 FOR c=0 TO 7
30 PAPER c
40 PRINT c
50 NEXT c
```

Display commands in colour

All the other display commands, OVER, INVERSE, FLASH and BRIGHT work in a way that is unaffected by whatever colours are set for ink and paper. For example, INVERSE will exchange ink and paper dots no matter what colours they correspond to. One difference is that in black and white, BRIGHT seems only to affect paper dots. This is because normally paper dots are white and so can be displayed as extra bright, but ink dots are black and the idea of a brighter black is a little odd! However for the other colours, BRIGHT does have an effect. You can also use any of the commands in combination and the result is usually easy to predict. For example try:

```
10 BRIGHT 1
20 FLASH 1
30 PRINT "bright flashing"
```

If you think in terms of ink and paper dots to form the shapes of

things on the screen, and the INK and PAPER commands setting the colours that the ink and paper dots show, then you shouldn't have any trouble understanding the Spectrum's colour display. As an example of how logical everything is, consider the statement CLS introduced earlier. CLS erases the contents of the screen by filling it with paper dots so if you want to change the whole screen to one colour try:

```
1Ø INPUT "colour=";c
2Ø PAPER c
3Ø CLS
4Ø GOTO 1Ø
```

When you RUN this you should see the screen change colour instantly.

Temporary colours

It would obviously be an advantage to be able to change the colours just for the duration of a PRINT statement. You can do this by including INK or PAPER as part of the PRINT list. For example:

```
1Ø PAPER 3
2Ø INK 6
3Ø CLS
4Ø PRINT "COLOUR 6"
5Ø PRINT INK 4;"COLOUR 4"
6Ø PRINT "COLOUR 6"
```

Any of the display control commands that we have already met can be used in this way, i.e. any of INVERSE, OVER, FLASH, BRIGHT, INK or PAPER can be used as part of a 'print list' and their effect is restricted to characters printed by the PRINT statement that they are in. The same holds true for the 'print list' used in an INPUT statement. Using such commands in a PRINT statement is normally the best way to control screen displays for all but the simplest programs. Set up the overall paper and ink colours and then every time you want to print something in another colour *imbed* the colour command in a PRINT statement. In this way you always know what colours will be produced on the screen.

Using graphics in games

We are now in a position to use graphics in programming applications. However, the program using what we've learned in this chapter is held over until after we've considered sound, an indispensable adjunct to writing exciting games.

Chapter Eight

Sound and Games

The command to produce sound from the Spectrum is very simple. However, it can take quite a lot of ingenuity to produce any sounds worth listening to. In the first part of this chapter we will examine some of the ideas involved in using sound to good effect. In the second part an example of a game involving both sound and graphics will be presented and explained. Although most of this game could have been written at the end of Chapter Seven it is remarkable how much excitement can be added to a game by the careful use of sound.

Simple sounds - BEEP

The Spectrum's only sound command is:

```
BEEP 'arithmetic expression 1','arithmetic expression 2'
```

where the value of 'arithmetic expression 1' specifies the time in seconds and 'arithmetic expression 2' specifies the pitch in semitones above or below middle C. For example:

```
BEEP 1,0
```

will sound middle C for one second exactly. Notice that all computing stops for the duration that the note is sounding.

To hear the range of notes that you have at your command try:

```
10 FOR i=69 TO -60 STEP -1  
20 BEEP .1,i  
30 NEXT i
```

The quality of the very high notes is poor - more of a warbling than a steady note. The lower notes at first sound like a rasping noise and then like a series of clicks. This is not surprising as a click is the only noise that the Spectrum can really make! Steady tones are produced

as very fast streams of clicks. (Later in the chapter a method is given for making clicks from BASIC.)

If you know a little bit about music theory then you can use BEEP to write your own tunes. However, if you would first of all like to hear what the Spectrum can do on its own try:

```
10 BEEP .1,50-INT(RND*100)
20 GOTO 10
```

The noise that this program produces is interesting at first but soon becomes boring. The trouble is that music which is too random just doesn't sound interesting. Although it is very difficult to introduce enough *order* into the computer-generated music to make it sound anything like traditional music, you can see the overall effect of increasing the order if you try the following program:

```
10 LET n=0
20 LET n=n+SGN(1-2*RND)
30 BEEP .01,n
40 GOTO 20
```

This plays a long sequence of notes that go either up or down by one semitone at most and sounds just a little more like music than the first program. In fact it's not a bad imitation of the *Flight of the Bumble Bee*! Before leaving the subject of random music it is interesting to hear the effect of changing the note length in each of the above programs. Try substituting values of 1, 0.5, 0.1, 0.01 seconds in line 30.

Instead of random music you might feel that being able to *play* the Spectrum is a better idea and indeed it is not difficult to turn the Spectrum's keyboard into a musical keyboard! Try the following simple program:

```
10 DATA 0,2,4,5,7,9,11,12
20 DIM n(8)
30 FOR i=1 TO 8
40 READ n(i)
50 NEXT i
60 LET a$=INKEY$
70 IF a$="" THEN GOTO 60
80 BEEP .3,n(VAL(a$))
90 GOTO 60
```

This will allow you to play notes with the top row of number keys from 1 to 8. It works by continually *scanning* the keyboard

using INKEY\$ (see Chapter Six). Each number is entered as a string variable and is then converted from a string to a number by the VAL function and used in the BEEP statement (line 80) to control the pitch. The array 'n' holds the pitch values for each note from middle C to C an octave above. You should be able to write a program to make every key on the board produce a different note. What is more difficult is to find an arrangement of keys and notes that makes the Spectrum easy to play!

If you want to be able to hear the output of the Spectrum a little louder, then you might be interested to know that the sound signal is also available from the cassette sockets. If you simply set the tape recorder that you use to SAVE and LOAD programs to record while the Spectrum is playing something, you can rewind and play it back at a higher volume later. If you are really keen on Spectrum sounds there is nothing to stop you from connecting an amplifier to the MIC socket.

Programming tunes

Programming either well-known tunes or even tunes that you have composed is all a matter of working out the sequence of notes and their durations. This is easy if you have the tune written down. If you can find middle C on the music staff then a note drawn on this line

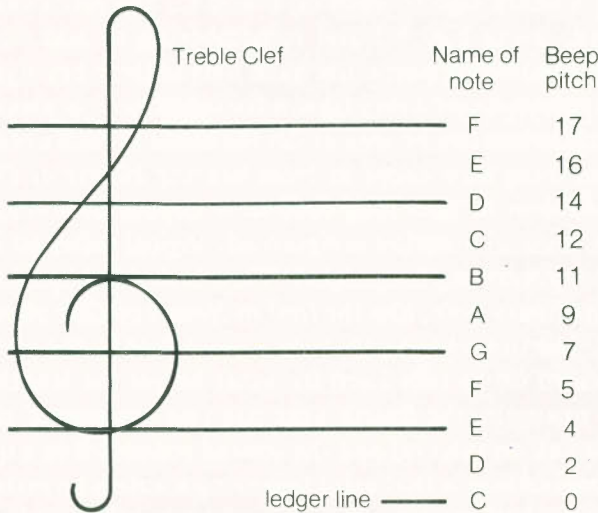


Fig. 8.1.

corresponds to a pitch of \emptyset . Moving up or down by one place on the staff increases or decreases the pitch by 2 or 1, (see Fig. 8.1). The reason for this is that notes sometimes differ by a whole tone and sometimes by only a semitone. The pattern of tone/semitone differences is easy to remember because it is exactly the same as the arrangement of black and white notes on the piano.

For example, starting from C gives the following pattern of tone/semitone differences:

C - D - E - F - G - A - B - C
T T S T T T S

An additional trouble is that most music involves sharps and flats. These are easy to deal with once you realise that a sharp raises the value of the note by one and a flat lowers it by one. For example, C is \emptyset , C sharp is 1 and C flat is -1 . The only thing that you have to remember is that if a note is shown as sharp or flat at the start of the music (i.e. in the key signature) then it and all its octaves must be sharpened or flattened.

The well-known beginning of *Hearts of Oak*, apart from being a good tune, could form the basis for a *jingle* suitable for a game involving ships. The first eleven notes can be seen in Fig. 8.2 and converting them to pitch values is easy enough. The three sharp signs (#) at the beginning apply to all Gs, Fs and Cs in the tune and this rule is best applied by writing the name of each note underneath and then writing a sharp sign by each G, F and C. The pitch values are then assigned, using Fig. 8.1 and remembering to add one for a sharp. When converting tunes that have flats in their key signatures you have to subtract one every time a flattened note is played. The resulting pitch values can be seen under the name of each note in Fig. 8.2. Only one thing now keeps us from hearing *Hearts of Oak* and this is the problem of how long each note should be sounded for. Fortunately, musical notation is rigorously logical (after all it is one of the first programming languages)! Time is divided into intervals and a plain ordinary note, like the first in *Hearts of Oak* should last one interval. The time that a note lasts is shortened by the number of *streamers* drawn on its tail. Each streamer halves the length of the note, so for example, the fourth note has two 'streamers' the first shortens it to half an interval and the second reduces it to a quarter. The only complication is that a dot following a note is an instruction to lengthen it by half the time that it would normally last (it makes you wonder how musicians cope). So the third note would normally

be one half a time interval but because it is followed by a dot it has to be sounded for one half plus one quarter, i.e. three quarters. Translating this musical notation into fractions of the time interval, gives the results written under the pitch values in Fig. 8.2. There are

The musical notation shows the melody for 'Hearts of Oak' in G major. The notes are: E, A, A, A, A, C#, B, A, G#, F#, E. Below the notes are their pitch values and time intervals.

E	A	A	A	A	C#	B	A	G#	F#	E
4	9	9	9	9	13	11	9	8	6	4
1	1	3/4	1/4	1	3/4	1/4	1	3/4	1/4	1 1/2

'Hearts of Oak'

Fig. 8.2.

two notes that do not occur in *Hearts of Oak* that have to be sounded for twice as long and four times as long. These are included in Fig. 8.3, along with all the other note values.

The time has come to start programming! Each note of the tune now has two numbers associated with it – its pitch and the time that it should sound. This information is best stored in a DATA statement and then read into two variables. Try the following:

Note	Time
	4
	2
	1
	1/2
	1/4
	1/8

Fig. 8.3. Lengths of notes


```

1Ø DATA 4,1,9,1,9,.75,9,.25,9,1,13,.75,11,.25,9,1,8,.75,6,.25,
  4,1.5,99,99
2Ø LET temp=.25
3Ø READ p,t
4Ø IF p=99 THEN STOP
5Ø BEEP t*temp,p
6Ø GOTO 3Ø

```

The DATA statement is terminated by two values of 99 and this is used to detect the end of the tune. The variable 'temp' sets the length of the fundamental time interval, in this example one quarter of a second, but you might like to experiment with other values.

Resting - PAUSE

The only thing that the BEEP command doesn't allow us to do is to pause for a period of silence. The Spectrum does have another command that will cause it to stop doing anything for a specified period of time but its format is slightly different. The command:

```
PAUSE 'arithmetic expression'
```

will stop the Spectrum from doing anything for a time given by the value of the arithmetic expression. The only trouble is that the time is measured in fiftieths of a second so:

```
PAUSE 5Ø
```

will cause the Spectrum to do nothing for one second. This isn't too much of a problem however, because:

```
PAUSE t*5Ø
```

will pause for 't' seconds.

There is one other special feature of the PAUSE command and that is that pressing any key on the keyboard will immediately cut the pause short and make the Spectrum continue with the program. If you use:

```
PAUSE Ø
```

then the pause isn't timed and the only way to make the Spectrum continue is to press a key.

Although the PAUSE command has been introduced as a way of leaving silences in music, it can be used to alter the time that any part

of a program takes to complete. The command PAUSE \emptyset is particularly useful for making the Spectrum pause until you want it to move on.

Some sound effects

The trouble with using the Spectrum to produce sound effects is that, while any noise is coming from the loudspeaker, it stops doing any calculations. So even if you write a program that makes an excellent sound of a rocket taking off it would be difficult to add any simultaneous graphics of a rocket taking off. The best you could do is to make a bit of noise, move the rocket a little, then make a bit more noise, then move the rocket a little more and so on. A continuous *woosh* accompanying a moving rocket is not something that can be achieved from BASIC alone.

However, there are a few extra noises, other than BEEPs, that can be produced from BASIC. You can produce a single click by using the following subroutine:

```
1000 LET a=PEEK 23624/8
1010 OUT 254,a-16
1020 OUT 254,a
1030 RETURN
```

The way that this works is not too important and involves an understanding of the Spectrum's hardware so don't worry, just use it! The subroutine will produce a click every time the pair of instructions OUT 254,a-16 and OUT 254,a are executed. For example, try:

```
1000 LET a=PEEK 23624/8
1010 FOR i=1 TO 20
1020 OUT 254,a-16
1030 REM
1040 OUT 254,a
1050 NEXT i
```

which will produce a low pitched rasping noise. The REM statement in line 1030 is there just to use up a little time between clicks. By increasing the number of REM statements it is possible to make a sort of machine-gun noise.

Other sound effects can be produced by combining short duration notes of different frequencies with individual clicks. The pro-

gramming methods involved aren't difficult, they just need a lot of patience and time spent trying out different combinations of all possible ideas.

Attack the saucer - the SCREEN\$ function

The game listed below uses most of the BASIC commands that have been introduced in this chapter and Chapter Seven.

```

10 LET t=15
20 LET f=0
30 LET fx=10
40 LET fy=10
50 GOSUB 1000
60 FOR x=0 TO 30
70 GOSUB 2000
80 GOSUB 3000
90 PRINT AT 20,x;" [A]"
100 NEXT x
110 FOR x=30 TO 0 STEP -1
120 GOSUB 2000
130 GOSUB 3000
140 PRINT AT 20 ,x;" [A]"
150 NEXT x
160 GOTO 60

1000 POKE USR "[A]" , BIN 00000000
1010 POKE USR "[A]" + 1, BIN 00111100
1020 POKE USR "[A]" + 2, BIN 00111100
1030 POKE USR "[A]" + 3, BIN 01111110
1040 INK 2
1050 PAPER 5
1060 CLS
1070 RETURN

2000 LET f$=INKEY$
2010 IF f$="" THEN RETURN
2020 BEEP .01,10
2030 PRINT AT fy,fx;" "
2040 LET fx=x
2050 LET fy=21
2060 LET f=1
2070 RETURN

```

```

3000 LET t=t+RND*2-1
3010 IF t<5 THEN LET t=t+1
3020 IF t>25 THEN LET t=t-1
3030 PRINT AT 5,t;" *** "
3040 IF f=0 THEN PAUSE 5:RETURN
3050 PRINT AT fy,fx;" "
3060 LET fy=fy-1
3070 PRINT AT fy,fx;"1"
3080 BEEP .001,50-fy*2
3090 IF fy<4 THEN LET f=0:PRINT AT fy,fx;"":RETURN
3100 IF SCREEN$(fy-1,fx)<>"*" THEN RETURN
3110 GOSUB 4000
3120 RETURN

4000 PRINT AT 5,t+1;FLASH 1;" *** "
4010 BEEP .01,40
4020 LET a=PEEK 23624/8
4030 FOR i=1 TO 20
4040 OUT 254,a-16
4050 OUT 254,a
4060 NEXT i
4070 RETURN

```

The game itself is relatively straightforward to play. An alien 'flying saucer', in the form of three asterisks, moves rather jerkily across the screen. A ship, whose purpose is to attack the alien, moves rapidly backwards and forwards at the bottom of the screen. A missile can be launched at any time from the attacking ship by pressing any key. Once a missile has been fired it moves up the screen accompanied by a whistling noise, increasing in pitch, until it either misses the saucer or hits it with a resulting explosion. If at any time during the flight of a missile another key is pressed, then the first missile is erased from the screen and a new missile fired at the saucer.

The program has been written as a small collection of subroutines and is not particularly difficult to understand. It is easier to follow the main part of the program after a description of each subroutine. Subroutine 1000-1070 sets up the user-defined graphics character for the attacking ship (lines 1000-1030) and sets up the overall ink and paper colours (lines 1040-1070). Subroutine 2000-2070 checks to see if any key has been pressed and fires the missile. If no key has been pressed then control is returned to the main part of the program (line

2010). If any key has been pressed then any existing missile is removed from the screen by printing a blank (line 2030) at the current missile position stored in 'fy' and 'fx'. Then the current missile position is set to the current position of the attacking ship (lines 2040-2050) and variable 'f' is set to 1 (line 2060) to indicate that a missile has been fired and is in flight. Subroutine 3000-3120 moves the saucer a random amount to the right or left, prints the missile if one is in flight and checks to see if it has hit the saucer. Lines 3000-3030 are responsible for moving the saucer. Notice the checks to stop it from moving off the edge of the screen in lines 3010 and 3020. The printing of the saucer in line 3030 also serves to remove the old saucer from the screen because of the blanks included at either end of the string of asterisks. Lines 3040-3080 look after moving the missile. Line 3040 checks to see if there is a missile in flight (i.e. f=1). If there isn't, control is passed back to the main part of the program. The PAUSE is included to make the attack ship move at the same rate even if there isn't a missile in flight. Lines 3050-3080 move the missile up the screen by one line. Line 3050 blanks out the old missile and line 3070 prints it at its new position. Line 3080 makes a sound that increases in pitch as the missile moves higher up the screen. Lines 3090-3110 test to see if the missile has hit or missed the saucer. Line 3090 tests to see if the missile's position is such that it has passed the saucer and is about to go off the screen. If this is the case a blank is printed to remove the missile and the variable 'f' is set to zero to indicate that there are no missiles in flight. Line 3100 uses the SCREEN\$ function to discover what character is at the next screen location that the missile will move into. The SCREEN\$ function hasn't been discussed so far but it is very easy to understand how it works. The function SCREEN\$(y,x) returns the character displayed on the screen to column x, line y. It is used in line 3100 to discover if the character just above the missile is an asterisk. If it is, then the missile is about to hit the saucer and the explosion subroutine is called.

Subroutine 4000-4070 is the explosion subroutine. Line 4000 prints the saucer flashing to indicate an explosion on the screen. Lines (4020-4060) make the rasping noise described in the last section to stand in for the sound of a real explosion.

Now that all the subroutines have been described, the working of the main part of the program is easy to understand. Lines 10-40 set up values of some of the important variables - 't' is the horizontal position of the saucer, 'f' is used to indicate when a missile has been fired and 'fx' and 'fy' are the co-ordinates of the missile. Then

subroutine 1000 is called to set up the user-defined graphics and the colours used. The main work of the program is done by the two FOR loops 60-100 and 110-150. The first FOR loop moves the attack ship to the right one place at a time. Each time the attack ship moves subroutine 2000 is called to check for a 'fire missile' command and subroutine 3000 is called to move the saucer and the missile. The second FOR loop moves the attack ship to the left but otherwise it is identical to the first FOR loop.

This concludes the description of this short games program. If you study it to the point where you are sure that you understand it, the way to find out if you're right is to try to modify it! The game would be made much more exciting by the addition of only a few very simple features. You could, for example, add a routine to keep a score of the number of saucers hit, or give the attacking ship only a limited number of moves before the saucer fires a missile back at it! Try experimenting with these suggestions and your own ideas. After all, the only way to learn to program is to program!

Chapter Nine

High-resolution Graphics

The presence of high-resolution graphics (hi-res graphics) is something that might have lured you into buying your Spectrum. However, as Chapters Seven and Eight might have convinced you, hi-res graphics is not really needed for most applications. Drawing things in hi-res graphics generally takes longer and in most cases you are limited to two colours. Having placed hi-res graphics in context, it has been said that the Spectrum's commands which deal with this facility are not difficult to understand and use, once you have mastered the ideas behind low-resolution graphics, and that some of the effects that can be achieved are very good indeed.

The high-resolution screen

The Spectrum uses the same method for delaying high- and low-resolution graphics. This is not the case with other microcomputers and, as it means that you can mix text with hi-res graphics, it is to the Spectrum's advantage!

The way that hi-res works is easy to understand in terms of the eight-by-eight square of dots that makes up each character location on the screen. Low-resolution commands can only alter entire eight-by-eight blocks at a time but hi-res commands can change as little as a single dot in any character location. Notice that this implies that all the *rules* for displaying a dot in a character location remain the same. For example, the colour that a dot is displayed in still depends on whether it is an ink or paper dot and what INK or PAPER command has been given. The hi-res commands only increase the ways that we can change the dots in a character location, not the colour or any other attributes of a character location.

Obviously, if the hi-res graphics commands can be used to change single dots there must be some way of specifying which point. Now,

theoretically you could do this by using the existing numbering of the character positions and refer to individual dots as, say, the third dot in a particular character, but it turns out that it is much easier to have a completely new numbering system for the dots. As there are 32 characters to a line and each character is eight dots wide there are a total of 32×8 , or 256, dots to a line. As there are 22 lines and each line is eight dots high there are 22×8 or 176 dots vertically. Thus the hi-res screen is composed of 256 dots horizontally and 176 dots vertically and any single dot can be picked out by stating which column and row it is in. The columns are numbered from zero starting at the far left, and so the column number, or *x co-ordinate* as it is called, ranges from 0 to 255. The rows are numbered from zero starting at the bottom of the screen and so the row number, or *y co-ordinate* as it is called, ranges from 0 to 175. (Notice that the row numbers go from the bottom to the top unlike the line numbering which starts at the top.) Any dot on the screen can be specified by giving two numbers, its *x co-ordinate* and its *y co-ordinate*. It is usual to write these two numbers in brackets with the *x co-ordinate* first. So (0,0) is the bottom left hand corner and (255,175) is the top right hand corner. After a little practise, using *x,y co-ordinates* will become second nature.

The graphics commands - PLOT, DRAW and CIRCLE

The hi-res graphics commands are easier to understand when they are working with only two colours and so this is where we'll begin! Through all of the following discussion it is assumed that either you haven't altered the initial setting of black ink and white paper on your Spectrum, or that you have set up two reasonably contrasting colours using INK and PAPER.

The simplest hi-res command is:

```
PLOT 'arithmetic expression 1','arithmetic expression 2'
```

This changes the dot at the *x co-ordinate* given by 'arithmetic expression 1' and the *y co-ordinate* given by 'arithmetic expression 2' to an ink dot. Try the following program, both to discover how PLOT works and to investigate the way *x,y co-ordinates* work:

```
10 INPUT "x=";x;" y=";y
20 PLOT x,y
30 GOTO 10
```


If you enter a value for x and y that takes the point outside the screen you will get an "Integer out of range" error. For an automatic demonstration try:

```
1Ø LET x=INT(RND*256)
2Ø LET y=INT(RND*176)
3Ø PLOT x,y
4Ø GOTO 1Ø
```

The most useful of the hi-res commands is:

DRAW 'arithmetic expression 1','arithmetic expression 2'

which produces a straight line. The starting position of the line is where the last PLOT or DRAW finished and its end is 'arithmetic expression 1' to the right and 'arithmetic expression 2' up. For example, if the last PLOT was PLOT Ø,Ø the command:

```
DRAW 1ØØ,1ØØ
```

will produce a line from (Ø,Ø) to (1ØØ,1ØØ). But if the last PLOT was PLOT 5Ø,5Ø the line would have started at (5Ø,5Ø) and ended at (15Ø,15Ø). It is important to notice that DRAW uses co-ordinates in a way that is completely different from PLOT. When using hi-res graphics commands, you can imagine that there is a graphics *cursor* that moves around the screen with the current graphics position in much the same way that the test cursor moves around the screen with the current printing position. The command PLOT x,y moves the graphics cursor to the point (x,y) and then plots a point. The command DRAW x,y moves the graphics cursor x units horizontally and y units vertically and then draws a line between the old position of the cursor and the new. Commands such as RUN, CLEAR, CLS and NEW reset the graphics cursor to the point (Ø,Ø). After this the graphics cursor is moved around the screen by each graphics command. The clearest indication that DRAW x,y is different from PLOT x,y is in commands such as:

```
DRAW -1Ø,1Ø
```

which leaves the graphics cursor 10 units to the left and 10 units up. Negative co-ordinates are not allowed in PLOT!

The form of the DRAW command is often very convenient but it can be difficult to draw a line between two given points. However, the following combination will draw a line between the point (x1,y1) and (x2,y2):

PLOT x1,y1:DRAW x2-x1,y2-y1

To see the sort of thing that DRAW can do, try the following program:

```

10 FOR i=1 TO 175 STEP 4
20 PLOT 0,i
30 DRAW 255-i,-i
40 PLOT i,0
50 DRAW 255-i,i
60 PLOT 0,i-175
70 DRAW 255-i,i
80 PLOT i,175
90 DRAW 255-i,-i
100 NEXT i
    
```

95 - border 5
 96 - Pause 2
 97 - border 2
 98 - Pause 4
 99 - border 3

works well.

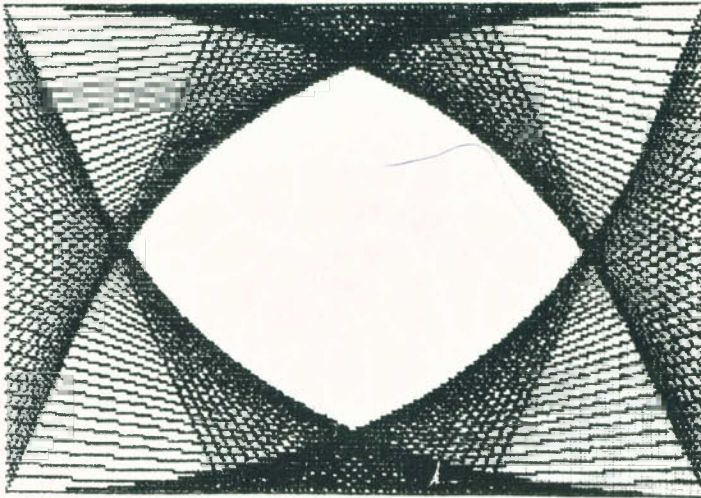


Fig. 9.1. String pattern program ✓

The output from this program is reproduced in Fig. 9.1. You will find that you get different effects by altering the values of the STEP in line 10. Although we haven't quite exhausted the full extent of the DRAW command, its additional feature is easier to understand after a discussion of the final hi-res graphics command.

The command:

```

CIRCLE 'arithmetic 'arithmetic 'arithmetic
        expression 1', expression 2', expression 3'
    
```

will result in a circle centered on the point (x,y), where x and y are the

values 'of the first two arithmetic expressions, and with a radius determined by 'arithmetic expression 3' being drawn on the screen. For example:

```
CIRCLE 100,50,40
```

draws a circle centered at (100,50) and radius 40. As an example of the circle command the following program draws random circles, as illustrated in Fig. 9.2:

```
10 LET r=RND*50
20 LET x=r+RND*(255-2*r)
30 LET y=r+RND*(175-2*r)
40 CIRCLE x,y,r
50 GOTO 10
```

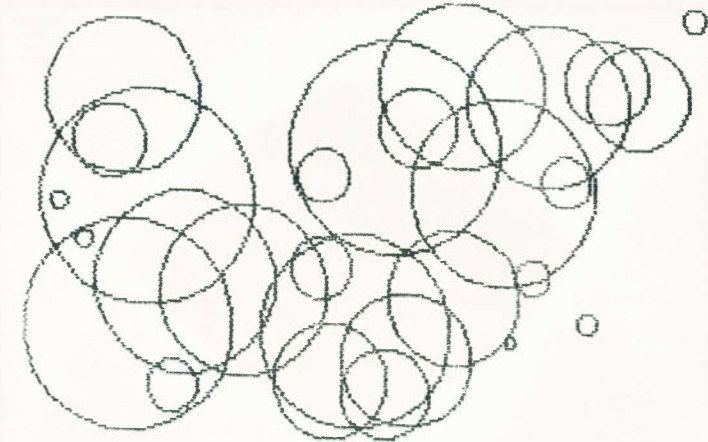


Fig. 9.2. Random circle program

Notice that when drawing random circles you have to be careful not to go off the edge of the screen. This is allowed for in this program by the inclusion of $-2*r$ in lines 20 and 30.

Now that the CIRCLE command has been explained, the additional feature of the DRAW command mentioned earlier can be dealt with. In addition to being able to produce straight lines between two points, DRAW can be used to produce parts of circles between two points. The general form of the DRAW command is:

```
DRAW      'arithmetic   'arithmetic   'arithmetic
          expression 1', expression 2', expression 3'
```

The meaning of the first two arithmetic expressions has already been explained and is unaltered by the existence of the third. However, the third parameter specifies an angle that is used to determine how much of a circle is drawn. To understand how an angle can specify how much of a circle is drawn, all you have to do is to imagine that you are standing at the centre of a circle. If you hold out your arms with the given angle, then the part of the circle that is between your arms is the part of the circle produced by DRAW. For example, 180 degrees specifies half a circle and 90 degrees a quarter. The only complication is that the Spectrum measures angles in radians rather than degrees. To convert from degrees to radians all you have to do is multiply by 180 and divide by $\text{PI}(\pi)$. This results in an angle of PI specifying a semicircle, an angle of $\text{PI}/2$ specifying a quarter circle and so on. As an example of the DRAW command try:

```
10 PLOT 100,100:DRAW 50,50,PI/2
```

The PLOT first moves the graphics cursor to (100,100) and then DRAW command produces a quarter of a circle starting at (100,100) and ending at (150,150). Notice that the part of a circle that is produced by a DRAW command is determined solely by the value of the third parameter. The size and orientation of the part of the circle is governed by both the current position of the graphics cursor and the values of the first two parameters.

You may have noticed that, unlike a straight line, there are two circular arcs between any two points. For example, if you imagine two points on a horizontal line, then you can draw a semicircle from the one on the left to the one on the right going either clockwise or anti-clockwise. The clockwise semicircle would be above the horizontal line and the anti-clockwise semicircle would be below it. This description in terms of clockwise and anti-clockwise circles is exactly how the Spectrum solves the problem of which circle to

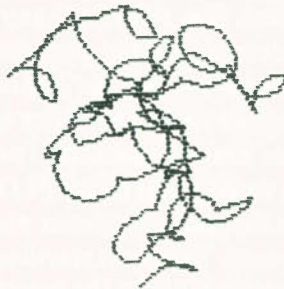


Fig. 9.3. Tangled string program

draw. Positive angles result in circular arcs being drawn between two points in an anti-clockwise direction and negative angles result in clockwise arcs.

As an example of the complete DRAW command try the following program that produces an output that can only be described as tangled string (see Fig. 9.3):

```

10 PLOT 100,100
20 LET x=25-RND*50
30 LET y=25-RND*50
40 LET p=PI*RND
50 DRAW x,y,p
60 GOTO 20

```

Line 10 sets the graphics cursor roughly to the middle of the screen and lines 20 to 30 set random values for the three parameters of DRAW. The random angle is set in line 40 as a fraction of PI so the DRAW command in line 50 draws fractions of semicircles. If you leave this program to run long enough it will stop with an error message when one of the arcs finally goes outside the screen area. In general, it is quite difficult to detect whether a circular arc produced by DRAW will go off the screen – until it actually happens, that is!

High-resolution colours

You can use all of the colour and attribute commands such as INK, PAPER and OVER, that were introduced in Chapter Seven, to control the way the high-resolution dots are displayed. For example, if you want to produce blue dots on a yellow background all you have to do is to put INK 1:PAPER 6 before any PLOT, CIRCLE, or DRAW commands. You can even embed colour commands within hi-res graphics commands in the same way as for the PRINT command. In this case the effect of any colour commands is temporary and only determines the way dots are produced by the command that they are embedded in. The only thing that you have to remember is that colours and attributes (apart from OVER and INVERSE) apply to entire character printing positions. For example, if you plot a single point using PLOT INK 4;100,100, then *all* of the ink points in the same character position as (100,100) will change to colour 4 (green). The most startling illustration of the way INK and PAPER commands affect the entire character block can be seen by DRAWing a line using a different PAPER colour to the rest

of the screen. For example:

```
1Ø PAPER 6
2Ø CLS
3Ø DRAW PAPER 1;INK 4;1ØØ,1ØØ
```

produces a very odd display because each point plotted changes all the paper dots in the character position that it falls in to colour 1 (blue). If there were any ink dots of another colour in the character position these would be changed to green.

The fact that colours (including BRIGHT and FLASH) apply to complete character positions places a severe restriction on the way hi-res graphics can use colour. If you DRAW two lines with different ink colours, then there is no problem unless they meet in a single character position. When this happens, the first line produced changes its colour in the character position where it meets the second line, to the colour of the second line. The rule is that you can only display two colours in any character position – the ink colour and the paper colour. As long as hi-res lines and circles, etc., keep to their own areas of the screen they can be any of the eight colours but if any of them share the same character position they must have the same ink *and* paper colour. For some applications this restriction is not important because lines of different colours naturally occupy different areas of the screen. If this isn't the case, however, there is nothing that can be done to overcome this limitation of the Spectrum's hi-res graphics. To avoid any such problems the only thing that you can do is to treat the hi-res graphics screen as a two-colour display.

Un-plotting – OVER and INVERSE

Although the colour commands, INK, PAPER, FLASH and BRIGHT affect all of the dots in a character square, the commands OVER and INVERSE merely affect the way each dot is produced on the screen. This means that OVER and INVERSE are true hi-res commands that can be used within PLOT, DRAW or CIRCLE without affecting anything else already on the screen. For example, INVERSE 1 results in a paper dot being produced instead of ink dots. So the program:

```
1Ø CIRCLE INVERSE Ø;1ØØ,1ØØ,5Ø
2Ø CIRCLE INVERSE 1;1ØØ,1ØØ,5Ø
3Ø GOTO 1Ø
```

first produces a circle of ink dots and then erases it by plotting paper dots in the same place.

The command **OVER** is more difficult to understand than **INVERSE** because its effect depends on what is already present at the plotting position. Recalling the rules given for how **OVER** works in Chapter Seven, you should be able to see that a line of dots plotted in hi-res graphics following **OVER 1** will appear as ink dots, if they are replacing paper dots, but as paper dots, if they are replacing ink dots. In practice, this is very useful because it gives us a way of plotting and un-plotting hi-res dots without losing any dots that are already on the screen. To see this in operation, try:

```
1Ø CIRCLE 1ØØ,1ØØ,5Ø
2Ø PLOT Ø,1ØØ
3Ø DRAW OVER 1;255,Ø
4Ø PAUSE 2Ø
5Ø GOTO 2Ø
```

The first **DRAW** command produces a line of ink dots passing through the circle plotted by line 1Ø, except where the line cuts through the circle itself where, by the rules given above, paper dots result. After the pause generated by line 4Ø so that you can look at the result, the **PLOT** and **DRAW** commands are carried out again. This time the ink dots in the line are changed to paper and the line vanishes, except for the paper dots where the line cut the circle which are restored to their original ink value! This appearing and disappearing of the line continues until you break into the program. So if you want to produce some temporary lines or circles, produce them using **over 1** and then remove them by a second use of **OVER 1**.

It is worth summarising the use of **OVER** and **INVERSE** in hi-res commands using **PLOT** as an example:

PLOT	produces an ink dot.
PLOT INVERSE 1	produces a paper dot.
PLOT OVER 1	changes the colour of the dot already present to the opposite colour, i.e. ink to paper and paper to ink.
PLOT INVERSE 1;OVER 1	has no effect at all apart from moving the graphics cursor.

Finding out what's on the screen - POINT

In the same way that **SCREEN\$** could be used to discover what

character is displayed on the screen at any character location, the function POINT can be used to find out what sort of dot is on the screen at any given hi-res position. The general form of the POINT function is:

POINT ('arithmetic expression 1', 'arithmetic expression 2')

where 'arithmetic expression 1' is the x co-ordinate and 'arithmetic expression 2' is the y co-ordinate. Notice that, unlike most functions on the Spectrum, a pair of brackets *is* necessary. The value returned by POINT is 1 if there is an ink dot at the specified co-ordinates and a \emptyset if there is a paper dot. The sort of thing that POINT is used for is similar to the use of SCREEN\$ in the saucer program given in Chapter Eight.

Using hi-res graphics

Although there are only a few hi-res graphics commands, it can be difficult to see how they can be used to produce effective displays. As always, the best way to learn is to have a look at some examples and then try to write your own programs.

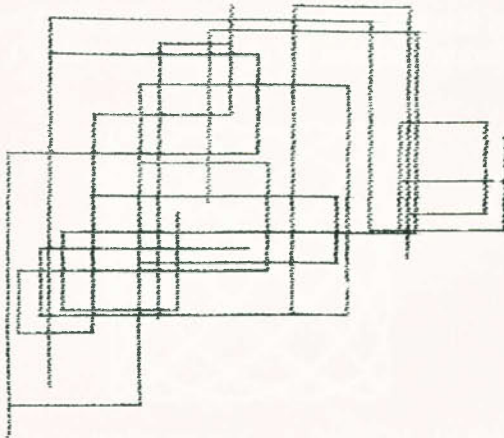


Fig. 9.4. Etch-a-sketch program

The first example uses the INKEY\$ function to control the position of a dot on the screen. By pressing the appropriate arrow keys you can draw shapes on the screen:


```

10 LET x=127
20 LET y=80
30 LET a$=INKEY$
40 IF a$="5" THEN LET x=x-1
50 IF a$="8" THEN LET x=x+1
60 IF a$="6" THEN LET y=y-1
70 IF a$="7" THEN LET y=y+1
80 PLOT x,y
90 GOTO 30

```

you can see a sample of the output of this program in Fig. 9.4. You could try to add some improvements of your own such as diagonal movements and being able to move from one place to another without drawing a line.

The second example is based on a set of patterns discovered by the nineteenth century French physicist, Lissajous and aptly called 'Lissajous figures':

```

10 LET t=0
20 LET t=t+0.1
30 LET x=50*(1+SIN(1.1*t))
40 LET y=50*(1+COS t)
50 PLOT x+50,y+50
60 GOTO 20

```

The output of this program can be seen in Fig. 9.5. You can produce a range of different patterns by changing the value 1.1 in line 30.

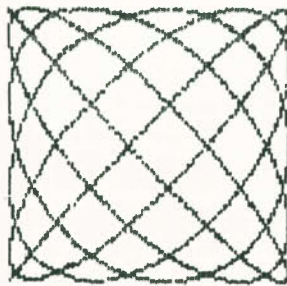


Fig. 9.5. Lissajous figure

The final example in this chapter is a program that plots the graph of $\text{SIN}(x)/x$. This produces a particularly interesting shape, as you can see in Fig.9.6. One difficulty to be aware of is that $\text{SIN}(x)/x$ is

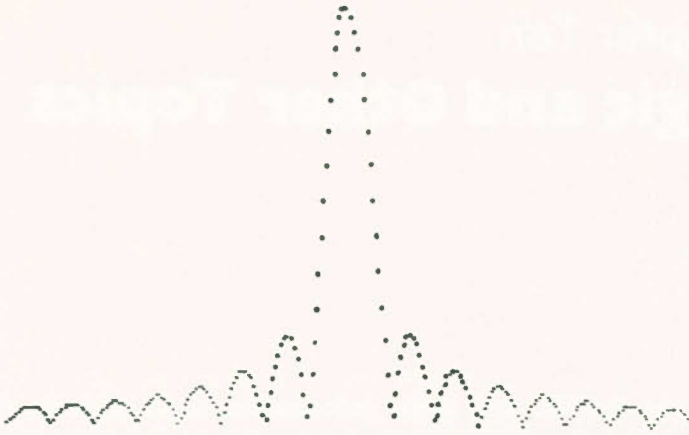


Fig. 9.6 $\frac{\text{SIN } x}{x}$ program

impossible to work out when x is zero so this point has to be carefully left out of the graph.

```

10 FOR i=1 TO 255
20 IF i=127 THEN GOTO 60
30 LET x=(i-127)/5
40 LET y=150*SIN(x)/x
50 PLOT i,y
60 NEXT i

```

When using graphics in your own programs, the best strategy is actually to limit your use of high-resolution graphics to those occasions when they perform an essential function. In other words, it is advisable to start off using low resolution graphics but to keep an eye open for situations where high-resolution graphics actually do the job better.

Chapter Ten

Logic and Other Topics

Using the BASIC and other information dealt with in earlier chapters, you should by now find it possible to write any program that you want to. However, there are facilities on the Spectrum that, although not entirely necessary, do make things easier or go beyond what can be done from simple BASIC. This chapter collects together these extras and explains a little of how they work.

The first topic to be covered has the rather daunting title of *logic* and introduces the commands AND, OR, and NOT. The second area deals with the commands PEEK, POKE, IN, OUT,USR and CLEAR which affect the inner working of the Spectrum. Thirdly, we consider the command ATTR which can be used to discover the colour of any point on the screen and finally we look at one of the simplest of BASIC statements, REM. It would be misleading to think of this as a collection of unimportant topics just because they have not figured so far in this book. Rather it is a case of them being introduced last but not least.

Logic and the conditional expressions

In everyday speech we often say things like, 'did you buy apples and oranges?', 'do you prefer tea or coffee?'. The use of words like *and* and *or* are so common that we rarely stop to think about them. It would obviously be a great advantage if the use of *and* and *or* could be extended to BASIC conditional expressions. Luckily, most versions of BASIC do allow the use of these everyday concepts and in Spectrum BASIC you can write expressions such as:

```
a<0 AND b=3  
a<0 OR b=3
```

The meaning of each of these expressions is in line with the usual

English meaning of *and* and *or*. The first of the above expressions is true if both of the conditions 'a<0' 'b=3' are true and the second expression is true if either of the two conditions is true. As well as AND and OR the Spectrum also allows the use of NOT, which simply changes the value of a conditional expression from true to false and vice versa. For example, '3=2' is false but 'NOT 3=2' is true.

You can combine AND, OR and NOT with any of the conditions we met in Chapter Four to make more complicated expressions that evaluate to one of the values *true* or *false*. (As you already know, from Chapter Four, *true* is represented by 1 and *false* by 0 in Spectrum BASIC which allows you to use them very easily in arithmetic expressions.) Conditional expressions that include AND, OR or NOT are usually called *logical expressions* and we can now re-write the definition of the IF statement as:

IF 'logical expression' THEN 'BASIC statement'

For example, if you want to check that you're not about to use an x co-ordinate that goes outside the screen area, you could use:

IF x<0 OR x>255 THEN ...

in place of the two IF statements that would be required without the use of OR.

Forming logical expressions to test for *overall* conditions is usually straightforward. However, there are a few traps that even experts fall into. If you want to translate the English statement 'a equals b and c' then you must repeat the condition '='. In other words, you must write:

a=b AND a=c

and not use:

a=b AND c

which will give a result that depends on whether c is 0 or 1. You should also be careful when using NOT. For example:

NOT(a=b AND a=c)

isn't the same as:

NOT(a=b) AND NOT(a=c)

To see that this is the case try working the two expressions out for a few values of 'a', 'b' and 'c'. The moral is that you should always

beware of using logical expressions without thinking about *exactly* what they mean.

To close the subject of logical expressions it is worth introducing the idea of a *truth table*. If you consider the logical expression:

a AND b

where 'a' and 'b' are variables that are either \emptyset , for false, or 1, for true. You can draw up a table that lists the result of the expression for all possible values of 'a' and 'b' as follows:

a	b	a AND b
\emptyset	\emptyset	\emptyset
\emptyset	1	\emptyset
1	\emptyset	\emptyset
1	1	1

Such a table is called a truth table because it lists the conditions under which the logical expression is true or false. You can draw up truth tables for any logical expression and this is one way to check that you understand what is happening. For example, OR and NOT have the following truth tables:

a	b	a OR b	NOT a
\emptyset	\emptyset	\emptyset	1
\emptyset	1	1	1
1	\emptyset	1	\emptyset
1	1	1	\emptyset

The OR that we have used so far is not entirely equivalent to the English word 'or'. Most uses of the English 'or' mean 'one or the other but not both'. For example, 'You can have jam or marmalade' means that you can pick one but not (normally) both! However, the logical OR means that you can have either one or both (look at the truth table if you are unsure of this). The logical OR is more properly called the *inclusive OR* because it includes the possibility of both. The usual English 'or' is known as the *exclusive or* because it excludes the possibility of both. You can make up a logical expression that is equivalent to the exclusive or:

exclusive or = (NOT(a) AND b) OR (a AND NOT(b))

as can be seen from the truth table:

a	b	(NOT(a) AND b) OR (a AND NOT(b))
0	0	0
0	1	1
1	0	1
1	1	0

This pattern of 0 (false) and 1 (true) may perhaps remind you of the pattern of ink and paper dots in Chapter Seven where the OVER command was discussed. In fact, following OVER 1 the dot produced on the screen is the exclusive or of the old dot and the dot that you are trying to plot. Once you start looking out for it, you'll notice that logic crops up in some strange places!

Inside the Spectrum - BIN, PEEK, POKE, IN, OUT,USR and CLEAR

It may come as something of a surprise to learn that BASIC includes a number of commands that allow access to the inner workings of the Spectrum. The reason why this might seem strange is that all the BASIC that we have looked at so far has done its best to avoid getting involved with details of how the machine carries out commands. However, there are some applications where the normal instructions of BASIC are in some way deficient. For example, they might be too slow or fail to take account of some important feature of the machine. To allow the programme to find a way around such difficulties, most versions of BASIC include instructions that allow you to gain access to the inside of the machine.

We have already met the function BIN in Chapter Seven, where it was used without explanation in the construction of user-defined characters. Although, from the point of view of the BASIC programmer, the Spectrum seems to do its arithmetic in terms of decimal numbers, in fact it works things out in a more fundamental system called *binary*. Humans count in decimal simply because they have ten fingers. If we had only two fingers then we would count in the same way that computers do, in binary. Although it is not important for anyone to know very much about binary, because BASIC takes care of converting decimal to binary and back again, it is important if you ever want to use your Spectrum directly without the help of BASIC. Even then, all you really need to know is that the function BIN will take a binary number and convert it to decimal. A binary number is simply a number that contains only zeros and ones.

you don't really have to know any more than that because BASIC and BIN will look after you. For example, try:

```
PRINT BIN 'number'
```

which will print the decimal equivalent of any binary number that you care to enter. You will find that 11 is 3, 111 is 7 and 11111111 (eight ones) is 225. (Notice that BIN, unlike all the other Spectrum functions won't allow you to use an expression.)

We learned very early on that a variable is a named area of computer memory. However, it is sometimes necessary to side-step this method of using computer memory and use in preference direct access, via PEEK and POKE. PEEK is a function that will return the contents of a memory location and POKE is a command that will alter the contents of a memory location. It is as simple as that, except that you need to know how to specify which memory location and what sort of number can be stored in a memory location. The first problem is easily solved because the Spectrum, like all computers, numbers all its memory locations sequentially starting from zero. So PEEK(543) will return the contents of the five hundred and forty-third memory location. The second problem is also easily solved once you know that a memory location can store a binary number with up to eight zeros or ones in it and as BIN 11111111 evaluates to 255 this is the largest number that can be held in a single memory location. So POKE 10000,200 will store 200 in memory location 10000. However, POKE 10000,600 will give an error message because 600 is greater than 255. In general, to use PEEK and POKE you have to have a knowledge of what is stored where inside your Spectrum and this is often not easy to find out. For this reason, PEEKing and POKEing are best avoided unless you are absolutely sure that you know what you are doing.

However, there are one or two standard applications of PEEK and POKE that are worth knowing about and also demonstrate the typical way that PEEK and POKE are used. The Spectrum has a *clock* that ticks in fiftieths of a second buried deep inside it. From the programming point of view, it looks like three memory locations that continuously change the values stored in them. The three memory locations work together to extend the range of time beyond what can be held in one memory location. The memory location at 23672 counts fiftieths of a second, and as the largest number that can be stored in a memory location is 255 it counts 255 fiftieths of a second and then goes back to zero. You can think of this as a hand on a clock going round every 256 ticks. The second memory location, at

23673, counts how many times the first memory location has, as it were, gone round and so counts in units of 256 fiftieths of a second. In the same way, the third memory location, at 23674, counts how many times the second memory location has gone round and so it counts in units of 256*256 fiftieths of a second. You can make use of this information by using the PEEK function to discover what is in each memory location and converting it to a number that represents a time in seconds. For example:

$$(65536 * \text{PEEK } 23674 + 256 * \text{PEEK } 23673 + \text{PEEK } 23672) / 50$$

gives you the time in seconds since the clock was started, normally when you switched on. In the same way, you can use POKE to set the three locations to any time that you desire. For example, to zero the clock use:

$$\text{POKE } 23674,0:\text{POKE } 23673,0:\text{POKE } 23672,0$$

Most applications of PEEK and POKE are similar in that they require you to know something about where the Spectrum stores some piece of information that is normally hidden from you. For example, once you know that location 23692 is used to store the number of lines that will be scrolled up the screen before the familiar "Scroll? y/n" message appears on the screen, then you can use POKE 23692,255 to ensure that the message doesn't appear for 255 scrolls.

In the same way that PEEK and POKE examine and alter the contents of memory, IN and OUT can be used to communicate with any external devices connected to the Spectrum, such as with the ZX printer. However, unless you have a very special application there is no real call to become involved in IN and OUT. You can see an example of IN and OUT in the short subroutine that was used to produce a click in Chapter Eight. From the Spectrum's point of view, its loudspeaker is an *external* device and this is why it requires these commands to be used to produce sounds.

We have already met the USR function in connection with user-defined characters. However, this use is very special and in general, the USR function transfers control out of BASIC and into a *machine code* program stored somewhere inside the Spectrum. Machine code is a completely new, and vast, topic and until you want to get involved in it the USR function will be of little interest to you.

The final instruction to be mentioned in this section is similarly in the province of machine code programming. The command CLEAR can be used to reserve some memory for storing machine code programs

so that the Spectrum doesn't allocate the area to variable storage. Because of this specialised use, you are unlikely to come across it very often.

Finding out the colour - ATTR

In the same way that the function SCREEN\$ and POINT can be used to find what is on the screen, the ATTR function can be used to find out what colours are being used for ink and paper and whether a particular character location is flashing or bright. The only trouble is that ATTR returns all of this information in the form of a single number that has to be *decoded* to find out what it means. However, it is possible to list expressions that will extract each piece of information:

$\text{INT}(\text{ATTR}(\text{line}, \text{col}) / 128)$

is 0 if the location is steady and 1 if the location is flashing and:

$\text{INT}((\text{ATTR}(\text{line}, \text{col}) - \text{INT}(\text{ATTR}(\text{line}, \text{col}) / 128) * 128) / 64)$

is 0 if the location is bright and 1 if the location is not bright. Similarly:

$\text{INT}((\text{ATTR}(\text{line}, \text{col}) - \text{INT}(\text{ATTR}(\text{line}, \text{col}) / 64) * 64) / 8)$

gives the code for the paper colour and:

$\text{ATTR}(\text{line}, \text{col}) - \text{INT}(\text{ATTR}(\text{line}, \text{col}) / 8) * 8$

gives the code for the ink colour at the locations specified.

REMark and good programming

The BASIC command REM is the simplest of all in that it does absolutely nothing! Its only purpose is to allow you to include comments that are not part of a program. For example, you could include in every program that you write a first line that tells you what the program is called:

10 REM Title of program

and the REM would alert the Spectrum to the fact that what followed wasn't to be taken as a line of BASIC for it to pay attention to, but as a note to any humans that might read the program.

You might think it strange that such a simple command has been left to the last chapter of a book on BASIC. The reason for this is that although REM is a simple command it can be used to very good effect in writing clear programs. After you have got over the initial difficulty of writing programs in BASIC you should look for ways of writing better programs. At first a program that works is a reward in itself but later on a well-written program is what you should aim for. What constitutes a well-written program is something that you will discover for yourself as you learn programming by trial and error and by reading other people's efforts.

The REM statement is part of better programming in that, while it certainly isn't necessary, it does help to make your programs easier to understand. It is a good idea to include REMs that explain what is happening in each section of your program as you write it. Good explanations will help you to understand your own programs more quickly when you return to them at a later date and will assist other people, to whom you may give them, to grasp what you intend each stage to do.

Where next?

As I have already said many times in this book, the real route to learning programming is to write programs. Certainly, books can help you but only if you are prepared to experiment on your own behalf. Don't be worried if your first programs don't attempt anything very ambitious. It is better to try out your ideas in short and simple routines at first. If you try anything too complicated there is much more chance that you'll make mistakes that you can't locate. Try writing program snippets that do just a few things at a time – if you look back through this book you'll find lots of such examples. When all your mini-programs work then it is time to start putting them together to build more extensive ones. The main thing is to go ahead and to have some fun with your Spectrum.

Further Reading

Once you've read this book you should be reasonably proficient at writing programs but in some ways that's just the beginning. Hopefully this book will leave you wanting to learn more – about BASIC, about the Spectrum and about computers generally. Lots of books are being published all the time on these subjects so all I can hope to do here is to mention some that I know about that I think you might like to look out for.

As far as BASIC is concerned, I'll include a title that starts where this book ends, *The Complete Programmer* by Mike James will be published by Granada in 1983. Once you've mastered the rudiments of BASIC you can also learn a lot from books of other people's programs. They can be used as a source of ideas and inspiration. They are particularly useful if they include programming details and helpful hints. Mike James, Kay Ewbank and I have collaborated to write just that sort of book. It's *The Spectrum Book of Games* and it contains some games that we found a challenge to program and fun to play. It too is published by Granada.

Turning to the Spectrum, if you've not already read it, look out for *The ZX Spectrum and how to get the most from it* by Ian Sinclair (published by Granada in 1982).

You may also be interested in finding out more about the history of computers and their impact on our day-to-day lives. If so, I can recommend *Introducing Computers* by Ron Condon, a Macdonalds Guidelines book and *The Making of the Micro* by Christopher Evans, published by Gollanz in 1981.

If you want to keep up-to-date with what is happening in the world of microcomputers, probably the best way is through the magazines. The monthly magazine *Computing Today* is one that I read – and write for – and I also recommend *ZX Computing* which is published every two months. Both these should be easy to obtain from your newsagent or can be bought on subscription.

Index

- ABS, 73
- Alphabetical order, 62
- AND, 130-33
- Apostrophe, 87-8
- Arccosine, 76
- Arcsine, 76
- Arctangent, 76
- Arithmetic expression, 26, 43
- Arithmetic functions, 73-5
- Arithmetic operators, 26-28
- Array, 64-7
- Arrow keys, 13
- AT, 88-90
- ATTR, 130, 136

- Backspace, 13
- BEEP, 15, 107-109
- BIN, 97-8, 133
- Binary, 133
- BORDER, 102
- Brackets, 28
- BREAK, 14
- BRIGHT, 101-104, 125
- Byte, 3

- CAPS LOCK, 16
- CAPS SHIFT, 10-12, 93
- Cassette recorder, 4, 17, 69-70
- Central Processing Unit, 2
- CHR\$, 76, 94
- CIRCLE, 121-5
- CLEAR, 120, 130, 135-6
- Clock, 134
- CLS, 90, 120
- CODE, 76
- Colon, 53-4
- Comma, 86-8
- Condition, 41, 43
- Conditional expressions, 43-4, 130-31
- Conditional loops, 47, 56

- Constant, 29
- Control keys, 10-12
- COS, 75
- Cursor, 11
- Cursor control keys, 16

- DATA, 68-70
- Default flow of control, 38-9, 56
- Deferred mode, 14
- DEF FN, 80-83
- Degrees, 123
- DELETE, 13, 16, 21
- Dollar sign (\$), 57
- DIM, 64-7
- DRAW, 119-25
- Dummy parameters, 81, 82

- Edit, 21
- Editing, 17
- ENTER, 12
- Error messages, 61
- Exclusive or, 132-3
- EXP, 73-4
- Exponential number, 73
- Extended mode, 14

- False, 43-4, 131-3
- FLASH, 101, 125
- Flow of control, 38-56
- FOR, 49-52
- FN, 80-83
- Functions, 71-83

- GOSUB, 83-5
- GOTO, 13, 38-50, 83
- GRAPHICS, 15, 93
- Graphics characters, 93-5
- Graphics mode, 15-17, 20, 93

- High-resolution graphics, 87, 118-29

- IF, 41-9
- Immediate mode, 14
- IN, 130-35
- Index, variable, 49
- Infinite loop, 39-41
- INK, 102-105, 124-5
- INKEY\$, 79-80
- INPUT, 29-36, 58, 91-3, 105
- Input area, 21
- Input device, 2
- INT, 74
- Integer, 74
- INVERSE, 99-101, 104, 125-6
- INV VIDEO, 16
- Inverse graphics, 15, 16
- Iteration, 40

- Keyboard, 2, 7-10
- Keywords, 11

- LEN, 63, 76
- LET, 12, 25, 29-36
- Line numbers, 12, 25
- Lissajous figures, 128
- LIST, 14, 21
- Literal string, 32
- LN, 74
- LOAD, 19, 69-70
- Logarithm, 74
- Logic, 130, 133
- Logical expressions, 131, 133
- Loop, 40-56
- Low-resolution graphics, 87-106, 118

- Machine code, 135
- Memory, 3
- Microdrive, 4, 6
- Musical notation, 109-111

- NEW, 19, 120
- NEXT, 49, 52
- NOT, 130-33
- Null string, 61
- Numeric variable, 24

- One-dimensional arrays, 64
- OR, 130-33
- Order of evaluation, 27
- OUT, 113, 130, 135
- Output device, 2
- OVER, 99-101, 104, 124-6

- PAPER, 103-105, 124-5

- Parameters, 72
- PAUSE, 112-13
- PEEK, 130, 133-5
- PI, 74-5, 123
- PLOT, 119-26
- POINT, 126-7
- POKE, 97-8, 130, 133-5
- Power supply, 8
- PRINT, 13, 25-6, 29-38, 58, 87-91, 102-104
- Printer, 2, 5
- Program, 2, 4, 23
- Pseudo colours, 104
- Pseudo randomness, 77

- Radians, 75, 123
- RANDOMISE, 78-9
- Random number generator, 77
- READ, 68-9
- Relation, 42-3
- REM, 130, 136-7
- Repeat key, 17
- RESTORE, 69
- RETURN, 83-5
- RND, 15, 77-9
- RUN, 11, 21, 120

- SAVE, 18, 69-70
- SCREEN\$, 116, 126-7
- Scroll, 14, 90-91
- Select, 45-6, 56
- Semicolon, 86, 87
- SGN, 74
- Sign, 74
- Simple variable, 24
- SIN, 75, 128-9
- Skip, 44-5, 56
- Sound effects, 107, 108, 113, 114
- SQR, 72, 75
- STEP, 51
- STOP, 19, 41, 54
- String, 32, 56-63
- String concatenation, 58
- String constant, 57
- String slicing, 59-62
- String variable, 57
- STR\$, 76-7
- Subroutines, 71, 83-5
- Substring, 59
- Substring extraction, 58
- Substring replacement, 59-61
- Substring searching, 59
- SYMBOL SHIFT, 11-17

- TAB, 88-90
- TAN, 75
- THEN, 42-56
- Timer, 134
- Trigonometrical functions, 73, 75-6
- True, 43-4, 131-3
- TRUE VIDEO, 16
- Truth table, 132-3
- TV set, 8-9
- Two-dimensional arrays, 66

- Unary minus, 27
- Until loop, 48
- User-defined functions, 71, 80-83
- User-defined graphics, 95-9

- USR, 97-9, 130, 135

- VAL, 77
- Variable, 23
- VERIFY, 18-19, 70

- While loop, 48

- X co-ordinate, 119

- Y co-ordinate, 119

- ZX-BASIC, 5-6
- ZX80, 5
- ZX81, 5

The appearance of the Sinclair ZX Spectrum in 1982 was a major event in personal computing. This book takes the Spectrum user in easy stages from his first steps in programming to a good level of competence.

Early chapters give a brief history of the ZX range, and give advice on how to set up the machine and use the keyboard. Subsequent chapters describe one's first steps in BASIC, looping and choice, handling text and numbers, and functions and subroutines. There are three chapters on graphics and sound, while the final chapter is devoted to logic and other advanced topics.

The book includes many programming examples, and most chapters contain at least one complete program listing, mainly for games applications. It is clearly and logically written, and will be invaluable to all Spectrum users, in the home, education and small business.

The Author

S. M. Gee is the co-author of a previous book on programming, and is a regular contributor to *Computing Today*.

'Ideal for beginners . . . a much better, more friendly and yet more informative introduction to Spectrum BASIC and programming techniques than the manual. I enjoyed reading this book, often responding to S M Gee's humour . . . This approach will make learning much more enjoyable - and that's how it should be.'

MICRO UPDATE

More books on the Spectrum from Granada

THE ZX SPECTRUM and how to get the most from it

Ian Sinclair

The essential book for all users

0 246 12018 5

THE SPECTRUM BOOK OF GAMES

M. James, S. M. Gee and K. Ewbank

Twenty-one high quality,
challenging games for your Spectrum

0 246 12047 9

Front cover shows ZX Spectrum personal computer,
designed and developed by Sinclair Research Ltd.

GRANADA PUBLISHING

Printed in Great Britain

0 246 12025 8

1 00583

£5.95 net
£5.95