

Learn BASIC Programming on the Sinclair ZX Spectrum

**by Andrew
and Veronica Colin**



LEARN BASIC PROGRAMMING

on the

Sinclair Spectrum

by
Andrew & Veronica Colin

Copyright

The distribution and sale of this product are intended for the original purchaser only. The course is copyrighted and all rights are reserved. The manual may not, in whole or in part be copied, photocopied, reprinted, translated, reduced to any electronic medium or machine readable form or reproduced in any manner without prior consent in writing from LOGIC 3 LTD. Duplicating, copying, selling or otherwise distributing the tapes supplied as part of this product is illegal.

Dedication

This book is dedicated to Lia and Antonio Nunzi, who lent us their villa on the outskirts of Perugia, Italy while we were writing the book.

We would also like to mention some of the other people who made the book possible :

The University of Strathclyde, who gave us the opportunity to develop the material and the methods used.

Our friend and publisher, Andrew Goltz, who inspired the project.

The firm of Studio Systems in Perugia, who trustingly lent us a word processor when our own broke down in the Italian summer heat.

The members of our family, who showed more patience and understanding than any author is entitled to.

To all these, and above all to Lia and Antonio, we express our grateful thanks.

Perugia, August 1983

Andrew Colin
Veronica Colin

Introduction

The Sinclair SPECTRUM is the product of an industrial revolution.

The invention of integrated circuits - and specifically the microprocessor - has completely changed many aspects of life today. Computers are so flexible that there seems no limit to what they can do: talk, control robots, navigate, diagnose and help to cure serious illnesses, and act as storehouses and curators of vast amounts of knowledge, all accessible in an instant.

The overall result of these developments can only be to speed up our intellectual progress. For example, if you have a brilliant new idea - perhaps for a revolutionary kind of power generator or a new motor to be used in space - you no longer have to spend years constructing a pilot version: you can quickly program the computer to tell you whether it will work. The machines can be used to remove the need for drudgery and repetitive work in almost every area, and they can help you do everything from planning the best way to grow crops to writing a book.

In short, the Second Industrial Revolution has happened; the computers are there for everyone to use, in all sorts of interesting and valuable ways.

What kind of people use computers? In a way almost everybody does. But most people miss out on the vital element of control because they simply let the machine run 'standard' programs which they buy or borrow from other people. A video game is an interesting and amusing pastime, but it will never help you do your homework or find out whether your space motor is going to work!

The computer itself is almost infinitely flexible, but the programs you buy for it are not. It follows that you must learn how to write your own programs. Only then will the full power of the machine be really open to you.

This book is for the reader who seriously wants to learn the science of programming. Perhaps you have got tired of playing computer games you buy in the shops and want to write your own: this book is for you.

works; this book is for you. If your maths isn't too good, don't worry - it is not mathematical skill that makes good programmers, but the ability to think logically, to persevere with patience and care, and to strive for perfection. Programming is like climbing a great mountain. Some parts of the route are easy, but some are hard. Some sections are exciting, whilst others - let's admit it - are boring but unavoidable. At any moment you can turn round and admire the panorama of topics you have already covered and programs you've written.

The top of this mountain is enormously rewarding. Once you're there, the full power of the computer will be at your feet. You can innovate, experiment, and write programs which are useful to others who haven't even begun the climb. A new and fascinating profession will be open to you.

This course is meant for someone who has never done any programming before. It takes matters slowly, and is divided into two parts.

'Learn BASIC...' gives you a useful and self-contained account of the easier aspects of SPECTRUM BASIC, the language used to program the SPECTRUM. **'Advanced BASIC...'** covers the more difficult topics.

The book consists of a number of chapters, and you should try and master each one before going on to the next. Give each chapter a preliminary read-through by all means; but be warned that reading by itself alone will not enable you to learn programming. You must answer the quizzes and do the programming exercises.

Never abandon a program until you've got it working properly.

Now you are ready to start. Don't rush it! Go slowly, methodically, and get as much practice as you can.

Good luck.

Chapter 1

Introduction

Installing the SPECTRUM

Typing simple commands

Loading programs

1.1 Introduction

This chapter helps you to set up and begin using your SPECTRUM. It explains several straightforward practical points which often create difficulties for people who have never used a computer before. Understanding this chapter can make all the difference to your success in learning this fascinating new technology.

To start with, find a quiet room where you can work undisturbed away from the rest of your family or friends. Take the phone off the hook and tell everyone you are busy. Choose a time when you are at your most alert - preferably not last thing at night - and set aside at least two hours for each session. Programming is great fun but it demands concentration. To get anywhere you must take the whole business seriously.

You will need a cassette recorder to load the programs and quizzes which go with this course - and also to save your own programs. You will find that any make of cassette recorder will work well with the SPECTRUM provided it has two 3.5 millimetre sockets marked MIC and EAR. A tape counter is useful but not essential. The shop where you bought your SPECTRUM will be able to advise you on a suitable machine.

If you have already installed your SPECTRUM, you can skip this next section. You may, however, want to look through it quickly and check that you really understand all the details.

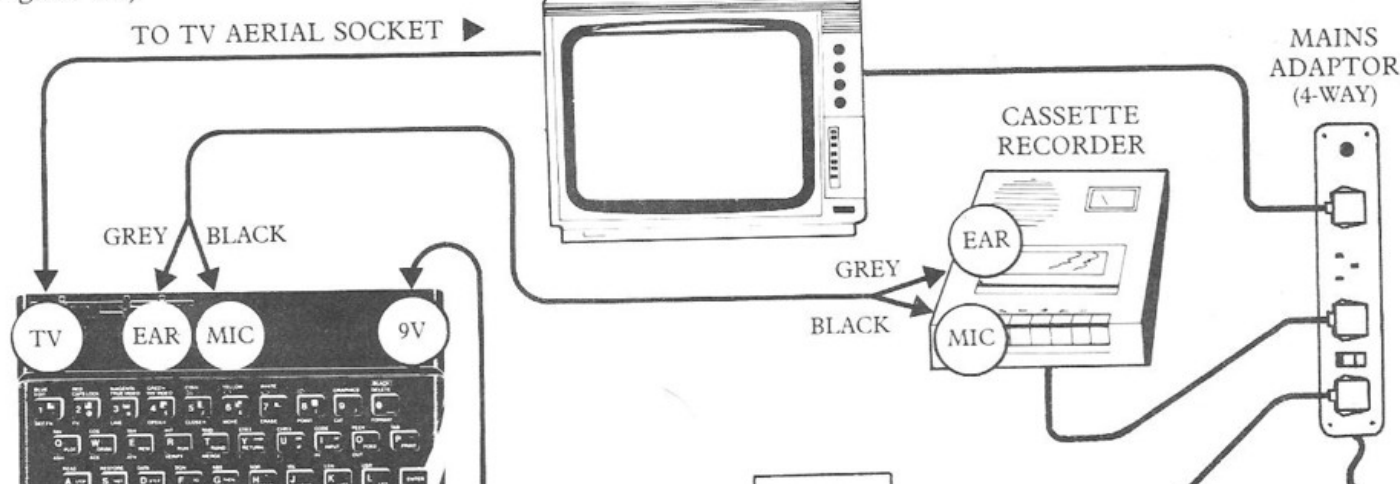
1.2 Installing the SPECTRUM

First set out the SPECTRUM, power supply and cassette recorder on the desk or table immediately in front of you. Put your TV set a little further away - about 4 feet (1.5 metres). You will need to be close enough to read what appears on the screen but if the TV is too near you will find it uncomfortable to work with. A single extension power lead with several sockets will be useful. It will allow you to use the cassette recorder and will give you a lot more freedom to arrange your home computer in the way which suits you best.

The various units connect together as shown in the diagram.

Be gentle with the plugs; examine them carefully before making connections and never use too much force.

(Figure 1.1)



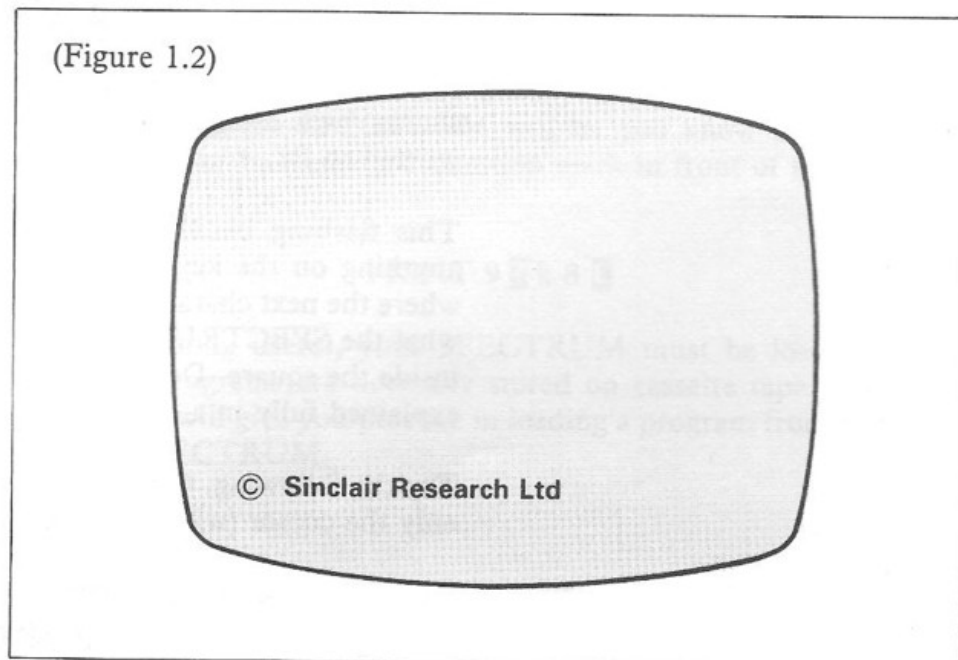
The plug from the power supply lead connects with the socket marked '9V DC' at the back of the SPECTRUM on the right and acts as an ON/OFF switch. The plug on the aerial lead connects with the socket marked 'TV' at the back of the computer on the left. The other end of the aerial lead fits in to the aerial socket on your television set. Note that your SPECTRUM will work with a black and white receiver, but in order to enjoy the full benefits of the machine's range of colour and graphics, you will need a colour set.

Leave the cassette recorder and the lead with the two plugs at each end aside for the time being - you'll find out how to connect it later in this chapter.

Now connect the TV and the power unit to the mains and switch on the TV. If your TV set has just a simple tuning knob, tune it in to the SPECTRUM somewhere near to 'Channel 36' on the dial. If your set has push-button tuning, choose a button which you don't normally use to receive TV programmes, and adjust the internal tuning control carefully. You will find tuning instructions in the booklet which came with the TV set. If you aren't sure how to do it, get some advice; but PLEASE don't take off the back cover of the TV set or poke around inside with a screwdriver or other instrument - this could kill you!

As you adjust the tuning the following message will suddenly appear on your screen.

(Figure 1.2)



Once the picture appears, you can turn the volume control right down - the SPECTRUM has its own built-in loudspeaker - and you may need to adjust the frame hold and line hold to keep the picture steady.

If you have any difficulty go back and check the following:

- * Is your TV set working? Try tuning into one of the broadcast channels (remember to replace your aerial) and if you can't receive anything at all, get your TV set repaired.

- * Make sure that the connections between the TV and the SPECTRUM

- Check:
- a) that the plug from the power supply is pushed firmly into the SPECTRUM's power socket.
 - b) that there is no general power failure.
 - c) That the fuses in either the mains extension plug or the power plugs for the SPECTRUM and TV are intact -change them if necessary.

WARNING: Don't try wiring up plugs or changing fuses unless you know exactly how to do it: mistakes could be fatal!

If your system still doesn't work go back to your dealer for advice and repair.

1.3 Typing Simple Commands

Now - if you haven't done this already - press some keys on your SPECTRUM keyboard. You will notice that the copyright message disappears and a line of nonsense - words, letters, numbers etc. appears in the lower half of the screen. These random doodlings don't serve much useful purpose and the first thing you will want to do is to get rid of them. One method is to 'rub out' what you have typed. You can do this by finding and pressing down the key marked CAPS SHIFT

A rectangular key with the text "CAPS SHIFT" in white capital letters on a dark background.

on the bottom left hand side of your keyboard. Keeping the CAPS SHIFT key down, press the DELETE key

A rectangular key with a white circle containing a diagonal slash, representing the delete function, on a dark background.

on the top right hand side of the keyboard. You will see a flashing black square with a letter on it sliding back along your lines of type erasing everything you have written so far.

This flashing black square is called the **Cursor**. When you type anything on the keyboard the cursor shows you in advance exactly where the next character will be displayed on the screen. It also shows what the SPECTRUM expects from you next by changing the letter inside the square. Don't worry about this for the moment - it will be explained fully in a later chapter.

Try the following. Clear the screen by using the DELETE key until only the cursor (with the letter K on it) remains.

Find and press the key marked

A rectangular key with a large white "P" and a double quote symbol above the word "PRINT" in white capital letters on a dark background.

The keyword PRINT appears on the screen, followed by the cursor which now has the letter L on it. Next type the number 5, followed by the '+' sign. To get this you must find the SYMBOL SHIFT key

A rectangular key with the text "SYMBOL SHIFT" in white capital letters on a dark background.

- the key with red lettering on the bottom right hand side of the keyboard, hold it down and then press the

SHIFT key and type the number 8. Finally press the key marked ENTER .


 A dark rectangular button with the word "ENTER" in white capital letters.

You have just typed an addition sum. The answer will appear at the top of the screen and at the bottom you will see the message

0 OK, 0:1

which means that the SPECTRUM has done what you asked it to do.

The prime function of the ENTER key is to tell the computer to carry out your instruction - in this case, to print or display the result of adding two numbers. Repeat the process several times, using any numbers you choose. Try it again, this time subtracting one number from another. Use the minus sign - SYMBOL SHIFT and the


 A dark rectangular button with a white 'J' on the left and a white minus sign '-' above the word "LOAD" on the right.

key.

Remember - there is absolutely no way you can break your SPECTRUM by typing in the wrong thing. Don't be afraid to experiment. Try PRINTing an addition sum and deliberately forget to press the SYMBOL SHIFT key when you type '+'. As soon as you press ENTER, the SPECTRUM will let you know that you have made a mistake by flashing a question mark in front of the incorrect sign.

PRINT 9 ? k 8 L

1.4 Loading Programs

To do anything useful, your SPECTRUM must be loaded with a program. Programs are normally stored on cassette tape. This first experiment will give you practice in loading a program from a cassette into the SPECTRUM.

Experiment 1a

1. Connect your cassette recorder to the SPECTRUM using the cable with the two plugs at each end. The two ends are exactly the same and it doesn't matter which way round you put the cable. Take one grey plug and put it into the MIC socket at the back of the SPECTRUM, then put the other grey plug into the MIC socket on the cassette recorder. Now put the black plug into the EAR socket on the SPECTRUM and the other black plug into the EAR socket on the recorder. *Don't* use the AUX socket if your recorder has one of these. Finally, plug in the cassette recorder to the mains and switch on.

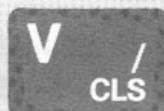
2. Press STOP on the cassette unit.

3. Open the holder on the cassette unit, take out any tapes which might be there already and insert the tape with "testcard" on it, the label uppermost and with the tape window towards you. Close the holder - if it doesn't close, don't use force but check that you have put the tape in correctly.

4. Make sure that you are at the beginning of the tape. Press the REWIND key and watch the cassette through the window. If you see the tape spinning, wait until it stops, then press STOP again.

5. Set the volume control on the cassette unit to a point slightly above halfway - THIS IS EXTREMELY IMPORTANT.

6. Clear the TV screen by pressing the CLS



key (short for CLeAr Screen)

and then ENTER.

Now type the following:

You type	Keys to press	What appears on the screen
LOAD		LOAD
"	and	LOAD"
testcard	8 letter keys	LOAD"testcard
"	and	LOAD"testcard"
ENTER		LOAD"testcard"

You make twelve keystrokes in all, not counting SYMBOL SHIFT. If you make a mistake, get rid of it with the DELETE key.

7. Your TV screen will go completely blank. Press the PLAY key on the recorder. After a short interval you should see some coloured and moving stripes, whilst the SPECTRUM makes a warbling noise. The message 'program testcard' appears on the screen.

Eventually, the program will finish loading and a new picture will come up. This picture consists of bars of all the colours the SPECTRUM can display. Stop the tape and make fine adjustments to your TV set to make the colours as bright and satisfying as you can.

If this loading process doesn't work (and quite likely it won't the first time), try it again with different settings of the volume control on the recorder. You have to find the right volume for your SPECTRUM, and once you've found it, leave it alone! Sometimes you may find that your SPECTRUM seems dead; it won't respond to any keys or do anything else. You can *always* restart it by pulling out the plug from the power supply (the one that goes into the socket marked '9VDC') and putting it back in again.

There can be other reasons for LOAD not working. If you have persistent trouble, read the instructions again carefully. Better still, get someone to confirm that you have understood them correctly. If need be, check your recorder by playing some music on it. As a last resort, go to your SPECTRUM dealer for help; but this is most unlikely to be necessary if you follow the instructions precisely.

Once you know how to load "testcard", you may find it useful to run the program whenever you need to adjust the colours on your TV set.

<i>Experiment 1a completed.</i>	
---------------------------------	--

Experiment 1b

Stop the "testcard" program by holding down the CAPS SHIFT key and typing BREAK SPACE. Load the program "quiz1" in the same way that you loaded "testcard". You type.

LOAD "quiz1" (followed by ENTER).

This quiz consists of a few simple questions for you to answer and it should help you get familiar with the keyboard of the SPECTRUM.

<i>Experiment 1b completed.</i>	
---------------------------------	--

A final word of advice: Once you've got your system working, it is far better to leave it set up without disturbing it. Every time you connect or disconnect a plug you cause a mechanical shock which weakens the connections inside the machine. If you keep having to 'put your computer away', you are much more likely to have expensive breakdowns!

Chapter 2

The keyboard



2.1 Introduction to the Keyboard

This chapter is about the SPECTRUM's keyboard, the part of the machine which you use constantly and which you should get to know very well. The exercise which goes with the chapter will help to improve your speed and accuracy in typing by giving you plenty of practice. You will find this useful as you continue with the course.

We shall look at the way the keyboard is used to type information or data, such as a shopping list or a list of names and addresses. At this point, we are not going to ask you to type programs. There is a special way of doing this which will be described in later chapters.

Look at the keyboard on your SPECTRUM carefully.

There are 40 keys and every one, except the CAPS SHIFT, SYMBOL SHIFT, BREAK SPACE and ENTER keys, is marked with a letter or number in white and a symbol or word in red (such as +, AND, =). The keys with letters on them also have words in white (such as PRINT or LET) and some of the keys with numbers have square shapes called 'graphics'. For the time being, ignore the words and signs printed on the SPECTRUM case above and below the keys - we'll come to those later.

If you've ever used an ordinary typewriter, you will notice that the numbers and letters are positioned in the usual place. Don't worry if you've never used a typewriter keyboard before, though - you'll soon get the hang of it!

The rules are simple:

1. To type a small (lower case) letter, just hit the right key.
2. To type a capital letter, hold down the CAPS SHIFT key with one hand and strike the correct letter key with the other.
3. To type a red symbol, hold down the SYMBOL SHIFT key and then strike the key with the symbol you need.
4. To type a space, just hit BREAK SPACE by itself. Don't hold down the CAPS SHIFT key at the same time. If you do you'll get some strange results.
5. To delete or rub out a mistake, hold down the CAPS SHIFT key and hit the DELETE key.
6. Don't hold down any key too long or it will start 'repeating'. This could be useful if you wanted to type lots of dots, dashes or spaces, but the need won't arise at this stage.
7. Note the difference between 0 and O. Throughout the book we use 0 to represent zero to distinguish it from the letter O - the computer is very fussy about the difference. Be careful, too, not to confuse the digit 1 with the letters I or l.

Experiment 2a

Start up the SPECTRUM.

Rewind the cassette tape, stop it and then type

LOAD "typex" (followed by ENTER as usual)

Press PLAY on your cassette recorder and wait for the program to load and for the instructions to come up on the screen. Don't forget to stop the recorder as soon as the program is loaded.

Run through the program several times and watch your rating improve. As you progress through the course, come back to this program again and again. Remember, practice makes perfect!

Earlier in this chapter you were warned about pressing CAPS SHIFT and BREAK SPACE at the same time. It's entirely probable that you might do this by mistake at some point. Everything will stop - rather as if an emergency brake had been put on. Don't panic - to get back to where you were, press the key marked CONT



, followed by ENTER, and the program will start up again.

If things go completely wrong, reload the program and start again.

<i>Experiment 2a completed.</i>	
---------------------------------	--

Chapter 3

Typing commands

The PRINT command

Variables and LET commands

3.1 Typing Commands

In the first part of this course, we have concentrated almost entirely on the SPECTRUM keyboard and on the way in which it can be used to display text on your TV screen. Now we shall look at some of the functions the SPECTRUM can carry out for you.

As you have already discovered, the SPECTRUM will execute various tasks - such as LOAD or PRINT - when instructed to do so. These instructions or commands are written in BASIC, a simple and widely-used language which was designed by Kemeny and Kurtz at Dartmouth College, U.S.A. in 1964. All languages have rules of grammar and BASIC is no exception - but the rules are simple and easily memorised with practice.

Every BASIC command has the same overall structure. It always starts with a keyword such as LOAD or PRINT. The keyword is usually followed by further details - what to print, which program to load and so on. The command always ends with the ENTER key.

In most computers the keyword must be typed in full, letter by letter, but the SPECTRUM - like the ZX81 - needs just a single keystroke. When the SPECTRUM expects you to type a Keyword, it flashes up an inviting K under the cursor. All you need to do is to strike the key with the right keyword on it : P for PRINT, J for LOAD etc. As soon as you've hit the keyword, the cursor switches to L.

Notice that the initial letters of the keywords don't always correspond to the big white letters on the keys.

If the keyword is the first item on a line, then the computer will obey the command as soon as you press ENTER.

You may make a mistake when typing a command. To correct it, rub out the characters with the DELETE key and retype. Note that each time you delete you get rid of one keystroke. This could be a single character or complete keyword.

3.2 The Print Command

PRINT is one of the most useful and flexible commands in BASIC. It tells the computer to work out something and then display the result on the screen. The word PRINT is used because the original BASIC system at Dartmouth used teleprinters to print out the answers on rolls of paper.

Experiment 3a

This experiment is divided into three parts. First, try out a number of different PRINT commands and note down your results. Then read over the points that arise, and finally look at some new PRINT commands and try to forecast what the computer will do. Obviously you can always check your answers by using the SPECTRUM.

Start by typing these commands, ending each one with the ENTER key. Make sure that you haven't made any typing errors - correct them if necessary - then use the boxes below to note down carefully what appears in the upper half of the screen. Write down what appears in the lower half of the screen only if it is different from the usual

Ø OK, Ø:1

Don't confuse the — sign (which is on key J) with the underline on key Ø.

The first two boxes have been filled in for you already.

PRINT 1984	1984
PRINT "hello"	hello
PRINT — 7	
PRINT 7 + 4 + 9	
PRINT 8 * 7	
PRINT 24/6	
PRINT "SPECTRUM"	
PRINT SPECTRUM	
PRINT 7,4	
PRINT 7;4	
PRINT "orange", "lemon"	
PRINT "butter";"fly"	
PRINT "66 + 99"	
PRINT 17 — 10 ; "tigers"	
PRINT 17 — 10 ; " space tigers"	
PRINT 1;2;3;4	
PRINT 1,2,3,4	

Before going on, look through the results and see how many different features of PRINT you can spot.

Here are the most important aspects of the PRINT command.

1. The command can handle both numbers and 'strings' which are

A number can be given either explicitly - such as 25 - or in the form of an expression or sum which the computer has to work out. The expressions in the examples above were

$7 + 4 + 9$, $8 * 7$, $24/6$ and $17 - 10$,

so clearly the SPECTRUM can add, take away, multiply and divide. The signs $*$ and $/$ mean multiply and divide respectively. If you want to use the SPECTRUM for more advanced calculations, you will be glad to know that expressions can be as complicated as you need and can include brackets and all the special functions that you'd expect from a scientific calculator. **Appendix A** at the end of this book goes into this in greater detail.

A string is a sequence of characters, such as letters in upper and lower case, numbers, or other symbols which are enclosed by double quote marks. When you use the PRINT command, these characters are reproduced exactly without the computer trying to process them in any way. The strings in the examples were:

"hello", "SPECTRUM", "orange", "lemon", "butterfly", "66 + 99", "tigers", "space tigers"

Note that although "66 + 99" looks like an expression, it is actually a string because it is enclosed in double quote marks.

If you want to display a string but forget to put the double quote marks around it, the computer will come up with an error message at the bottom of the screen, such as:

2 variable not found, 0:1

This happened when you typed in PRINT SPECTRUM. If you leave out one set of quote marks, the computer will display the usual flashing question mark when you press ENTER.

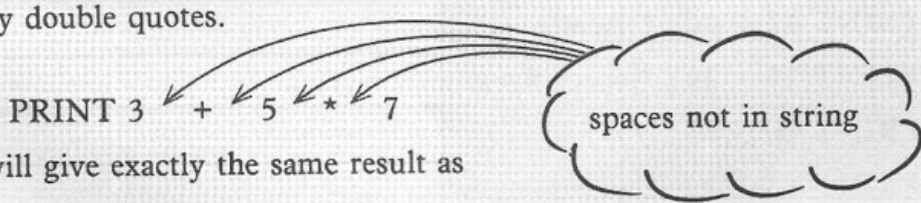
2. The PRINT command can handle two or more quantities or strings at the same time. If these are separated by commas, then the second result is spaced well away from the first - just over halfway across the screen. If a semicolon is used then there is no separation at all. You will have seen this with "butterfly" and 1234.

3. We next look at spaces inside the PRINT command. These are completely ignored except when they are part of a string and enclosed by double quotes.

PRINT 3 + 5 * 7
will give exactly the same result as
PRINT 3+5*7.

On the other hand

PRINT "oranges and lemons"
will be displayed on the screen as
oranges and lemons



spaces not in string

as

orangesandlemons

4. If you type an expression which doesn't make sense - such as

$12 + * 3$,

then the SPECTRUM will flash up the inevitable question mark. You must correct the mistake and try again.

Now go through the following list of PRINT commands and try to work out what the SPECTRUM will do in each case. Show, too, how the results will be spaced - this is as important as getting the right answer. Warning - look out for deliberate mistakes.

Check your predictions on the SPECTRUM. If you can't understand why you have made a mistake, go back to the beginning of this experiment and repeat it until you are quite clear about all the ideas.

Note that the SPECTRUM always carries out multiplication and division before addition and subtraction.

Examples:

PRINT $5 + 6 * 4$ will give 29 (not 44)

and

PRINT $4 + 8 + 6/2 - 2$ will give 13 (not 7)

COMMAND	YOUR PREDICTION	SPECTRUM'S RESULT
PRINT 20		
PRINT $94 - 78$		
PRINT $4 * 5 + 5$		
PRINT "KEEP OFF THE GRASS"		
PRINT DANGER		
PRINT "8*/6"		
PRINT 3,55		
PRINT $3 * 2; 3 + 2; 2 * 2; 2 + 1$		
PRINT "water";"melon"		
PRINT "16","cows"		
PRINT $3 + * 9$		
PRINT $-1;-3;-5;-7$		

Experiment 3a completed.

3.3 Variables and LET Commands

When a computer runs a program, it carries out a whole chain of simple commands one after the other. The links between the commands, which allow them to work sensibly with each other, are provided by special quantities called variables.

Before we go on to look at BASIC variables, you may find it helpful to study this example.

You have been asked to take part in a local traffic survey to find out how many cars, lorries, buses and bicycles pass a given point in one hour. You are given the following instructions.

Before you start, draw four boxes and label each with the name of one of the forms of transport. You haven't started your count yet so write 0 in the corner of each box.

CARS	LORRIES	BUSES	BICYCLES
0	0	0	0

Whenever a car goes past, replace the most recent number in the CAR box with the same number plus one. Do the same for the LORRIES, BUSES and BICYCLES.

After ten minutes, your record sheet could well look like this:

CARS	LORRIES	BUSES	BICYCLES
0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5	0	0 1 2

When the hour is up, you must hand in your results, listing the names of the four types of transport together with the latest figures inside their respective boxes.

CARS	LORRIES	BUSES	BICYCLES
84	32	1	19

In this example, the boxes correspond to variables. The numbers in the boxes keep track of the different totals and change from time to time but the labels - or names of the different forms of transport - stay the same throughout.

The SPECTRUM has a memory which can be compared to a row of empty pigeon-holes or boxes. When the machine is first started up, the boxes are all empty and unlabelled. As soon as a variable is mentioned, the computer 'chooses a pigeon-hole' by setting aside part of the memory and labelling it with the name given to it by the programmer. Then it 'writes inside it' by putting the value of the variable inside the chosen part of the memory.

LET x = 5 ENTER

The variable name here is x. The SPECTRUM will create a pigeonhole called x (if it hasn't done so earlier) and will put the number 5 inside it. If the variable x already exists, because it has been mentioned in a previous LET command, then no more space is allocated and any value which is already there is replaced by the value 5.

Look at the result of LET x = 5 in the following examples:

CASE 1	x does not exist
Before	The memory is empty
After	x = 5

CASE 2	x already exists
Before	x = 63 (or any other value)
After	x = 5

When you give a LET command followed by ENTER you will get no response from the SPECTRUM, apart from the message:

0 OK, 0:1

at the bottom of the screen. The only way to find out whether the machine has actually done anything is to use a PRINT command which gives you the value of a variable whenever it is mentioned by name. Try the following sequence of commands.

LET y = 21

PRINT y

LET y = 34

PRINT y

If you keep to this order, the first value to be displayed is 21 and the second is 34. The first LET command creates a variable called y and gives it the value of 21, the second merely changes it to 34.

At this point we must give you a few simple rules about variables and their names.

There are two kinds of variables in BASIC:

1. **numeric variables** for storing numbers
2. **string variables** for storing strings - words or phrases.

You can be as inventive as you like when choosing a name for a

Some examples of permissible names are:

y
a
zt
z5
r2d2
averylongnameforanumericvariable

It is usually easier to stick to short names. Note that it makes no difference whether the name is written in upper or lower case or in a combination of the two.

sixteen
SIXTEEN
SixTeeN

would be considered by the SPECTRUM to be the same name.

Not allowed as numeric variable names are:

25 - it begins with a number
3 blind mice - also begins with a number
apple—pie - contains a minus

Names of string variables are more restricted. Each one must consist of a single letter followed by a dollar sign:

c\$, z\$, x\$

To show the use of string variables, try typing in the following:

```
LET t$ = "locomotive"  
PRINT t$
```

The value that follows the = sign in a LET command does not have to be a simple number or string. It can also be a variable name or an expression, which can use the names of variables. Look at these commands:

```
LET a = 7  
LET q = a + 4
```

The first command creates a variable called a and gives it the value of 7. The second command creates another variable called q and then takes the value of a, adds 4 and puts the result in q. Type in these two commands on the SPECTRUM and then check what the machine has done by typing

```
PRINT a, q
```


Now look at the following sequences and try to predict what the SPECTRUM will do in each case:

LET xyz = 25

LET a = 60 - xyz

PRINT xyz, a

LET c = 5

LET h = c * c + 4

LET t = h - c

PRINT c;h;t

LET j = 6

LET j = j + 1

PRINT j

You may have found that the last example needed a little more thought. A variable can hold any number of values, but only one at a time. A command like

LET j = j + 1

means:

'First work out the expression by taking the current value of j and adding one. Then put the result in the part of the memory labelled j, replacing the current value.'

In other words, the command makes the SPECTRUM add 1 to the current value of j.

The signs which are used to combine numbers in various ways are called arithmetic operators. They are +, -, *, and /. BASIC also allows you to join two strings together. This process is called 'concatenation' (from the Latin 'catena' which means 'chain'), and is shown by a '+' sign. The operator chains the second string to the first so that, for example,

"con" + "cat" + "e" + "nation" = "concatenation"

Look at the following sequence of commands and predict what will be displayed with the PRINT commands. Then try the sequence on the SPECTRUM. Don't forget the space before each of the closing quotes.

LET a\$ = "LOOK space"

LET b\$ = "BEFORE YOU space"

LET c\$ = "LEAP space"

LET d\$ = a\$ + b\$ + c\$

LET e\$ = c\$ + b\$ + a\$

use the space key

PRINT and LET are the two most frequently used commands in BASIC. Although the keyword LET can sometimes be omitted when using BASIC on other computers, it must always be used in SPECTRUM BASIC - the single key depression makes this a simple operation.

Experiment 3b

The program which comes with this chapter is designed to give you plenty of practice with PRINT and LET commands. It is called "c3drill". Work through it until you are sure that you completely understand the use of numeric and string variables.

<i>Experiment 3b completed.</i>	
---------------------------------	--

Chapter 4

Stored commands
Loops

4.1 Stored Commands

Up to now we have been giving the SPECTRUM commands or series of commands and getting the results back straightaway - one result for each command we have typed. When you consider that the computer is capable of hundreds or even thousands of elementary operations every second, then this is clearly not the most efficient way of working. It's as if a Formula One racing car were to be pulled along by a team of oxen.

The memory in a computer is a versatile device. Not only can it hold the values of variables, but it can also store commands. These commands can be brought out and used very much faster than you could possibly type them in. This section explains how this is done.

Start up your SPECTRUM - or, if it already has a program in it, type the keyword NEW, followed by the ENTER key. NEW is a powerful command which completely wipes the SPECTRUM's memory so that programs stored there before cannot interfere with what you want to do next - rather like cleaning a blackboard or taking a fresh sheet of paper. Only use NEW if you are quite sure you really mean it.

Type in the command:

```
10 PRINT 29 + 74
```

Make sure that you use the numeric keys, 1 and 0, not the letters, I and O. Then press the ENTER key.

You will see the line you have just typed leap to the top of the screen, leaving the keyword cursor flashing at the bottom waiting for your next instruction. Notice that the command is not obeyed - not yet, anyway.

Up to now you have seen this top region of the screen used to display strings and numbers. The lower part of the screen could be described as your work bench. It's where you type in commands and the lines of programs you are working on. When you press ENTER after typing a labelled command, it's as if you hand it over to a super-efficient secretary, who files it safely away and at the same time displays a copy of it for you on a notice board.

Let's see whether the PRINT command has been filed away. Clear the screen by typing CLS and then ENTER. Then type the keyword LIST



, followed, again, by ENTER. Your PRINT command reappears on the screen, proving that the SPECTRUM really did have it stored away safely.

So far, so good. You have stored and retrieved a command. You still have to find out what 29 and 74 add up to. To do this, type the command GO TO 10 - you'll find the keyword GO TO on the G key. Remember to follow the 10 with ENTER. The SPECTRUM now fetches and obeys the command labelled 10, and the answer appears immediately below the program line in the top half of the screen. Type GO TO 10 several more times and the answer is repeated, proving

time, and is only limited by the size of its memory. Every stored command is given its own label which must be different from all the others. You can use any whole number in the range 1 to 9999 as a label. The SPECTRUM stores and lists commands in increasing order of label number and obeys them in this order unless specifically told otherwise.

Now try typing:

NEW

10 PRINT "Two numbers"

20 LET a = 7

30 LET b = 10

40 PRINT a * b, a + b

Remember to end each command with ENTER.

Check the display on the screen carefully. Have you made any mistakes? There are two very simple ways of correcting them.

a) If you want to get rid of a complete command, type only its label number, followed by ENTER.

b) To change an incorrect command, just retype it correctly, making sure that you have given it the same label number as before. Then press ENTER. This new version will replace the old one.

When you are sure that your program is correct, clear the screen with CLS and try a LIST, followed by a GO TO 10. Are the results what you expected?

The SPECTRUM will arrange your commands in the right order, even if they are not typed in that order. For example, if you type:

30 LET b = 10

10 PRINT "Two numbers"

40 PRINT a * b, a + b

20 LET a = 7

you will see the commands being slotted into their correct sequence in the display at the top of the screen. Clear the memory with NEW and try it for yourself.

It is common practice to make your label numbers go up in steps of 10. If you want to put some extra commands in later, you can choose label numbers that fall in between, such as 15 or 47.

Why do we want to store commands? For two excellent reasons:

1. The SPECTRUM can execute stored commands very much faster than type-as-you-go commands.

2. Commands which are stored can be repeated as many times as you

command. Look at the following program:

```
10 PRINT "spring"  
20 PRINT "summer"  
30 PRINT "autumn"  
40 PRINT "winter"  
50 GO TO 10
```

When the program is started at label 10, the SPECTRUM obeys the first four commands in sequence. The next command sends it back to label 10, and it will repeat the process again and again - until you stop it.

Type the program into the SPECTRUM (remember to type NEW first) and start the program by typing this unlabelled command:

```
GO TO 10
```

You will see the four words repeated over and over again - 22 times in all - in a column on the upper part of the screen. At the bottom you will see the word, 'scroll ?' Press any key (except N or BREAK SPACE) and the display will repeat itself. To stop it, press BREAK SPACE followed by ENTER and this will bring up your program on the screen again. When you answer the scroll question, you don't use the CAPS SHIFT key.

You can also stop the display and retrieve the program by typing N followed by ENTER.

At this point we shall show the use of having label numbers separated by 10. Suppose you wanted to alter your program so that it included the months of the year under the appropriate season:

```
spring  
March April May  
summer  
June July August  
autumn  
September October November  
winter  
December January February
```

You will need four new instructions in between the existing ones. If you number them 15, 25, 35 and 45 they will go in the right places.

Type the following (don't type NEW!):

```
15 PRINT "March April May"  
25 PRINT "June July August"
```

As you follow each command with ENTER, you will see your four new commands being inserted in their correct places. Start the program with GO TO 10.

Another easier way to start your programs is to use the keyword RUN. You don't need to specify a line number. RUN will make the SPECTRUM start automatically at the command with the lowest label number.

Experiment 4a

In this experiment, we'll ask you to write a program to fill up the screen with a pattern. So far, every PRINT command you have given uses a new line, but if you put a semi-colon after a number or string, the SPECTRUM will continue along the same line until it reaches the right-hand edge and then start a new line. By typing:

```
10 PRINT "*";
```

```
20 GO TO 10
```

```
RUN
```

the screen will rapidly fill up with stars.

Try writing and testing your own program along similar lines, using words, numbers or symbols.

<i>Experiment 4a completed.</i>	
---------------------------------	--

4.2 Loops

The name given to a sequence of commands repeated over and over again is a **loop**. A loop can contain many different kinds of commands, including a LET which we discussed in 3.3. Remembering that LET gives a new value to a variable, look at the following program and see if you can predict what it will do.

```
10 LET x = 1
20 PRINT x
30 LET x = x + 1
40 GO TO 20
```

Work through the commands step by step in exactly the same way as the computer does. What happens to x and its values ? Don't go on until you have really thought it through, and then write down the result in the box below.



Now enter the program and run it.

Here is an explanation of the result - although you probably found this problem quite easy.

The program starts by obeying the command labelled 10 which gives x the value 1. The next command displays this value on the screen. Command 30 replaces x by x + 1, simply adding 1 to the old value of x. The next command GO TO 20 makes the SPECTRUM return to command 20. The value of x is now displayed as 2. The computer works through the loop, 20, 30, 40, again and again, and each time the value of x is increased by 1. This gives you the sequence:

1
2
3
4
5

As extra practice, see if you can predict the first few lines which will be displayed by these programs. (Remember that * means multiply).

```
10 LET a = 0          10 LET y = 1
20 PRINT a             20 LET z = y * y
30 LET a = a + 2       30 PRINT y, z
40 GO TO 20            40 LET y = y + 1
                      50 GO TO 20
```

Enter these programs and check if you were right.

Strings can be used in loops in the same way.

```
10 LET a$ = "+"
20 PRINT a$
30 LET a$ = a$ + "+"
40 GO TO 20
```

The successive values of a\$ displayed as the program goes round and round the loop will be:

```

+
++
+++
++++
+++++
etc.
```

The string a\$ gets longer and longer and uses up more and more space on the screen. The largest number of characters allowed in a SPECTRUM string is well over 1,000 and it will take some time before this limit is reached.

Here are some details about the GO TO command which you should know. The number in a GO TO should normally refer to a line which is present in the program. If there is no line with this number, the computer jumps to the next highest numbered line. If there is no such line the computer stops.

To give an example, the following program will display only the numbers 2 and 7, and then stop:

```
10 GO TO 30
20 PRINT 5
30 PRINT 2
40 GO TO 55
50 PRINT 22
60 PRINT 7
```


Try to predict what will happen in the following programs.

```
10 LET a$ = "***"  
20 PRINT a$  
30 LET a$ = "A" + a$ + "B"  
40 GO TO 20
```

```
10 LET x$ = "AB"  
20 PRINT x$  
30 LET x$ = x$ + x$  
40 GO TO 15
```

Remember that if a letter comes inside a string, it really is a letter and not the name of a variable.

Experiment 4b

As an interesting exercise, here is a program with a simple loop.

```
10 LET j = 0
20 LET j = j + 1
30 GO TO 20
```

Run this program for exactly 30 seconds, timing it with your watch. Stop it by holding down CAPS SHIFT and hitting BREAK SPACE. You need CAPS SHIFT because the program is in an endless loop, not printing anything or asking to be scrolled. Then find out the value of j by giving the command

PRINT j

This will allow you to work out exactly how many commands have been obeyed in that time. How many commands does the SPECTRUM obey in one second? Write your answer below.

Experiment 4b completed.

Now load "quiz4" and answer the questions.

Chapter 5

Editing programs

Saving programs

Further points on security

The aim of this book is to start you off designing and building your own programs. Chapter 5 could be described as a programmer's tool-kit. It tells you about certain important techniques and methods which will make your job of programming very much easier. Read the chapter through carefully. Practice the techniques it describes and make sure you can use them whenever the need arises. Don't be afraid to refer back to the chapter whenever you have to.

5.1 Editing Programs

It is a basic fact of life that everybody using a computer makes mistakes. One measure of a good computer system is that it lets you correct your errors easily and quickly.

Let's begin by reviewing the methods you already know for correcting the mistakes you make when you type commands on the SPECTRUM.

- a) If an error is reported when you press ENTER, use the DELETE key to make the correction.
- b) To replace a command which has already been entered, type the new command with the same label number as the old one.
- c) To put an extra command into a program, just type it with a suitable line number - it will go directly to the right place. For example, if you want a command to go between 40 and 50, label it 45.
- d) To get rid of a command completely, just type its line number by itself.

So far these methods of correcting mistakes have worked well. The program lines have been short and our mistakes - hopefully - have been few!

Soon, as you gain skill in programming, the lines will grow longer and more complex. The task of retyping a whole line to correct a single error can be extremely tedious. Fortunately, there is a much quicker method of correcting and changing programs which we shall now discuss.

Load the program "graph". Unlike previous programs in the course this program will not run itself but comes up with the message:

```
0 OK,0:1
```

At this point stop the tape recorder and start the program by keying RUN followed by ENTER. You will see a pattern displayed on the screen which you can repeat as many times as you like. When you have seen enough, stop the program by typing CAPS SHIFT and BREAK SPACE. Clear the screen (with CLS and ENTER) and then list the program by typing LIST, followed by ENTER as usual. The first 22 lines of the program (but only 19 commands as some take up more than one line) are displayed with scroll ? at the bottom of the screen. You can list the rest of the program by typing any key except N or BREAK SPACE.

The complete listing is given below. Don't worry if you don't understand much of it just now!

```
10 REM graphs
20 REM COPYRIGHT TALENT COMPUTER SYSTEMS 1983
30 DEF FN x (q) = 120 + 119 * SIN (q * 2 * PI/n)
40 DEF FN y (q) = 80 + 74 * COS (q * 2 * PI/n)
```

```
70 FOR j = 1 TO n
80 LET x (j) = FN x (j)
90 LET y (j) = FN y (j)
100 NEXT j
110 CLS
120 PAPER 7
130 INK 0
140 FOR k = 1 TO n - 1
150 FOR m = k + 1 TO n
160 PLOT x (k), y (k)
170 DRAW x (m) - x (k), y (m) - y (k)
180 NEXT m
190 NEXT k
200 PRINT AT 21,0; "Hit a key to repeat pattern"
210 PAUSE 1000
220 CLS
230 GO TO 140
```

The LIST command works in a curious way and it is helpful to understand what it does.

If you just type

LIST ENTER

then, as you have seen, the machine displays the whole program in a series of screenfuls or 'pages'. At the end of each page it asks whether you want the next page.

If you follow LIST with a particular number, as in

LIST 100 ENTER

the display starts at line 100 (or the nearest line to it). If line 100 exists, you'll see it marked on the screen with a > sign which appears between the line number and the command; this is called the program cursor.

Type

LIST 90 ENTER

and see if you can find the cursor.

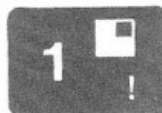
The program cursor is one of the most useful features of the SPECTRUM and you should know exactly how to use it.

If you need to alter an existing line in a program (other than by retyping it completely) you must first mark it with the program cursor. One way to do this is to LIST the line you want to edit; but another method, which is easier if you only want to move the cursor a few lines up or down, is to use the cursor control keys marked 6 and 7. Do a LIST which shows the cursor. Then hold down CAPS SHIFT

moves it up. Experiment with both keys until you're quite sure how to manage them. What happens if you try to move the cursor beyond the end of a program ?

If you try to list a command that isn't there, the cursor goes into hiding. You can get it back by moving the cursor either up or down. Try it for yourself.

Once you've selected a line with the program cursor, you can 'bring it down' to the input section of the screen to edit. Just hold down CAPS SHIFT and hit



, EDIT, and the line will jump down for you. Here's an example. Look at line 50 of "graph":

```
50 LET n = 14
```

The variable *n* represents the number of points in the design. The more points it has, the more complicated the design becomes. Let us see what happens when the value of *n* is changed to 20.

Instead of retyping the whole command, we shall edit the one already in the SPECTRUM. To do this, position the program cursor at line 50. Then type CAPS SHIFT and the EDIT key. Line 50 is brought down to the bottom part of the screen and the statement cursor (which flashes and is different from the program cursor) appears. You can move the statement cursor left or right using the 5 and 8 keys, together with CAPS SHIFT. The statement cursor controls the place where any changes are to be made. For example, the DELETE key rubs out the character immediately before the statement cursor, whilst most other keys insert new characters immediately in front of it, shuffling the rest of the line along to make room.

To remove part of a command, put the cursor after the part you want to get rid of, and press DELETE until it is all gone.

To put new characters in a command, put the cursor at the place you want them and just type them in.

To change a command, go through the two steps we have just described: get rid of the part you don't need and fill in the characters you want instead.

Let's be specific. To change command 50 from

```
LET n = 14
```

to

```
LET n = 20
```

1) Use the CAPS SHIFT and 8 to put the cursor after the 14.

2) Hold down CAPS SHIFT and hit DELETE twice. The 14 disappears and you get

```
50 LET n =
```

top half of the screen to replace the old.

When you've tried the effect of this command by RUNning it, experiment with some more changes. You'll have to make them carefully, since most random alterations will simply stop the program from working. If you should spoil the program, simply reload from the tape and start again.

Experiment 5a

Here are some things to do. (Each change will create a different pattern. Run the program after each change to see what it does).

1. In line 50, change the value of n to 3, 12 and 30.
2. Change the values in lines 120 and 130. Any numbers between 0 and 7 will do. This will change the colours in the design. At the moment INK is set at 0, which means the lines are drawn in black. PAPER, the screen colour, is set at 7 which is white. Each number can call up a different colour. These are indicated on the SPECTRUM case immediately above the number keys. Why not try a red design on a green background - or the other way about? Remember - if you choose the same colour for both INK and PAPER, the drawing will be invisible!
3. Change the shape of the pattern by altering lines 30 and 40. The changes can be in either or in both commands. Try:

```
30 DEF FN x(q) = 120 + 119*SIN (q*q*q*6.28/n)
```

or

```
40 DEF FN y(q) = 80 + 74*COS (100 /q*6.28/n)
```

Don't worry about not understanding what the commands mean. The results are what count in this experiment!

Editing seems difficult at first, but gets much easier with practice.

<i>Experiment 5a completed.</i>	
---------------------------------	--

5.2 Saving Programs

When you've spent some time modifying a program you will probably want to preserve it so that you can run it again later. This applies even more if you have just written a program of your own!

This section describes how to use your cassette recorder to SAVE programs on a cassette tape.

Saving programs is a fiddly process; you have to get everything exactly right or it won't work for you. If things don't go so well the first time you try, don't get upset; just read the instructions again, and make another attempt.

To begin with, you'll need a blank cassette tape on which to save your program. Any tape will do, but you'll find it much more convenient to use a short 'computer' tape such as a C5 or C7 than a full-length one meant for music. It's also cheaper!

Let's suppose you want to save your own version of the "graph" program. The new version is in the computer, and you have tested it out by typing RUN and checking that the picture is one you like.

First, you'll have to think of a name for your program. The name can be any string of 10 characters or fewer, but it is advisable to choose a name which isn't too long and which you can remember easily. We'll take "mygraph" as a suitable example. Now make sure that your cassette recorder is plugged in to the mains and connected to the computer. Take out the program cassette you normally use and load your blank cassette instead - either way up will do. Rewind it to the beginning of the tape.

Next - and this is important - pull out the EAR plug from the recorder, leaving only the MIC plug connected. Don't move the volume control from its usual position.

The next stage is to type the command

SAVE "mygraph" ENTER

using the



key.

(of course you can use any other name instead of "mygraph"). The computer replies,

Start tape and press any key.

Start the tape by pressing the PLAY and RECORD keys at the same time. If the RECORD key won't go down, don't force it but make sure that the cassette is properly loaded.

One possible reason that you might not be able to press the RECORD key is that the cassette is 'write-protected'. Inspect the back edge of the cassette, where you should see two small plastic lugs. If they are broken off, leaving two square holes, it means that the tape contains valuable information and you are not intended to spoil it by recording anything new. To see what a write-protected cassette looks like, examine the tape provided with this course. If it turns out that the tape

When you have got the recorder started, count slowly to 10 to let the tape wind past the plastic 'leader' at the beginning. Then press any letter key on the SPECTRUM keyboard. The screen will show all sorts of stripes and jagged patterns, and after a few seconds (depending on the length of your program) the computer will come up with the message

OK

At this point, if all has gone well, your program should be recorded on the tape. Stop the recorder and rewind the tape. Unfortunately, things don't always go well. The tape might have a bad patch, or you may have loaded it crookedly or forgotten to press the RECORD key. It is always best to check that the program has been stored correctly. To do this, reconnect the EAR lead, and type

VERIFY "mygraph" ENTER

(You'll find the VERIFY symbol just below the 'R' key. To bring it up on the screen, hold down CAPS SHIFT and press SYMBOL SHIFT. The cursor changes to a flashing E. Then hold down SYMBOL SHIFT and press the R key).

The name of the program you give must be the same as the one you have just saved.

When you've typed ENTER, press the PLAY key on the recorder. (You should have already rewound the tape). The machine will play back the program and compare it to the version stored in its memory. If the recording is correct (and in most cases it will be) the SPECTRUM will say,

OK

Otherwise, it will tell you of a tape error. In this case, it is worth repeating the whole process again. It is very rare for the difficulties not to go away after two or three tries. If the problems persist, try using a different blank tape, or adjusting the volume setting (very slightly!). It also pays to read the instructions a second time, or to get someone else to read them and to confirm that you have understood them correctly. In the extremely unlikely event that the process still doesn't work, get some advice from your SPECTRUM dealer.

Once a program has been SAVE'd and VERIFY'ed, label the tape and put it away. You can reload the tape any time you like with the LOAD command.

Experiment 5b

If you haven't already done so, try SAVE'ing a version of the "graph" program now. Make sure that you can VERIFY it and reload it after typing

NEW

ENTER

<i>Experiment 5b completed.</i>	
---------------------------------	--

5.3 Further Points on SAVE'ing Programs

There are two more important points to be made about SAVE'ing programs.

The obvious time to preserve a program is when you have finished writing and testing it, and you're sure that it works correctly. Sometimes it is also worth SAVE'ing a program before it is ready! Sooner or later you will move on to writing large complex programs, which will take several hours or days to key into the SPECTRUM and get going. As you begin to handle problems of this size, you should get into the habit of saving your program every hour or so, even in its unfinished state. Then if there is an 'unfortunate accident' (such as someone tripping over the power cable and jerking the plug out of its socket) you will lose at most an hour of work rather than all you have done so far. Accidents don't only happen to equipment: it is perfectly possible for you to erase all your program just by typing the keyword NEW!

A good way to organise these 'safety dumps' is to give the program a different version number every time you save it. For instance, successive dumps of a program could be called "tarot1" , "tarot2" , "tarot3" and so on. You need only keep the last two versions, so you can use two cassettes alternately - one for the even-numbered dumps, and one for the odd-numbered ones. You could also use both sides of one cassette, but it is best not to do so, since you might just get the tape tangled when saving the program and spoil both versions at the same time. These things happen to the best programmers!

When you start building up your program library, you may want to store several programs on the same tape. To do this successfully, you'll have to be extremely careful, since it is only too easy to SAVE one program on top of another and spoil it. Some people believe the best plan is to buy lots of short tapes (they are quite cheap from computer shops) and record only one program per side. It certainly makes loading faster.

If you are determined to put several programs on to the same tape, here are a few hints:

1. Keep your programs well separated. Having a tape counter certainly helps. Otherwise, if you have just LOAD'ed, SAVE'd or VERIFY'ed a program, then you can stop the tape and record the next program without rewinding.

2. Use the SPECTRUM to keep track of the programs on each tape. If you load a cassette and type

LOAD "funny"

where "funny" is a name of a program not on the tape, the machine will search through the whole tape and tell you the names of all the programs it finds. It won't, however, tell you if they are correctly recorded.

- 3) Suppose you have a personal library tape which already contains programs "a" , "b" and "c" - in that order. Someone lends you another tape with a program "d" and you decide to put a copy of it on to your own library tape. First make sure you are allowed to copy the program - if it contains a copyright notice then copying is probably illegal. If there is no reason not to copy, the best procedure is this:

This will position your tape just past the end of program "c", the last program on your tape. Do not rewind!

b) Load the borrowed tape and type

LOAD "d" ENTER

This will put the new program into the store of the SPECTRUM.

c) Reload the library tape (still without rewinding), pull the EAR plug out of the recorder and type

SAVE "d" ENTER

Continue the SAVE process in the normal way.

d) When the machine says 'OK', put the EAR plug back, rewind the tape and type

VERIFY "d" ENTER

The machine will find its way past programs "a", "b" and "c", and tell you if "d" has been correctly recorded.

If you do all this correctly, the process will work well. If you make any mistakes at all (such as rewinding, even a little, when you are not supposed to) you run the risk of spoiling the other programs on your tape. **YOU HAVE BEEN WARNED!**

One last point: if you SAVE a program in the ordinary way and then LOAD it back, you still have to type RUN for the program to start. If you prefer the program to start automatically when it is loaded, you can SAVE it with a command like

SAVE "program" LINE 10

where the number is the first line of the program (usually 10) and the keyword LINE is made by pressing CAPS SHIFT and SYMBOL SHIFT together, and then holding down SYMBOL SHIFT and hitting the key marked 3.

The self-test quiz for this chapter is called "quiz5"

Chapter 6

Conditions

Conditional GO TO commands

Designing programs

What '=' means in BASIC

6.1 Conditions

The programs which we wrote in Chapter 4 were like blind automatons. They had no way of coming to a halt; once started many of them would have gone on churning out results forever unless we took independent action to stop them.

We now reach one of the fundamental aspects of computing: how to write programs that can take decisions. When you have understood this topic, you will have advanced a considerable way along the road to being a programmer. Read Chapter 6 through slowly and carefully, and go back if you are unsure about any of the points described. You will find that time and effort spent on this part of the course will be amply worth while.

Programs that can take decisions depend on an important concept - the condition.

When we talk to one another, we use

Commands	- Go and do that now!
Questions	- Where did I put my keys ?
and Statements	- The weather is terrible today.

The statements we make are usually true - or are meant to be taken as true. Conditions on the other hand are groups of words which might be true but might equally well be false. In English, conditions tend to come between the words if and then :

If that shop sells peaches then go and buy me some.

If your car doesn't start then you must catch a bus instead.

The conditions in these sentences are

that shop sells peaches
and
your car doesn't start

It is understood that they might be true or false. The speaker really doesn't know whether that shop does stock peaches or whether your car won't start - but he is making plans accordingly.

In BASIC, as in English, conditions come between the keywords IF and THEN. All the different 'things' we have seen so far in programs - numbers, strings, number variables and string variables - can be included in conditions. As in English, the conditions can be true or false. Every condition depends on one of six 'relationships':

=	('equals' or 'is the same as')
<>	('not equal to' or 'is different from')
<	('is less than')
>	('is more than')
<=	('is less than or equal to')
>=	('is more than or equal to')

Let's look at an example which uses the first of these relationships. Here is a BASIC condition which involves a numeric variable

FALSE for any other value of x : 0 or 2 or 2.39 ...

Another example, this time using a string variable, is

$a\$ <> \text{"JOE"}$

The condition is TRUE if $a\$$ is different from "JOE". For instance, it would be TRUE if $a\$$ were "JACK", "BILL" or "ANDREW". It is only FALSE if $a\$$ is actually "JOE".

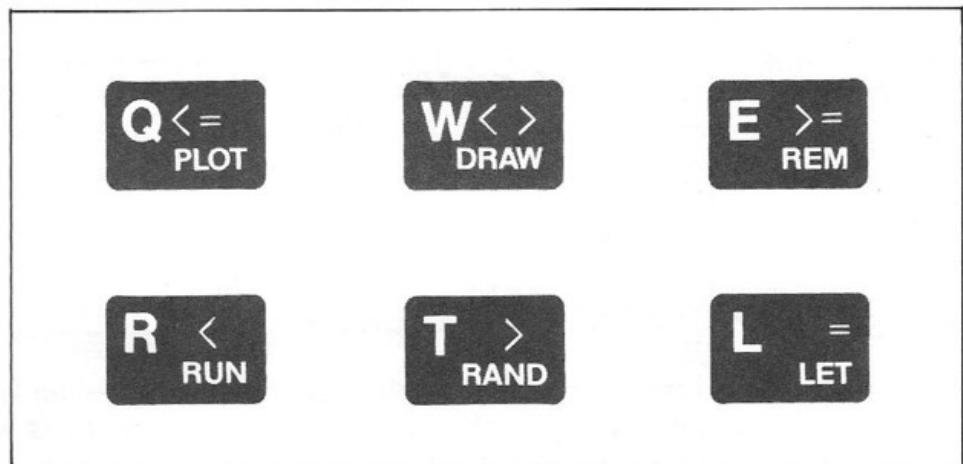
As another example, let's consider the condition

$zz > 6$

The condition will be true if variable zz has any value which is more than 6 (such as 6.01, or 82, or 999999). Otherwise the condition will be false. In particular, it will be false if zz is exactly 6.

You will find that all six relational symbols can be typed with single keystrokes on the SPECTRUM while holding down the SYMBOL SHIFT key. They are found on the following keys:

(Figure 6.1)



Conditions can also be written between pairs of numbers or pairs of strings. For example, $1 = 1$ is always true, and $6 < 3$ is always false. When the relations $>$, $<$, $>=$ and $<=$ are used between strings, 'less than' means 'nearer the beginning of the dictionary', so that:

"Elephant" $<$ "Mouse" is always true

"Paul" $>$ "Pauline" is always false

"Tiger" $>=$ "Tiger" is always true.

On the SPECTRUM, all the capitals come before the lower case letters. So $"Z" < "a"$ is true.

Experiment 6a

Let us suppose the computer has been asked to carry out the following instructions:

LET a\$ = "CHRIS"

LET c = 4

LET j = 9

Look at the conditions below. Are they true or false ? Write your answers in the boxes provided.

CONDITION	VALUE - True or False
c < 9	(T)
c > 7	(T)
a\$ <> "B"	
c <> j	
a\$ < "ANDY"	
a\$ > "CHRIS"	
j = 10	

Just as in LET commands, the quantities on either side of the relationship can be expressions and as complex as you like. You must be consistent, however, and compare numbers with numbers and strings with strings. If you write a condition with a string on one side and a number on the other, the SPECTRUM will not accept it.

Keeping the same value of a\$, c and j, work out each of the following conditions.

a\$ + "TOPHER" <> "CHRISTOPHER"	
4 > c	
c + j <= 13	
c + 5 = j	

You can check your answers at the back of the book in Appendix B.

Experiment 6a completed.

6.2 Conditional GO TO Commands

The way that we control programs in BASIC is to use the conditional GO TO command. This consists of:

- a) the keyword IF
- b) a condition
- c) THEN (a 'red' symbol which you type by holding down SYMBOL SHIFT and striking the G key)
- d) the keyword GO TO (just hit the G key)
- e) a label number

An example of a conditional GO TO command is

```
120 IF n > 47 THEN GO TO 170
```

The conditional command works very like the ordinary GO TO , but with an important difference; the jump only happens if the condition is true. For example, the command we have just given as an example would only make the machine jump to line 170 if the value of n was more than 47. Otherwise it would just obey the next command in sequence (probably 130).

Up to now, you have been used to knowing the exact values of all the string and numeric variables in your programs, and you may find the conditional GO TO command difficult to understand. Why should there be any question of whether n is greater than 47? The programmer defines his own variables so of course - it seems - he knows whether any condition is going to be true or not.

There are two occasions when conditions are necessary.

1. The conditional GO TO command may come in a loop, where a variable used in the condition has its value changed every time round. Thus the condition might be true for some values and false for others.
2. You are writing a program for someone else to use - such as the quiz programs included with some of the chapters. You don't know in advance how the user will answer - whether the answers will be right or wrong - and the program must react appropriately in every case. It will use a suitable condition to pick out the correct line of action.

In the programs you have written so far, your loops could only be stopped by drastic measures such as using BREAK SPACE (with or without CAPS SHIFT). When you include a conditional GO TO command in a loop, the loop can stop itself when it has been round a certain number of times.

Type in the following:

```
10 LET n$ = "Q"
20 PRINT n$
30 LET n$ = n$ + "C"
40 GO TO 20
```

When you run this program, you'll see the screen fill up with longer and longer lines of QCCCCCCCCCs and the scroll message at the bottom indicates that there is more to come! Stop the program (N or BREAK SPACE) and clear the screen. List the program and replace

new line:

```
50 STOP
```

You'll find the STOP symbol in red on the



key. Press down the SYMBOL SHIFT key and then STOP.

When you run the program again, it displays:

```
Q
QC
QCC
QCCC
```

and then stops. How does this work? It hinges on the condition

```
n$ <= "QCCC"
```

In the first three times around the loop the condition is true - because n\$ is QC, then QCC, then QCCC. On the fourth time round the loop n\$ is now QCCCC, which is alphabetically greater than QCCC. The condition is false and the jump to 20 doesn't happen. The computer continues on to line 50 and obeys the STOP command - which makes it end the program. Try giving the program different conditions to work on and see what happens. Make sure that the condition does become false at some point, though.

You could try the following conditions:

```
n$ <= "QC"
n$ <= "QCCCCCCCCCCCCC"
n$ <= "QCCQ"
```

You can use the same method of control with numerical variables. Here is an 'uncontrolled' program:

```
10 LET t = 12
20 LET v = 1
30 PRINT t;"*"; v; " = " t * v
40 LET v = v + 1
50 GO TO 30
```

Run this program and see what it does. Then stop it, list it and replace line 50 by

```
50 IF v <= 12 THEN GO TO 30
```

and add

```
60 STOP
```

Now run the program again. You should recognise it as the twelve times table. There is only one thing missing - the title or label for the display which should be placed above the columns of figures. A suitable heading could be 'Twelve times table', which you can insert using the PRINT command. 10 is the first label in the program so you will need to choose a label number before this.

Type in

```
5 PRINT "Twelve times table"
```

7 PRINT

and see the difference this makes to the appearance of the display.

In the next Experiment you will be writing some programs of your own. See if you can draw some conclusions from the two programs you have looked at so far:

A.

```
10 LET n$ = "Q"
20 PRINT n$
30 LET n$ = n$ + "C"
40 IF n$ <= "QCCC" THEN GO TO 20
50 STOP
```

B.

```
5 PRINT "Twelve times table"
7 PRINT
10 LET t = 12
20 LET v = 1
30 PRINT t; " * "; v; " = "; t * v
40 LET v = v + 1
50 IF v <= 12 THEN GO TO 30
60 STOP
```

Ignoring the heading commands in example B, you can see that both programs have certain features in common.

1. There is a variable which changes in a regular way as each loop repeats itself. In the first program it is n\$, in the second v. This variable is called a **control variable** and may be either a string or a number.

2. There is a command outside the loop which is only obeyed once and which sets the starting value of the control variable. In A this is:

```
10 LET n$ = "Q"
```

and in B it is

```
10 LET v = 1
```

3. Every time the control variable is changed, a command is obeyed. In both examples it is a PRINT command. This part of the loop - or body of the loop, as it is called - could include many different commands, all of which are obeyed every time the control variable changes.

4. The control variable grows by a certain value, or 'increment', every time the loop is repeated. n\$ gets longer by a C and v is increased by 1. The values can be changed in different ways. For example they could go up in steps of 2 or 10, and strings could be increased by four or six symbols. The values can also be decreased, by starting high and going down in a similar manner. The loop always includes a command which changes the value of the control variable each time round.

5. The control variable has a final value - QCCC and 12. When the loop containing this final value is reached, the program stops going back on itself and carries on with the next command. The last

Experiment 6b

In the following table, read through each program carefully - don't type them into the SPECTRUM! Fill in:

- the name of the control variable**
- the starting value**
- the final value (the value printed by the last command in each program)**
- the increment**
- the number of times the loop has been obeyed.** It might help you to jot down the value of the variable the first, second and third times round the loop and so on until the final value is reached - then to count the values.

Examples	a	b	c	d	e
A. 10 LET a\$ = "X" 20 PRINT a\$ 30 LET a\$ = a\$ + "Q" 40 IF a\$ <= "XQQQQ" THEN GO TO 20 50 PRINT a\$ 60 STOP					
B. 10 LET x = 0 20 PRINT x, x * x 30 LET x = x + 1 40 IF x <= 10 THEN GO TO 20 50 PRINT x 60 STOP					
C. 10 LET z\$ = "A" 20 PRINT z\$ 30 LET z\$ = z\$ + "BC" 40 IF z\$ <= "ABCBCBCBC" THEN GO TO 20 50 PRINT z\$ 60 STOP					
D. 10 LET m = 3 20 PRINT m, m * 2 30 LET m = m + 2 40 IF m <= 19 THEN GO TO 20 50 PRINT m 60 STOP					
E. 10 LET p = 13 20 LET q = 19 - p 30 PRINT p, q 40 LET p = p - 2 50 IF p <= 3 THEN GO TO 30 60 PRINT p 70 STOP					

When you have completed the table check your answers in Appendix B.

6.3 Designing Programs

The essence of good programming is careful planning. You would never expect a builder or a civil engineer to start constructing anything without detailed plans, otherwise houses would collapse and bridges not meet in the middle.

In the same way a professional programmer always thinks out and designs his programs with the utmost care.

Now the time has come to write your own programs, don't go near the SPECTRUM until the design is complete. Make sure you have plenty of paper and sharp pencils - and be prepared to use an eraser!

When you are planning out a loop for a program you must settle several essential points:

- a) the name of the control variable
- b) the type of control variable
- c) the starting value
- d) the final value
- e) the increment
- f) what should be included in the body of the loop

Look at this example which is already worked for you.

A British tourist travelling to Spain in the summer of 1983 could expect the following rate of exchange:

£1.00 Sterling : 165 Spanish Pesetas

We want a table to show the equivalent values of £s Sterling and pesetas. The amounts should be listed from £5 to £60, going up in steps of £5. The display should be set out as follows:

£	Spanish Pesetas
5	825
10	1650
15	2575

and so on.

Let's begin with the loop. The control variable will be a number which we call s. The starting value is 5, the final value is 60 and the increment is 5. The body of the loop must contain a command to print the value in £s, and the corresponding value in pesetas, which is 165 times more.

We can now set down the elements of the loop.

- | | |
|--------------------------------|----------------------------|
| 1. To set the initial value | LET s = 5 |
| 2. Body of loop | PRINT s, 165 * s |
| 3. Increment s | LET s = s + 5 |
| 4. Check if final value passed | IF s <= 60 THEN GO TO |

We have left the label number after GO TO blank because it still hasn't been decided.

What about the heading?

In

PRINT "£", "Spanish Pesetas"

Now we can write the complete program.

```
10 PRINT "£", "Spanish Pesetas"
20 PRINT
30 LET s = 5
40 PRINT s, 165 * s
50 LET s = s + 5
60 IF s <= 60 THEN GO TO 40
70 STOP
```

Experiment 6c

Now try these examples.

A. Write a program which displays a pattern of + signs, as follows:

```
+
++
+++
++++
```

and so on up to + + + + + + + + + +

B. Write a program which gives the equivalent of Italian lire for sums of British money between £10 and £30, going up in steps of £1. (2320 lire = £1)

When you have written and run these programs, check your results in Appendix B.

<i>Experiment 6c completed.</i>	
---------------------------------	--

6.4 What does '=' mean in BASIC?

This final section has been included to clarify an important point which might be causing you a little confusion - it is the use of the symbol =.

When you did mathematics at school, this symbol always meant 'has the same value as'. For example,

$$\begin{aligned}5 + 4 &= 9 \\ (a+b)(a-b) &= a^2 - b^2 \\ 3x + 4 &= 22\end{aligned}$$

In BASIC, the sign = is used in two different senses, neither of which is the same as the mathematical one.

a) In a LET command = means 'becomes'. The LET command is an instruction to the computer to calculate the value of the expression on the right and to put this value into the variable on the left. In

$$\text{LET } x = y + 2$$

the computer is not being *informed* that x has the same value as y + 2, but *commanded* to put the value of y + 2 into the variable called x.

LET r = r + 1 is a valid command but 'LET r + 1 = r' is not. Do you see why?

Also LET r = q is not the same as LET q = r (although both are valid commands in BASIC).

b) The other use of = in BASIC is as one of the six possible relationships between quantities:

e.g. IF x = 9 THEN GO TO 100
IF a\$ = "NO" THEN GO TO 90

As in the LET command, = is not a statement of fact - it is an instruction to the computer to work out whether the condition is true. In conditions it has the same logical effect as < or >.

To sum up:

In LET commands BASIC uses = in the sense of 'becomes'.

In conditions = means 'is the same as', but this may or may not be true.

The self-test program for this Chapter is called "quiz6"

Chapter 7

Program tracing
Tracing errors

7.1 Program Tracing

In Chapter Five, we looked at the ways the SPECTRUM can help you to edit programs, change them or correct mistakes. Now for the mistakes themselves. How can they be identified when they occur?

So far, the programs that you have written have been short and straightforward. As you progress with the course, you will be designing and writing longer and more complicated programs - and the opportunity for making blunders will be that much greater. For blunders there will be! Mistakes or 'bugs' in programs afflict everyone, from the complete novice to the experienced professional. If your program doesn't do what you expect, you mustn't feel angry or discouraged or leave the program half-working, saying to yourself, 'Well, it's almost going.' A program that almost goes is about as impressive as an aeroplane that almost flies.

Mistakes in programs are of three main types: grammatical errors, execution errors and errors of design. The SPECTRUM will help you find mistakes of the first two types.

A. Grammatical errors

Errors of grammar are detected by the computer when you type ENTER. The machine does not accept your command but displays a flashing `?` at the point where it finds you've broken the rules of BASIC grammar. For example, if you forget the quote signs round a string:

```
LET a$ = werewolf
```

the SPECTRUM flashes up the question mark at the end of the line, indicating that it is expecting a string instead of a long numeric variable name. Sometimes the `?` will not correspond to the real error; for example, if you forget the \$ sign (as in `LET a = "werewolf"`), the question mark still comes at the end of the line because the machine expects a number!

B. Errors discovered during the execution of a program

When the SPECTRUM stops carrying out a BASIC program, it could be for one of several reasons: the program has come to a natural stop, it has been halted by an outside agency or there is a command in the program which cannot be obeyed. The various messages or reports which are displayed at the bottom of the screen give you an indication as to what has happened.

If you look at the following sequence of commands

```
10 LET d = 8
20 PRINT d + a
30 GO TO 10
```

you will see that the variable `a` has not been given a value. Try entering this program into the SPECTRUM. When you run it, you will see the following message appear:

```
2 Variable not found, 20 : 1
```

which means that the SPECTRUM has been unable to obey the command labelled 20 as the variable `a` doesn't exist.

These error messages are very helpful for telling you when and where you have gone wrong. **Appendix D** at the back of the book gives you a list of all the ones you are likely to encounter during the course.

C. Errors of Design

up with the wrong result. You have studied the listing for hours, it seems, but you can't understand why it doesn't work. You are up against a blank wall!

This chapter introduces you to a method which you can use to get yourself out of this predicament. It is called **Program Tracing**.

Program tracing does not depend on flashes of inspiration or super intelligence - rather, the complete opposite. What you have to do is to imagine that you are the computer and work patiently through all the commands one at a time, finding out what actually does happen at each stage rather than what you think ought to happen.

To do this you should have some idea of how a computer functions. What happens to a program when you type it in ?

1. The program itself is stored in the memory in almost exactly the same form as you entered it.
2. Each variable used in the program is placed in a special part of the store and each has a value - a number or a string. Remember that a variable can only hold one value at a time.
3. The computer knows where it has got to in obeying the program. It remembers the label number of the next command it must obey, using a special variable called a program pointer.

As the program is obeyed, command by command, certain changes occur depending on which type of command has been carried out.

- * A PRINT command displays something on the screen.
- * A LET command creates a new variable - if one is needed - and puts a new value in it.
Both PRINT and LET move the program pointer on to the next command.

- * A GO TO command just moves the program pointer, breaking the normal sequence. For example:

```
50 GO TO 20
```

moves the pointer back to command 20.

- * A Conditional GO TO command has the same effect, except that the condition has to be worked out first. If it is TRUE, then the program pointer is moved to the place indicated by the GO TO command. If it is FALSE, then the GO TO command is ignored and the program pointer is moved on to the next command in sequence.

```
130 IF y = 20 THEN GO TO 160  
140 PRINT "YOU ARE WRONG"
```

In this example, if y has the value 20, then the condition is true and the program pointer is moved to 160. If y does not have the value 20, then the condition is false and the pointer is moved on to 140, the next command in sequence.

- * A STOP command ends the program and returns control to the user.

diagram below. Copy out the program accurately - otherwise the exercise will be pointless. Write the number of the first command to be obeyed in the program pointer box - in this case, it is 10.

PROGRAM	DISPLAY
10 LET x = 9 20 PRINT "Unit one = "; x 30 LET x = x * 2 40 LET y = x + 20 50 PRINT "Unit two = "; y 60 STOP	
PROGRAM POINTER: 10 VARIABLES:	

You now 'become' the computer and follow the program methodically. Don't take any short cuts. Start with 10, the command indicated by the program pointer. It is a LET command, giving a value for variable x. Does x already exist? No, so you create the variable x at the bottom of the chart, following it with a colon and the value 9. The program pointer is moved on to the next command in sequence. Cross out the 10 and write in 20.

PROGRAM	DISPLAY
10 LET x = 9 20 PRINT "Unit one = "; x 30 LET x = x * 2 40 LET y = x + 20 50 PRINT "Unit two = "; y 60 STOP	
PROGRAM POINTER: 10 20 VARIABLES:	

Work out the next command in the same way. It is a PRINT command which displays the string "Unit one=" and the value of x. Write down everything the command displays, putting it in the DISPLAY section of the chart. To find out what it puts for x, refer to the VARIABLES section where you'll find the latest value: 9. The program counter is moved one step on to 30.

PROGRAM	DISPLAY
10 LET x = 9 20 PRINT "Unit one = ";x 30 LET x = x * 2 40 LET y = x + 20 50 PRINT "Unit two = ";y 60 STOP	Unit one = 9
PROGRAM POINTER: 10 20 30 VARIABLES: x : 9	

The next command, 30, gives the variable x a new value. Work out this expression using the old value of 9, then cross it out and insert the new value. Command 40 creates a new variable, y. Continue tracing in this way until the end of the program. The chart now looks like this:

PROGRAM	DISPLAY
10 LET x = 9 20 PRINT "Unit One = ";x 30 LET x = x * 2 40 LET y = x + 20 50 PRINT "Unit two = ";y 60 STOP	Unit one = 9 Unit two = 38
PROGRAM POINTER: 10 20 30 40 50 60 VARIABLES: x : 18 y : 38	

Let's now look at a program with a simple loop.

```
10 LET a = 1
20 PRINT a, a * a
30 LET a = a + 1
40 IF a < 4 THEN GO TO 20
50 STOP
```

The first three commands are obeyed in sequence and your chart should look like this:

PROGRAM	DISPLAY
10 LET a = 1 20 PRINT a, a * a 30 LET a = a + 1 40 IF a < 4 THEN GO TO 20 50 STOP	1 1
PROGRAM POINTER: 10 20 30 40 VARIABLES: a : 2	

When you reach command 40, which is an IF condition, look at the value of A. It is 2, less than 4, so the condition is true and the program pointer is set back to 20.

PROGRAM	DISPLAY
10 LET a = 1 20 PRINT a, a * a 30 LET a = a + 1 40 IF a < 4 THEN GO TO 20 50 STOP	1 1
PROGRAM POINTER: 10 20 30 40 20	

The trace continues and the loop is repeated until at last the condition is false and the end of the program is reached. The chart's final appearance is:

PROGRAM	DISPLAY
10 LET a = 1	
20 PRINT a, a * a	1 1
30 LET a = a + 1	2 4
40 IF a < 4 THEN GO TO 20	3 9
50 STOP	
PROGRAM POINTER: 10 20 30 40 20 30 40 20 30 40 50	
VARIABLES: a : 1 2 3 4	

Experiment 7a

Here are two programs for you to practice tracing. Use a pencil in case you need to erase any mistakes.

A.

PROGRAM	DISPLAY
<pre> 10 LET a = 2 20 LET b = 5 30 LET c = a + b 40 LET d = b - a 50 PRINT a,b,c,d 60 STOP </pre>	
PROGRAM POINTER: 10 VARIABLES:	

B.

PROGRAM	DISPLAY
<pre> 10 LET s = 1 20 PRINT "Tinker" 30 PRINT "Tailor" 40 PRINT "Soldier" 50 PRINT "Sailor" 60 LET s = s + 1 70 IF s < 3 THEN GO TO 40 80 STOP </pre>	
PROGRAM POINTER: 10 VARIABLES:	

7.2 Tracing Errors

So far your program tracing has only dealt with programs that are correct. How can it help when you have a program that doesn't work?

Carry out the trace in the way described above, step by step. After writing down the effect of each command, allow your natural intelligence to break in to ask yourself - is this what should happen? If it is, then continue with the trace. If not, then you will usually see straightaway where the fault lies.

Look at this example. You want to display a table of squares, numbers multiplied by themselves, from 1 to 12, and have written the following program.

```
10 PRINT "Squares"
20 LET n = 1
30 LET n = n + 1
40 PRINT n, n * n
50 IF n <= 12 THEN GO TO 40
60 STOP
```

When you run this program, instead of the two columns of figures you expect, all you get is:

```
Squares
2      4
2      4
2      4
and so on.
```

You may have already spotted the mistake, but let's assume that you can't see what is wrong. After all, there is a loop and a PRINT command to display every line. What is happening?

Set out your program trace chart:

PROGRAM	DISPLAY
10 PRINT "Squares"	Squares
20 LET n = 1	2 4
30 LET n = n + 1	2 4
40 PRINT n, n * n	
50 IF n <= 12 THEN GO TO 40	
60 STOP	
PROGRAM POINTER: 10 20 30 40 50 40 50	
VARIABLES: n : 1 2 3 4	

You will soon realise that the program is working its way round the loop without changing anything because the `LET n = n + 1` is outside the loop. If you bring it inside the loop by changing the label number in the `GO TO` command, you will get:

```
10 PRINT "Squares"
20 LET n = 1
30 LET n = n + 1
40 PRINT n, n * n
50 IF n <= 12 THEN GO TO 30
60 STOP
```

This will now display the correct result.

Program tracing needs patience and a methodical approach. Tracing can be extremely boring but it will usually succeed in showing you where you are going wrong when all else fails.

Occasionally, however, program tracing will not work. Your program may be too long and tracing it step by step would be far too time-consuming. We'll describe appropriate methods to handle this later in the course. Program tracing is also useless if you have misunderstood some aspect of BASIC.

Suppose, for example, you think that the relationship `<` (is less than) means 'is more than'. This is not as silly as it seems - a lot of people make this mistake.

When you write a program using this relationship in a conditional `GO TO` command, such as

```
IF z < 3 THEN GO TO 100
```

you will never be able to understand why the computer always jumps the wrong way. The answer you get by tracing will not be the same as the one the computer gives you - and you won't see why.

If you do find that your program trace keeps giving different results to the ones displayed by the SPECTRUM, then there is probably something about BASIC programming that you haven't grasped properly. Try and get some help from someone who knows BASIC - there are plenty of such people around - or, failing that, go back to the very beginning of this course and check everything you know about the language, item by item. This will usually clear up the misunderstanding.

The very last thing to blame for a fault in your program is the SPECTRUM itself. Micro-computers hardly ever break down, and if they do, you usually know instantly. The copyright message on the screen might not appear or tapes will refuse to be loaded. If you need to take your computer to be repaired, indicate why you think it's not working and which program it won't run.

Program tracing is a powerful tool for tracking down errors - but you must want to get rid of those errors to start with. Don't be like the programmer who says that his program is 'virtually' working because 'virtually' means 'not'. Start now as you mean to go on, turning out well-designed programs, which you can be proud of - and which work!

Experiment 7b

Here are two programs with mistakes for you to find and correct.

This program was written to give the Fahrenheit equivalents of Centigrade temperatures, starting at 15° Centigrade and going up in steps of 1° to 30° Centigrade. The relationship between the Fahrenheit and Centigrade scales is expressed by the formula

$$f = 1.8 * c + 32$$

```
10 PRINT "CENT", "FAHR"
20 PRINT
30 LET c = 15
40 LET f = 1.8 * c + 32
50 PRINT c, f
60 LET c = c + 1
70 IF c > 31 THEN GO TO 40
80 STOP
```

This program is supposed to display a triangle of stars.

```
10 LET a$ = "*"
20 PRINT a$
30 LET a$ = "***"
40 IF a$ <> "*****" THEN GO TO 20
50 STOP
```

Experiment 7b completed.	
--------------------------	--

Experiment 7c

The program on tape "c7prog" is supposed to display the 9 times table - but it doesn't. Load the program, find and correct the mistakes.

Experiment 7c completed.	
--------------------------	--

Chapter 8

The INPUT command

8.1 INPUT Commands

In this course, you have progressed from typing simple commands to writing and running stored programs which can obey instructions many times over.

Your programs, however, have all been interesting exercises rather than useful tools. No matter how many times you run them, they always give the same results and cannot adapt themselves to changing conditions. For example, look at the program which we discussed in Chapter 6, for converting Sterling to Spanish Pesetas. You will remember that the program assumed an exchange rate of 165 Pesetas to the Pound. Here it is again:

```
10 PRINT "£", "Spanish Pesetas"
20 PRINT
30 LET s = 5
40 PRINT s, 165 * s
50 LET s = s + 5
60 IF s <= 60 THEN GO TO 40
70 STOP
```

Suppose you tried to persuade a professional currency dealer to use this program. Your demonstration run would go superbly as long as the exchange rate really was exactly 165 Pesetas to the Pound. If the rate moved even a little, the program in its present form would be useless. Of course, you could always change command 40 to read,

```
40 PRINT s, 163.5*s
```

or

```
40 PRINT s, 171*s
```

but you could hardly expect the dealer - who knows nothing about programming - to ring you up and ask you to fix his program every time that the rate was changed. He wants a program that will work for any rate of exchange, a flexible, useful and adaptable program that runs without constant attention from a programmer.

Clearly, what is needed is a new facility to allow the user - not the programmer - to give the computer information. Suppose the foreign exchange dealer could tell the computer the correct exchange rate for Pounds and Pesetas whenever he ran the program. The exchange program would become immediately and generally useful - which it certainly wasn't before.

Such a facility is provided by the INPUT command. This command allows information to be supplied directly to the computer by the user, when the program is already running. The computer then processes this information to produce the final result.

As an introduction to the INPUT command, type in the following little program:

```
10 INPUT "Type a number ";a
20 PRINT "Twice your number is ";2*a
30 STOP
```

When you run this program, it will display the message, 'Type a number' at the bottom of the screen. Key in any number you like, and press the ENTER key. The machine will immediately double your

gave you the right answer, even though we - the authors of this book - could not know in advance what number you would type. In our program the number was expressed as a variable called 'a', and its value was set to the number you typed on the keyboard. In this context you are the user, and we are the programmers.

Next, you should note that the INPUT command contained a short message (in quotes) which told you what sort of information the computer was expecting. This message is called a 'prompt'.

Finally, you may have noticed that when the computer obeys an INPUT command, it just stops and waits for your reply. There is no way for it to get to the next command until you have typed your answer and hit the ENTER key.

When you design a program which is to be used by someone else, you should decide in advance which quantities you know and which you expect the user to supply every time he runs the program. You allocate variable names to the unknown quantities. You then write the program, using these variables in the appropriate places. To set the values of the variables, you include INPUT commands, which will extract the necessary information from the user when the program is run.

Let's see how this scheme works in practice. In the exchange program, the only unknown is the rate of exchange. We begin by choosing a suitable name for this quantity. One possibility is 'rox'. Now we modify the existing program. First, we insert an INPUT command to let the user enter the day's rate of exchange. The rate is actually used inside the main loop of the program which starts at line 40, so the INPUT command must come before this line. 35 will be a suitable line number. The command will be:

```
35 INPUT "Today's rate ? ";rox
```

Next, we find the place where the rate of exchange is actually used, and replace the fixed value of 165 by the name of the variable:

```
40 PRINT s, rox * s
```

instead of:

```
40 PRINT s, 165*s
```

In its new version, the program now reads:

```
10 PRINT "£", "Spanish Pesetas"  
20 PRINT  
30 LET s = 5  
35 INPUT "Today's rate ? ";rox  
40 PRINT s, rox * s  
50 LET s = s + 5  
60 IF s <= 60 THEN GO TO 40  
70 STOP
```

Run the program several times and observe that it can adapt to any rate you choose to give it, even patently ridiculous ones. The program is now useful and will work by itself in any given situation.

Let's have a look at some further details of the INPUT command.

Clear the SPECTRUM with NEW and type in the following:

```
10 INPUT "What is your name ?";n$
20 PRINT "Greetings, ";n$
```

Run this program and see what happens. You will notice that INPUT works with strings as well as with numbers, but you must specify a string variable in the INPUT command. What the user types in will then be enclosed automatically in double quotes and the computer will assume it to be a string. If you said your name was 1234, this would be taken as a string and displayed accordingly. This doesn't work the other way around. If you try to enter anything except a number in response to an INPUT command when the SPECTRUM is expecting a number, you will usually get the error message:

variable not found

For example, if you typed 'twenty one' in response to the following

```
10 INPUT "How old are you ?";a
```

the SPECTRUM will not accept it. You should know about one exception to this rule. If you actually type the name of a variable which already exists in the program the INPUT command will simply take the value of that variable and use it instead of a number. For example, consider the following program:

```
10 LET squid = 8
20 INPUT "Any number please ? ";octopus
30 PRINT squid,octopus
40 STOP
```

If the user types - say '45', the machine replies:

8 45

just as you'd expect. If he types 'help' the computer will come up with the usual fault:

variable not found

but if he actually types 'squid' he will get:

8 8

In practice this feature is of limited use since you don't normally tell the user the names you have chosen for your variables. One possible application is in letting the machine understand the names of the numbers like 'one', 'two' and so on.

Prompts in INPUT commands must be enclosed in double quotes, and there must be a comma or semicolon immediately after. See for yourself what happens to the cursor when you type the following command, first using a semi-colon, then a comma:

```
10 INPUT "What is your name ? ";n$
10 INPUT "What is your name ? ",n$
```

When you use prompts, make sure that you insert a couple of spaces

```
10 INPUT "Give two numbers ";a,b
20 PRINT "Sum = "; a + b
```

The user must press ENTER after each value is typed in. It is possible to specify any number of variables at a time with INPUT, but it is usually best to stick to one or two to avoid confusion. In the command, the names of the variables are separated by commas.

Sometimes you may want to stop a program which is obeying an INPUT command and patiently displaying the cursor. This is quite difficult to do as CAPS SHIFT and BREAK SPACE together will not work. What you must do is to type the symbol STOP (SYMBOL SHIFT and A) making sure it is the first item in your reply. If the computer is waiting for a string, you will have to delete the first " before typing STOP.

Before you start to write the programs in Experiment 8a, here are some points to remember when using INPUT.

When you are designing a program for someone else to use, try to put yourself in their shoes. The user and the programmer are two completely different people whose only point of contact is the TV screen. There is no way that the user can get back to the programmer if he doesn't understand what is wanted of him. He certainly can't be expected to list the program to see what it does.

You must try to make the instructions on the screen and the results of the program as clear as possible. Use as many PRINT commands as you think you will need. (Sometimes you may even need to tell the user to press ENTER). Make sure that the words don't get split up between two lines - this might cause some misunderstanding. Check your spacing in strings; "Hello, Mike!" looks much better than "HelloMike". It often helps to repeat the information you have gleaned from the user, putting it on the main part of the screen. This is because any INPUT statement obliterates the previous one, and users feel more confident if they can see what they have actually said. One way of doing this is shown below:

```
20 INPUT "Name ? "; a$
30 PRINT "You say your name is ";a$
40 INPUT "Age ? ";n
50 PRINT "You claim to be ";n;" years old."
```

When the program is written, test it out on yourself and then try it out on someone else. If this person has to ask you what to do or what the results mean, then your program is obviously not working as it should and you must think again about the design.

Experiment 8a

Write a program to display a multiplication table for a value chosen by the user.

Experiment 8a completed.

Experiment 8b

A pound of apples costs 37p and a pound of grapes, 54p. Write a program to ask how many pounds of each kind of fruit are wanted and then display the total amount of money required.

Experiment 8b completed.

Experiment 8c

Write an inquisitive program which asks the following questions about any married couple you happen to know:

'What is her name?'

'How old is she?'

'What is his name?'

'How old is he?'

and comes up with a statement like:

'Albert is 2 years older than Jemima.'

Remember to allow for all possibilities. Avoid results like:

'Dennis is -5 years older than Margaret'

or

'Richard is 0 years older than Elisabeth'

Get your program to rephrase them into conventional English. Check your answers in Appendix B.

Experiment 8c completed.

Chapter 9

Flow charts
Robust programs

One of the most exciting features of programming is the enormous number of interesting and useful things it can get a computer to do. The same machine can play music, teach a child, play a game, calculate a payroll or monitor a sick patient. Yet the programs which instruct the computer to perform all these tasks are only composed of simple commands and statements. The skill of the programmer lies in knowing how to combine these commands and statements to achieve his purpose.

9.1 FLOW CHARTS

So far we have used six BASIC commands. They are:

PRINT LET GO TO STOP IF INPUT

The command which gives programs the greatest flexibility is the IF command. Up to now it has been used to control loops but it can also be used to test information supplied by the user and direct the computer how to proceed.

Let us imagine that you are in charge of the Grand Palace Cinema in Transylvania (which specialises in horror films). You plan to use a computer to help sell tickets in the Box Office. Patrons of the cinema can choose to sit either in the Circle - at 8 Crowns a ticket - or in the Stalls - at 6 Crowns a ticket.

Your first job is to write a 'box office' program which helps the clerk to charge the right amount for tickets. The program begins by asking each patron how many tickets he requires, then where he wants to sit. The computer then calculates the total amount due and displays this on the screen.

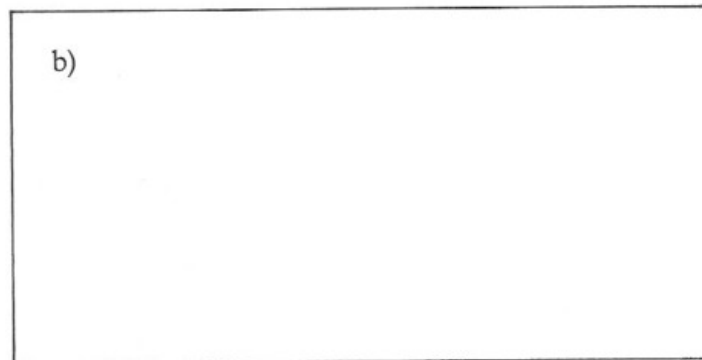
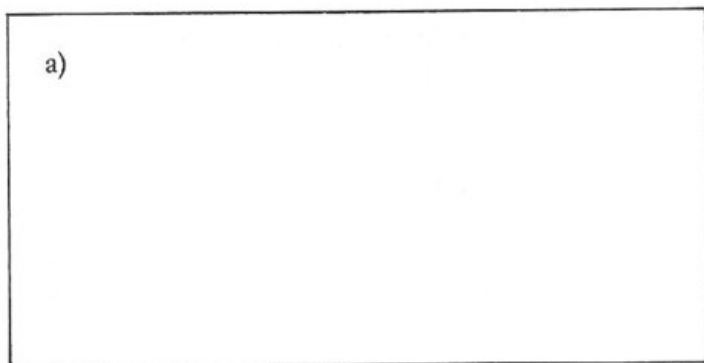
The program is given below. Study it carefully.

```
10 PRINT "GRAND PALACE CINEMA"  
20 PRINT "Tonight's Film"  
30 PRINT "BLOOD AND THUNDER"  
40 INPUT "How many seats? "; q  
50 INPUT "circle or stalls? "; p$  
60 IF p$ = "circle" THEN GO TO 90  
70 PRINT "You must pay "; 6 * q; " Crowns"  
80 STOP  
90 PRINT "You must pay "; 8 * q; " Crowns"  
100 STOP
```

The numeric variable q is used for the number of seats required and the string variable p\$ for the part of the cinema. The condition p\$ = "circle" is true if the patron types in "circle" in response to the question "circle or stalls?", false if he types in anything else. The expression 6 * q represents the total amount a 'stalls' patron must pay (q seats at 6 Crowns a seat). Likewise 8 * q is the total amount a 'circle' patron must pay.

When you have looked at the program, see if you have understood it correctly by predicting the final display for:

- a) A party of six who want seats in the Stalls.
- b) A couple who want seats in the Circle.

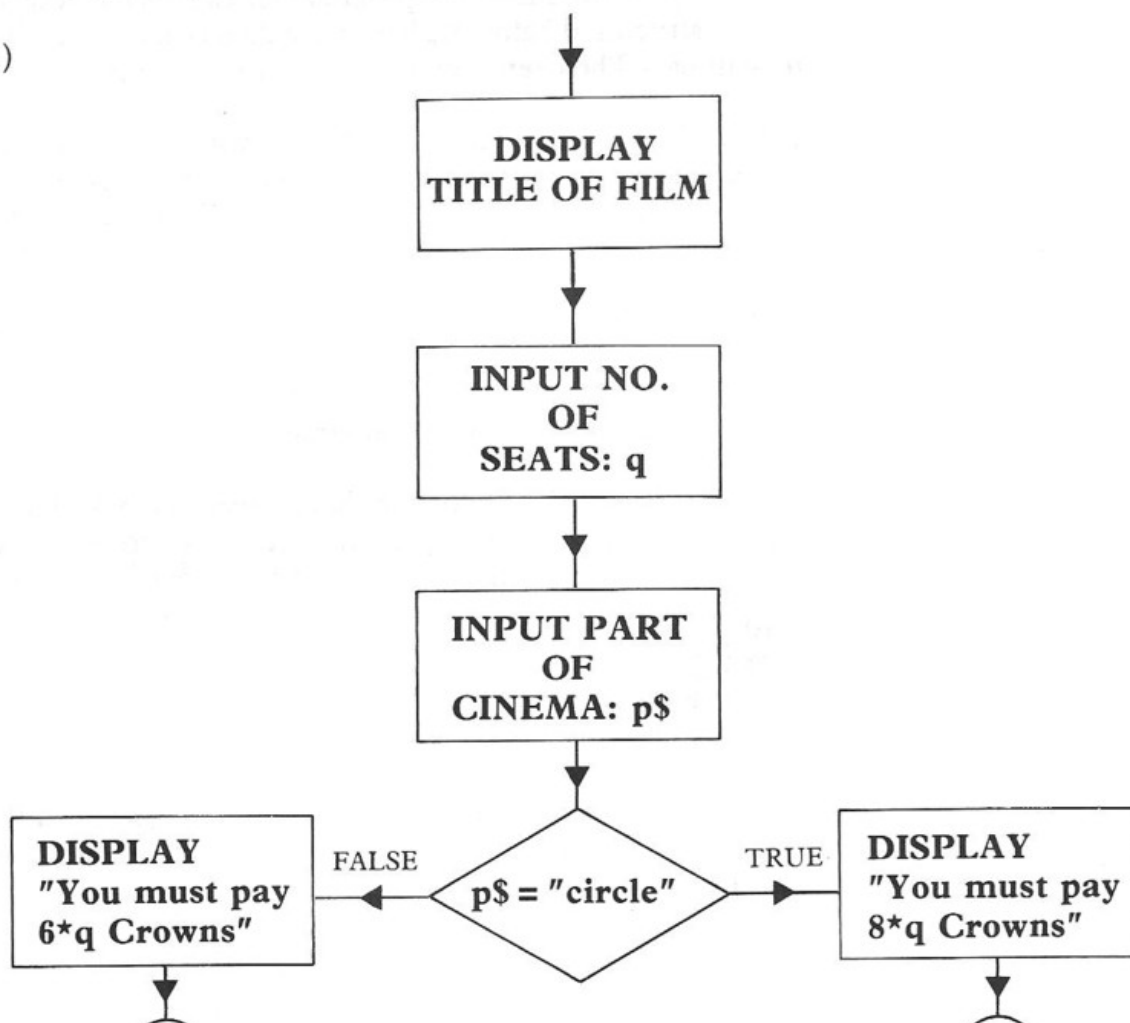


Now enter the program into the SPECTRUM, try it out on behalf of both sets of people and see if you were right. Remember to type the words 'circle' and 'stalls' exactly.

This is an extremely simple program. As we shall see later in this chapter, it has flaws. It is unlikely to be used in its present form, even in Transylvania. It does, however, show that the action of the computer need not be decided in advance by the programmer but can depend on information supplied by the user.

Decisions which are made by the computer can often be quite complicated. To help us plan them we use a special type of diagram called a flow chart. The stencil provided with the course will help you draw up your own flow charts neatly and accurately. The flow chart for the Box Office program is shown below.

(Fig 9.1)



A flow chart is a 'map' of your program. It consists of a number of different-shaped blocks connected by lines with arrows on them. The arrows indicate the route the program takes, or the order in which the commands are obeyed.

Types of blocks used in flow charts:

1. A rectangular or square box: this contains the description of a simple action, which can be represented by one or two BASIC commands.

The first three blocks in the flow chart (Fig. 9.1) are examples of this. The arrowed line shows the order in which the commands are obeyed.

2. A diamond: this contains a condition which can be true or false. It has one line going into it but two coming out, labelled True and False respectively. The diamond corresponds to an IF command which instructs the computer to test a condition. If the condition is true, then one course of action is followed, if false, another.

3. A small circle with STOP written in it: this is the terminal block, which instructs the computer to stop obeying a program.

4. A cloud: this doesn't form part of the example. It represents an action or group of actions which has either not been decided on or is too complex to set out in detail. Usually the 'cloud' can be extended into a complete flow chart of its own. Clouds are very useful in preliminary planning as the programmer can decide how his program will be structured before he has to decide on exactly which commands he will use. The overall picture isn't lost in a mass of detail.

A flow chart is a little like a board game with a counter or token being moved along from square to square in a certain order. If the token lands on a diamond it can go in one of two directions. If the condition is true, it follows the true line, if false, then the false line.

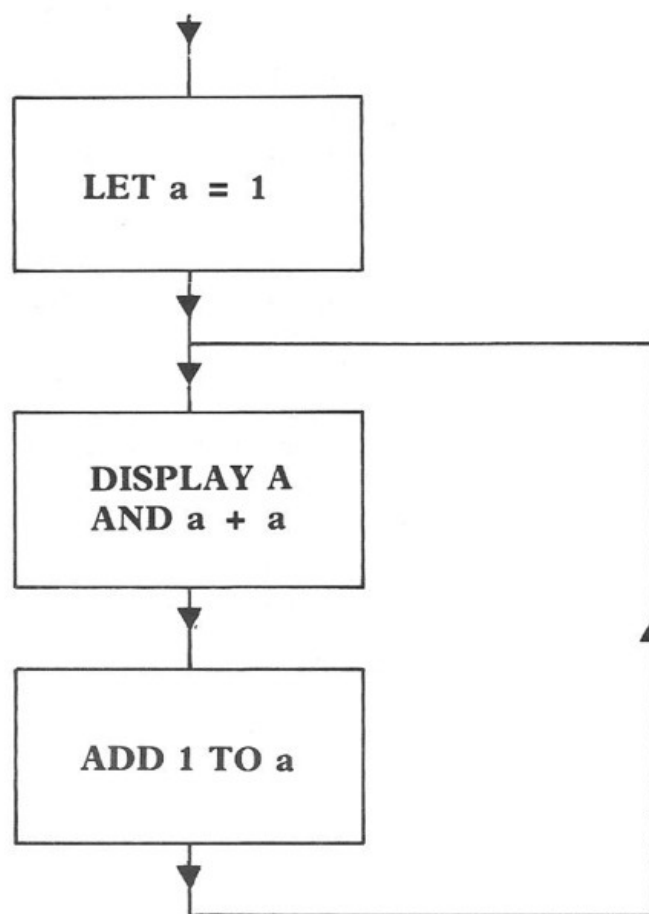
Just as a token can only be on one square at a time, so a computer can only obey one command at a time. There is only one token and the direction in which it moves, or sequence in which the commands are obeyed, is determined by the program.

The command GO TO does not have a special block. The function of GO TO is to tell the computer which command to obey next. You can see this indicated in the flow chart (Fig.9.2) by a connecting line.

```
10 LET a = 1
20 PRINT a, a + a
30 LET a = a + 1
40 GO TO 20
```

One final point: remember that the entry in the box of a flow chart is not supposed to be a BASIC command (although it may look like one). It is just an indication of what to do.

(Fig 9.2)



Experiment 9a

Draw a flow chart for the following program, (using the stencil provided with the course).

```
10 LET t = 1
20 PRINT t, 7 * t
30 LET t = t + 1
40 IF t <= 12 THEN GO TO 20
50 STOP
```

Check your answer in Appendix B.

9.2 Robust Programs

An important point to remember when you use the INPUT command is that you have no control over what the user is going to type. We have all met people who can't resist entering names like Donald Duck on official forms or giving their age as 195. These people are going to use your program. Will your program stand up to them - or will it meekly come up with some ridiculous result ?

Programmers should always consider the people who use their programs to be complete idiots! Sometimes people will make typing errors, sometimes they don't understand what is required of them, sometimes they are just silly. Whatever can possibly go wrong in an INPUT statement will go wrong - depend on it.

The programmer should be ready for all these possibilities and must try to prevent the worst errors. For example, in the Box Office program earlier in the chapter, the patron or user must:

- a) state how many tickets are wanted
- b) choose between the circle and the stalls

Suppose the patron types in 30006 for the first quantity. It is not a sensible value but the computer will not recognise this fact. It will continue with the next INPUT command:

circle or stalls ?

The word "upstairs" is typed in. The computer compares "upstairs" with "circle". The string is not the same, the condition is false and the user is told that he must pay for 30006 seats in the stalls at 6 Crowns each, a total of 180036 Crowns.

'Ask a silly question, get a silly answer' could be rephrased as 'Put in silly information, get silly results.' Standard computer jargon for this is 'GIGO', which stands for 'Garbage In, Garbage Out'.

To some extent, programmers can protect programs from really stupid or unlikely INPUTted information. The way to do this is to decide sensible ranges of values in advance. For example, if your program asked for a user's age, you might decide in advance that this is unlikely to be less than 0 or more than 125. If a value is given which falls outside these limits, then the program would ask the user to try again. The information which comes through this screening process is far more likely to give sensible results. The program has the quality of 'robustness' which means that it can stand up to all kinds of abuse.

We stay in Transylvania for the following example. Professor Vogelhoorn, a leading medical expert in that country, has proved scientifically that the correct weight for a healthy Transylvanian male is related to his height by the following formula:

$$w = 0.6 * h - 25$$

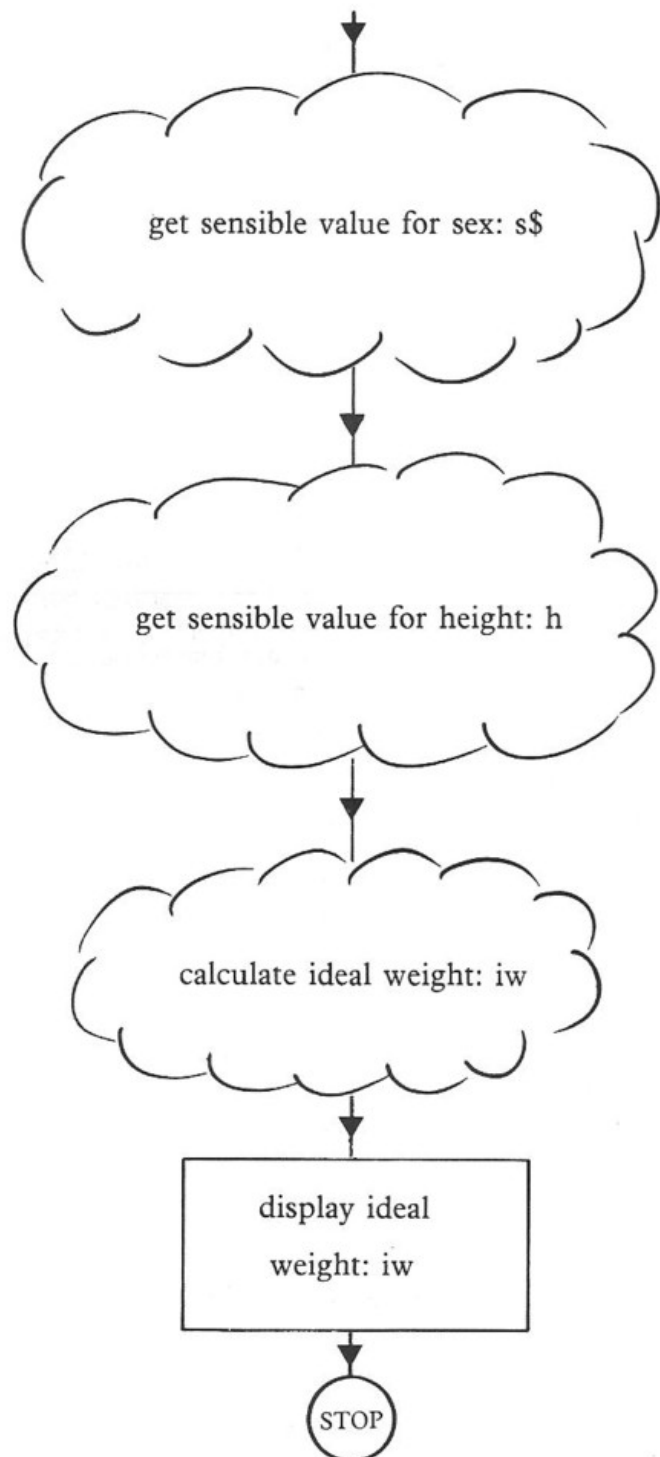
where w is the weight in kgs and h the height in cms. It means: allow 600 gm (0.6 kg) for each centimetre, and then take off 25 kgs. The Transylvanian female is somewhat lighter in build. The formula in her case is:

$$w = 0.5 * h - 25$$

We start off by choosing variable names as follows:

NAME	TYPE	PURPOSE
s\$	String	Sex
h	Number	Height
iw	Number	Ideal Weight

Our preliminary flow chart will look like this:

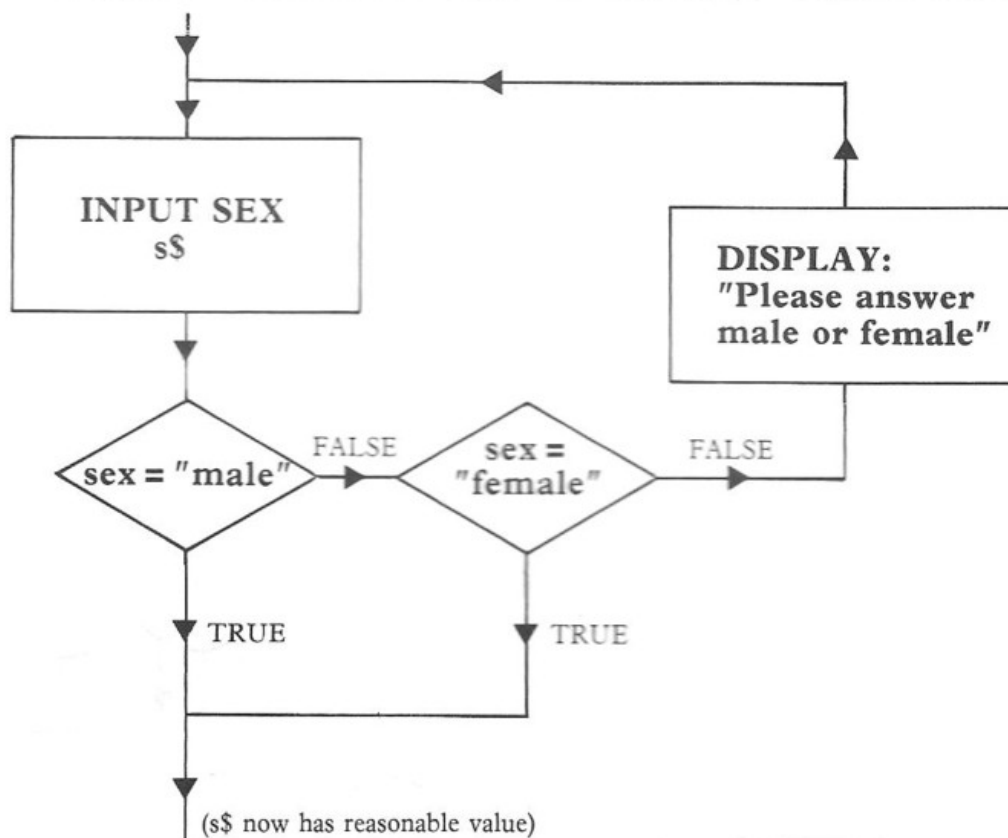


We use clouds in this first flow chart because we haven't yet decided what we actually mean by 'sensible'.

In the first cloud we need to find out the client's sex. There are many

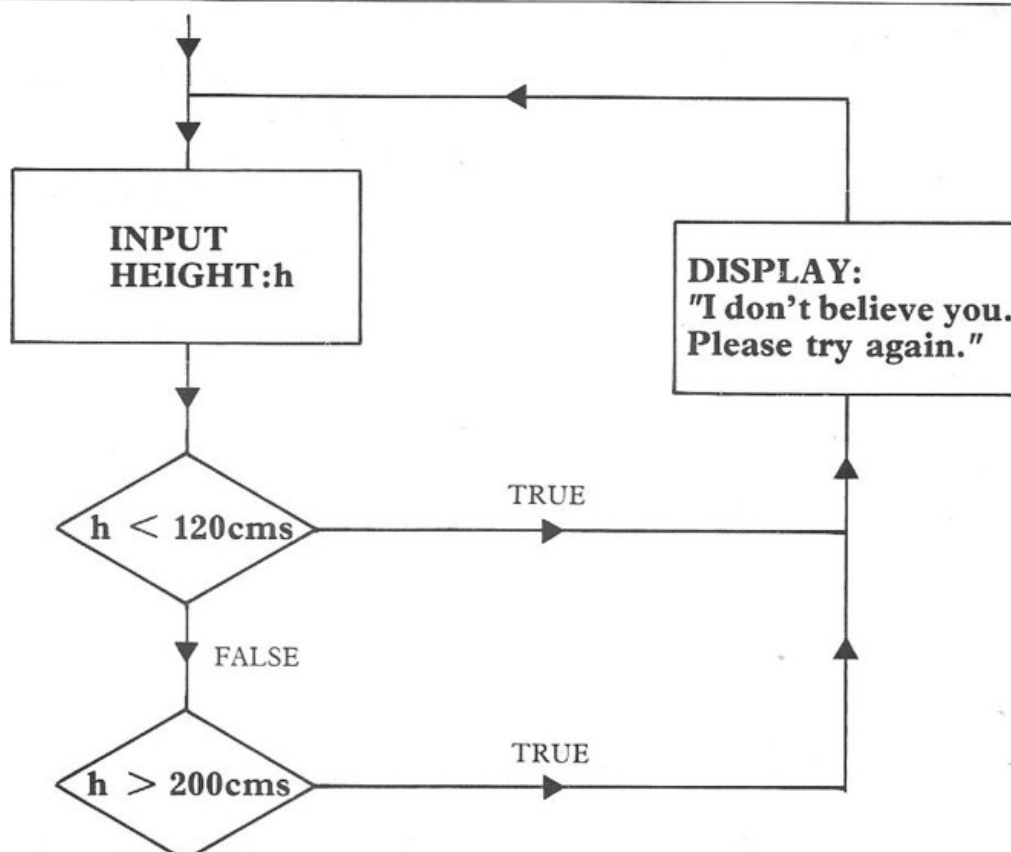
response is given in any other form, the program will ask for it to be repeated. The flow chart to replace the cloud will now look like this:

(Figure 9.3)



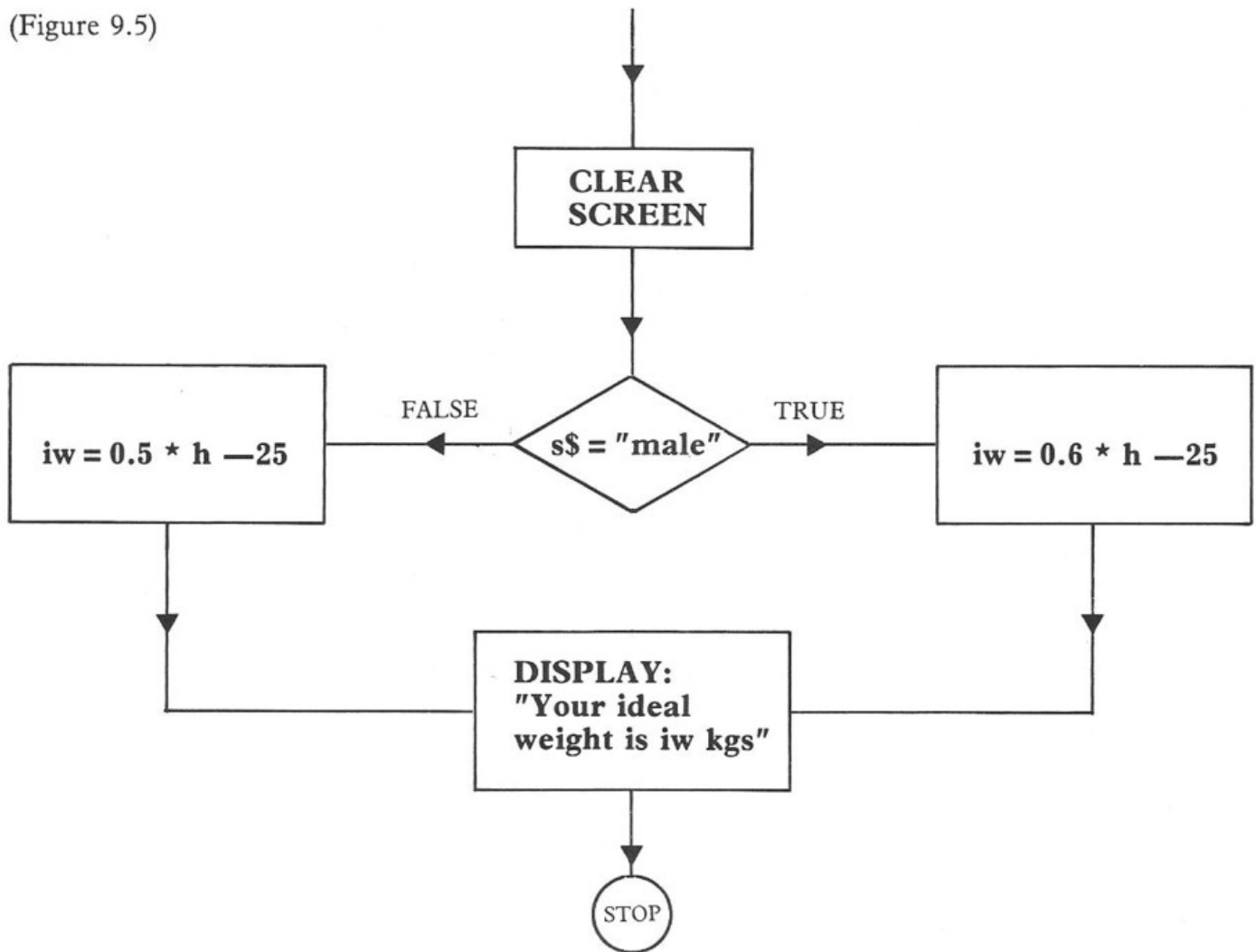
To get a sensible height, we shall decide to make the upper limit 200 cms and the lower limit 120 cms. The flow chart for this section could look like this:

(Figure 9.4)



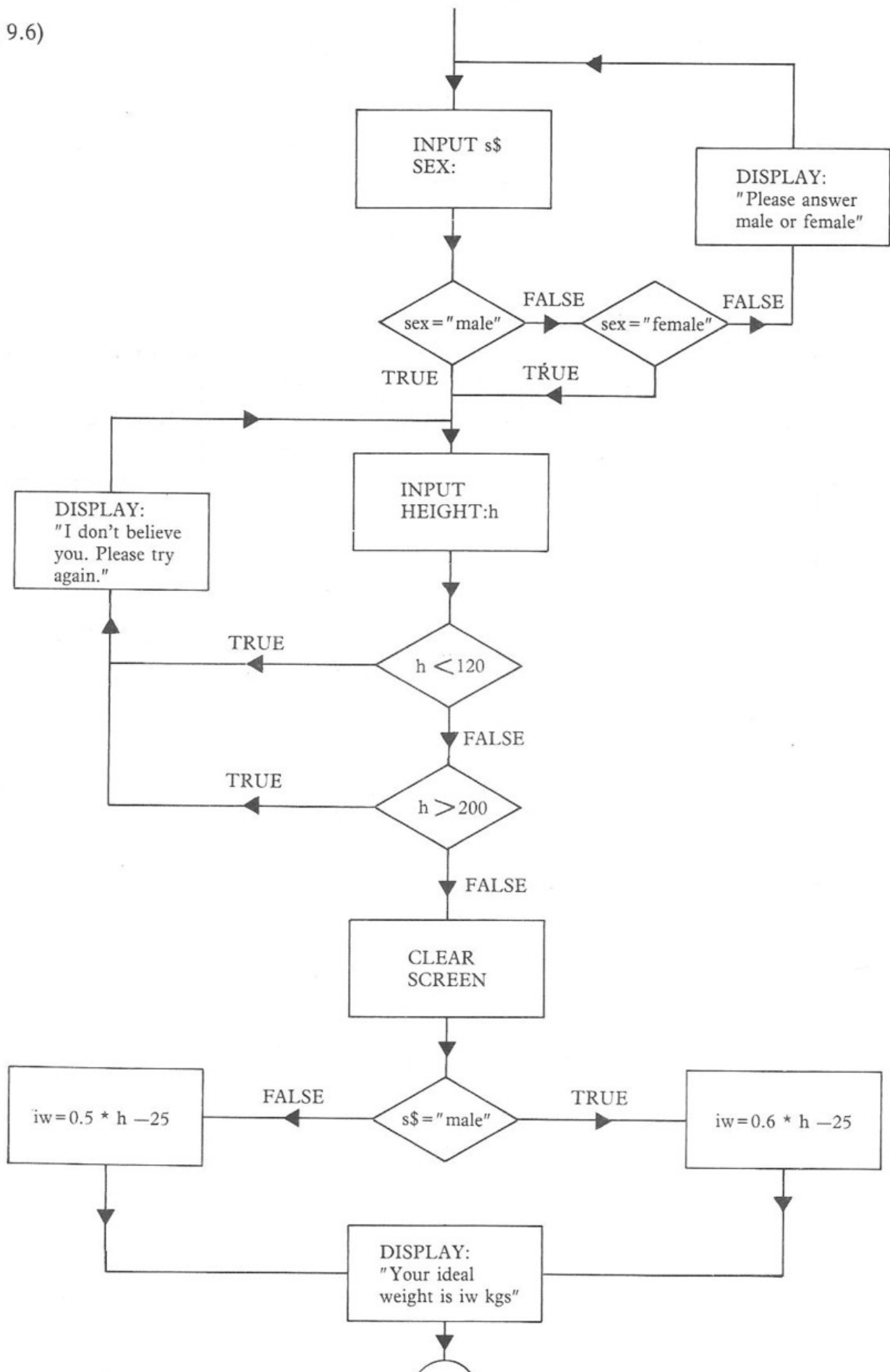
Now we have the calculation for the ideal weight to do. The screen at this point may contain all kinds of error messages such as 'Please try again,' so we clear it with a CLS command in the program. The correct bit of flow chart will be:

(Figure 9.5)



When we put the three sections together, and add a title, our flow chart will be complete.

(Fig 9.6)



Once the main flow chart or set of flow charts has been drawn up, translating it into a program is straightforward.

```

10 PRINT "KNOW YOUR IDEAL WEIGHT"
20 INPUT "male or female ? "; s$
30 IF s$ = "male" THEN GO TO 70
40 IF s$ = "female" THEN GO TO 70
50 PRINT "Please answer male or female."
60 GO TO 20
70 INPUT "What is your height in cms ? "; h
80 IF h < 120 THEN GO TO 170
90 IF h > 200 THEN GO TO 170
100 CLS
110 IF s$ = "male" THEN GO TO 140
120 LET iw = 0.5 * h - 25
130 GO TO 150
140 LET iw = 0.6 * h - 25
150 PRINT "Your ideal weight is "; iw; " kgs."
160 STOP
170 PRINT "I don't believe you."
180 PRINT "Please try again."
190 GO TO 70

```

Compare this program with the flow chart (Fig 9.6) and mark each box with the corresponding line number or numbers. Then type the program into the SPECTRUM and try it out. You will find it very much more robust than the Box Office program and far better at detecting and refusing silly answers. Note that the condition in command 110, `s$ = "male"`, is using "filtered" information, which means that if `s$` is not "male" then it must be "female". We don't need to check this again.

In the next experiment, you will be asked to design and write your own program. You may find this advice helpful.

1. Switch off the SPECTRUM and provide yourself with plenty of paper, pencils - and an eraser.
2. Look at the problem and work out what the results are likely to be with a couple of simple examples. Keep these as a check on your finished program.
3. Make a table or 'glossary' of the variables you decide to use. Put them under the headings - Name - Type - Purpose. For example, the Box Office program would have had the following glossary:

Name	Type	Purpose
q p\$	Number String	No. of seats Part of theatre (circle or stalls)

4. Draw up a flow chart for your program. You may have to do this several times over until you are satisfied with the result. Expect that this preliminary planning will take time and demand considerable effort.
5. Now translate the flow chart into BASIC. It should be easy - if it

the results match up with the ones you worked out earlier. If you want to keep the program, save it on tape and file away your glossary and flow chart.

This is the professional way to design and write programs. Many people, however, just key the program straight into the computer, patching and altering as they go along, until the overall structure of the program - if there ever was one - is totally lost. For short programs this method sometimes works, but for longer programs it leads to endless trouble. The results might only be right some of the time, and the programmer will find it impossible to work out where he went wrong. If you take 'short cuts' in designing your programs, you will take far longer to get them right.

Experiment 9b

You own a large, well-appointed camping site by the sea in France to which holiday-makers come in great numbers during the summer months.

The scale of charges per night is as follows:

Per car	5 francs
Per caravan or motorvan	25 francs
Per tent	10 francs
Per person	3 francs

Write a program which will ask the campers for details and display the total cost payable over a stay of one or more nights.

Try the program with the following groups of people.

- a) a family of four with a car and caravan staying for a week.
- b) two cyclists with a tent staying for two days.
- c) six people in a motorvan with two tents, staying for 4 days.

<i>Experiment 9b completed.</i>	
---------------------------------	--

Experiment 9c

Load and run the program "c9prog". When you have tried it a few times, list it, look at the code and draw up a flow chart and glossary for it.

Check your solution in Appendix B

Chapter 10

The FOR command

More about FOR commands

Null loops

Loops within loops

10.1 The FOR Command

All programs use loops. They are such an important feature in programming that BASIC gives you a special short-hand way to express their essential details.

You will remember that there are four vital parts which control a loop:

- * The choice of control variable
- * The starting value of the control variable
- * The final value for the control variable
- * The increment or amount by which the control variable grows each time round the loop.

For example:

```
10 LET a = 3
20 PRINT a, a * 2
30 LET a = a + 2
40 IF a <= 11 THEN GO TO 20
```

Another faster way of writing a loop uses the keywords FOR and NEXT. FOR comes at the beginning of the loop and specifies the control variable, the first value, the last value and the increment. NEXT serves to mark the end of the loop, and - to avoid confusion - mentions the name of the control variable. The lines between FOR and NEXT are called the 'body' of the loop. There may be any number of lines from zero up.

Here is an example which gives exactly the same result as the program above:

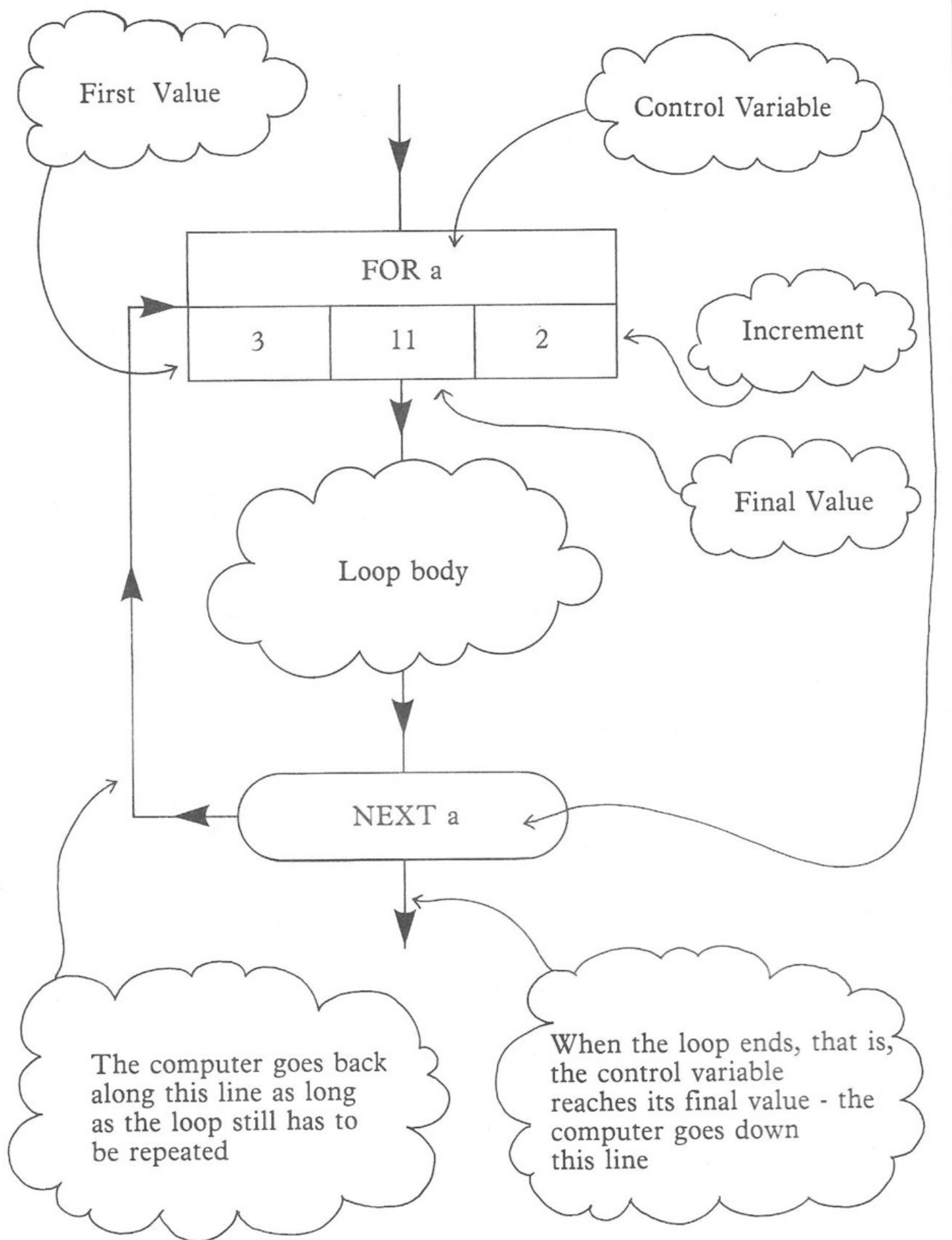
```
10 FOR a = 3 TO 11 STEP 2
20 PRINT a, a * 2
30 NEXT a
```

In both cases:

- The control variable is a
- The first value is 3
- The last value is 11
- The increment is 2

In flow charts, special blocks, which can't easily be confused with other types of action, are used to show loops.

Fig 10.1



Experiment 10a

In order to get the details of the FOR...NEXT command quite clear in your mind, predict what the two following programs will make the SPECTRUM display. Check your answers with the computer.

```
10 FOR s = 2 TO 14 STEP 3
20 PRINT s
30 NEXT s
40 STOP
```

```
10 FOR b = 24 TO 32 STEP 2
20 PRINT b, 32 - b
30 NEXT b
40 STOP
```

Experiment 10a completed.	
---------------------------	--

Experiment 10b

Convert the following program into a FOR ... NEXT version. Check your answer by running both programs on the SPECTRUM and make sure they both give the same result.

```
10 PRINT "Five times table"
20 LET n = 1
30 PRINT n; " times 5 = "; 5 * n
40 LET n = n + 1
50 IF n <= 12 THEN GO TO 30
60 STOP
```

Experiment 10b completed.	
---------------------------	--

Here are some important points you should remember about the FOR ... NEXT command:

A. The name of the control variable must always be a single letter.

B. If the increment or step size is 1, 'STEP 1' can be left out.

```
FOR n = 1 TO 12
is exactly the same command as
FOR n = 1 TO 12 STEP 1
```

C. The loop control can be made to count backwards if the STEP size is negative.

```
10 FOR y = 9 TO 5 STEP -1
20 PRINT y
30 NEXT y
```

will display the following:

```
9
8
7
6
5
```

D. Very sensibly, the SPECTRUM will not obey a loop at all if the final value is less than the starting value. For example, the program:

```
10 FOR f = 4 TO 2
20 PRINT f
30 NEXT f
40 STOP
```

will display nothing at all.

This also works the other way; for example

```
10 FOR q = 1 TO 4 STEP -1
20 PRINT q
30 NEXT q
```

won't be executed at all either.

E. The values in the FOR command needn't be numbers but can be expressions which include other variables. For example, the following program will display a line of stars across the screen, the number to be determined by the user.

```
10 INPUT "How many stars? " ;st
20 FOR d = 1 TO st
30 PRINT "*";
40 NEXT d
50 STOP
```

F. If the starting value, the increment and the final value don't fit together the control value will go as far as it can without exceeding the final value. For example:

```
FOR j = 1 TO 6 STEP 2
```

will produce the values 1, 3 and 5 (but not 6 or 7).

G. The control variable cannot be a string.

FOR a\$ = "*" TO "****" STEP "**"

is **not** BASIC. If you typed this in, as soon as you pressed ENTER, the SPECTRUM would flash up a query after the dollar sign.

Experiment 10c

Remembering these points, predict what the following programs will display and check your results on the SPECTRUM.

```
10 FOR b = 1 TO 5
20 PRINT b * b
30 NEXT b
40 STOP
```

```
10 FOR a = 4 TO 0 STEP -1
20 PRINT a
30 NEXT a
40 STOP
```

```
10 FOR m = 6 TO 3
20 PRINT m
30 NEXT m
40 STOP
```

```
10 LET a = 2
20 LET b = 24
30 LET c = 4
40 FOR d = a TO b STEP c
50 PRINT d
60 NEXT d
70 STOP
```

```
10 FOR g = 3 TO 10 STEP 2
20 PRINT g
30 NEXT g
40 STOP
```

Experiment 10c completed.	
---------------------------	--

10.2 More about FOR Commands

In order to focus on the details of the FOR and NEXT commands, we have deliberately kept the bodies of the loops very simple. In practice, however, a loop can include all sorts of commands and expressions; the one thing to remember is that they all get obeyed every time the computer goes round the loop.

Let's look at a practical example.

The Pyramids, tombs of the ancient rulers of Egypt, usually took many years to build. The Pharaoh, Ramentut, decreed that the base of his Pyramid should consist of 99 blocks by 99 blocks of polished granite. The next layer should be 98 blocks by 98 blocks, the third layer, 97 by 97 and so on until the final stone could be placed in position at the very top - making 99 layers of blocks in all to complete the Pyramid.

The Pharaoh's chief engineer had to calculate how many blocks of granite were needed to build the whole Pyramid. Let's see whether we can write a computer program to solve his problem.

First we must decide what variables we need.

We must keep a running total, to which will be added the number of blocks in every new layer of the Pyramid. The first layer consists of 99 blocks by 99 - or 9801 blocks. The second layer is 98 by 98 or 9604 blocks. Together the two layers total 19405 blocks. The third layer adds another 9409 blocks to the total - and so on. By the time we reach layer 99, we shall have the total of blocks needed for the whole Pyramid. A good name for this running total variable is *rt*.

As the program runs, it will deal with layer 1, then layer 2, finishing with layer 99. We need a variable to indicate which layer the program is dealing with at any moment, which we can call *z*. Since this variable will take all the values between 1 and 99, increasing by one every layer, it can be the control variable in a FOR command.

Our program glossary:

Name	Purpose
<i>rt</i>	To keep running total
<i>z</i>	Number of layer being dealt with at the moment (1 = bottom layer, 99 = top layer)
<i>s</i>	Number of blocks per side in current layer ($s = 100 - z$, because when $z = 1$, $s = 99$, when $z = 2$, $s = 98$ and so on).

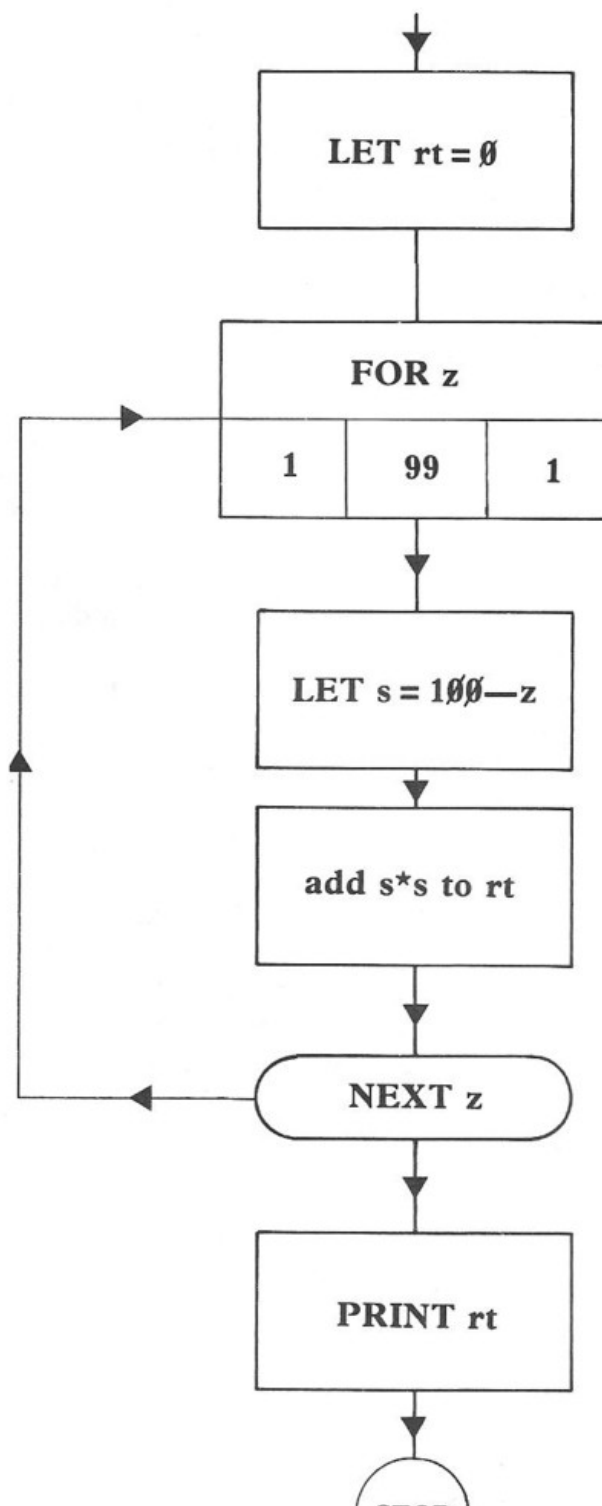
Now we must note down some of the actions the program needs to take.

Set <i>s</i> to $(100 - z)$	- set number of blocks per side
Add <i>s</i> squared to <i>rt</i>	- this adds the number of blocks for layer <i>z</i>
Print <i>rt</i>	- this displays the result
Set <i>rt</i> to 0	- starts the running total at zero
FOR <i>z</i> = 1 TO 99	- loop control for taking every
NEXT <i>z</i>	layer into account
STOP	

loop so we have to decide whether each command should be obeyed before the loop, inside the loop as part of the loop body or after the loop.

The command which sets rt to 0 should come before the loop. The command which sets s is obeyed once for each value of z , so it goes inside the loop. The same applies to the command which has to add the number of blocks in any particular layer to the running total. Finally, the PRINT command, which displays the total after all the calculations have been completed, comes after the loop. We can now draw up the flow chart for the program.

(Fig 10.2)



The program is:

```
10 LET rt = 0
20 FOR z = 1 TO 99
30 LET s = 100 - z
40 LET rt = rt + s * s
50 NEXT z
60 PRINT "Number of blocks = ";rt
70 STOP
```

Enter the program and see the answer for yourself.

Experiment 10d

Here is a problem for you. In competitive sports, such as skating or gymnastics, there is usually a panel of judges who award marks for such qualities as skill, interpretation etc. These marks are averaged out and the entrant with the highest average mark wins.

A program to work out this average would need to ask the user for the following information:

'Number of judges ?'	6	
'Marks ?'	5.6	Numbers
'Marks ?'	5.5	typed in
'Marks ?'	5.2	by user
'Marks ?'	5.4	
'Marks ?'	5.5	
'Marks ?'	5.2	

The average mark is 5.4

Displayed by the computer

Your task is to write the program for this problem. To help you we shall provide the glossary and all the commands. These are unlabelled and in the wrong order. You must first work out which are the loop commands and then slot the others in their right places. Run your program on the SPECTRUM to see if you are right. The correct answer is in Appendix B but don't refer to this unless you are really stuck.

Glossary	
Name	Purpose
j	- number of judges
p	- control variable for loop
rm	- used to add up the total marks
m	- marks from each judge

Program:

```

NEXT p
INPUT "How many judges ? "; j
INPUT "Marks ? "; m
PRINT "The average mark is ";rm/j
STOP
LET rm = 0
FOR p = 1 TO j
LET rm = rm +m

```

Experiment 10d completed.

10.3 Null Loops

Earlier, we mentioned that in certain cases there could be no commands at all inside a FOR ... NEXT loop. The purpose of an empty loop like:

```
30 FOR g = 1 TO 5000
40 NEXT g
```

is to keep the computer busy for a period of time. The computer starts by setting *g* to the value 1, and then it keeps counting up until it reaches 5000 without doing anything else at all. Such a loop is used to display a program title or a set of instructions for a fixed time before clearing the screen and doing something else. The computer takes about a second to count every thousand.

10.4 Loops within Loops

It is often useful for a program to include a loop inside another loop. You'll remember that the body of a loop may consist of any number of lines. There is nothing that says that these lines may not include another loop! The only restriction is that the two loops must not use the same control variable.

If you put a loop into another loop, then the whole of the inner loop is executed *every time* that the outer loop is obeyed once. For example, consider the simple loop:

```
FOR j = 1 TO 10
PRINT "+";
NEXT j
```

This will print 10 +'s. Now add a print command:

```
FOR j = 1 TO 10
PRINT "+";
NEXT j
PRINT
```

and you get 10 +'s on a line by themselves. Now put this loop inside another one:

```
FOR k = 1 TO 5
FOR j = 1 TO 10
PRINT "*";
NEXT j
PRINT
NEXT k
```

This program will print the line of +'s five times, giving you a block of 50 +'s. Try it!

Now try:

```
FOR q = 1 TO 15
FOR r = 1 TO q
PRINT "*";
NEXT r
PRINT
NEXT q
```

Experiment 10e

Write a program which invites the user to type ten names, one after the other, and then displays the one nearest the beginning of the alphabet.

(Hint: use a string variable which keeps track of the 'highest' name so far.)

Experiment 10e completed.	
---------------------------	--

Experiment 10f

The following section of code will display numbers up to a value *t* on a single line.

```
FOR j = 1 TO t
PRINT j; " ";
NEXT j
PRINT
```

For example, if *t* = 5, the display will be

```
1 2 3 4 5
```

Write a program which asks the user to input a value (not more than 12) and displays a triangle of numbers with this number of lines.

For example, if the user types 6 the program should display:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

Use a loop within a loop! This is quite easy if you don't panic.

Check your answers in Appendix B.

Experiment 10f completed.	
---------------------------	--

Chapter 11

The Keyboard

The Keyword mode

The Symbol Shift mode

The Extended mode

The Extended (Symbol Shift) mode

The Lower Case mode

The Caps Shift mode

The Caps Lock mode

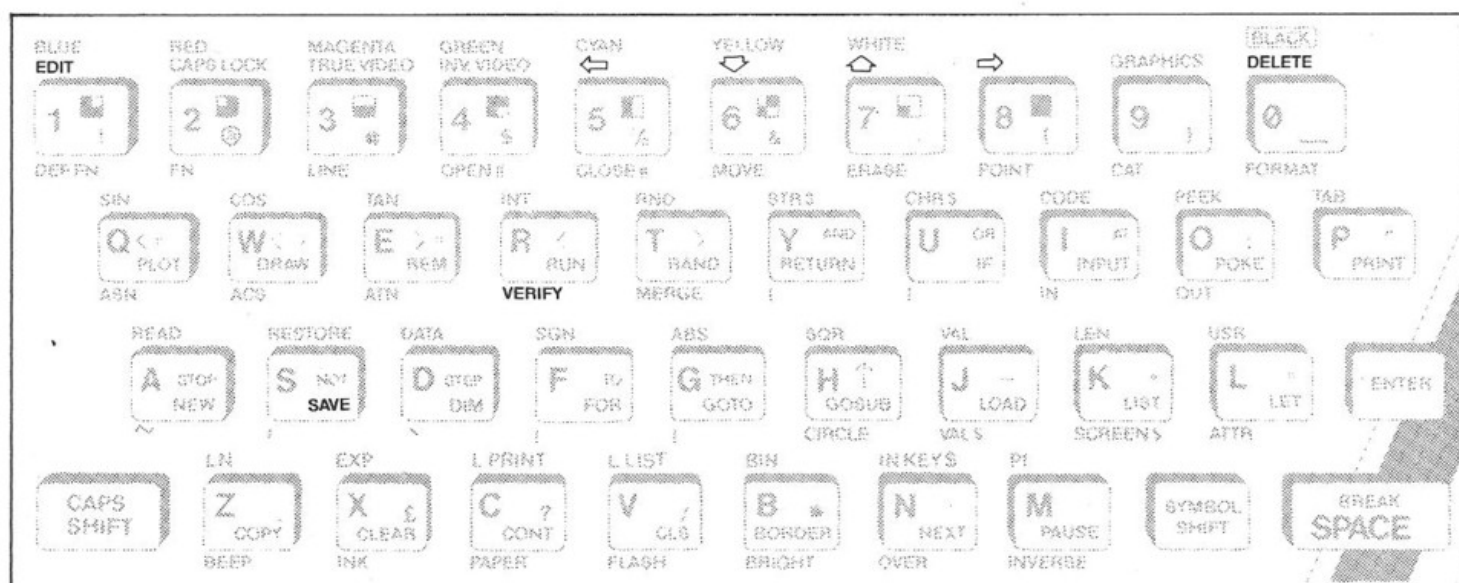
The Graphics mode

Summary

11.1 The Keyboard

This is a 'reference' chapter which you should read and understand. You don't have to learn the subject matter by heart, and there are no exercises, but you will probably need to refer back to the chapter from time to time as you work through the rest of the course.

The keyboard is one of the most complex features of the SPECTRUM. So far you've just been using it in a straightforward way, typing a keyword at the beginning of a command and then relying almost entirely on the words and symbols printed on the keys themselves. The exceptions were:



In each of these cases we have described a more or less complicated sequence to get the effect of the symbol you have chosen.

The problem is that on the one hand SPECTRUM BASIC contains over 200 different signs and words. This does not include the numerous 'operations' which have to be provided to help you enter your program, like deletion, editing and cursor movement. On the other hand, the machine only has 40 keys on its keyboard. Clearly, if every one of the signs, words and operations is to be represented by one key-stroke, each key must be made to serve at least five or six different purposes.

This chapter will help you understand how the SPECTRUM keyboard really works, so that you can use it with confidence. We will not at this stage explain all the various operations and symbols: that will come later. For the present, it will be enough if you can find and call up any symbol or operation without further explanation on our part.

Perhaps you'll find the keyboard like a strange city. At first all parts of it look alike and you despair of ever finding your way around; but after a while everything falls into place, and you find yourself going wherever you want to go without even thinking about it.

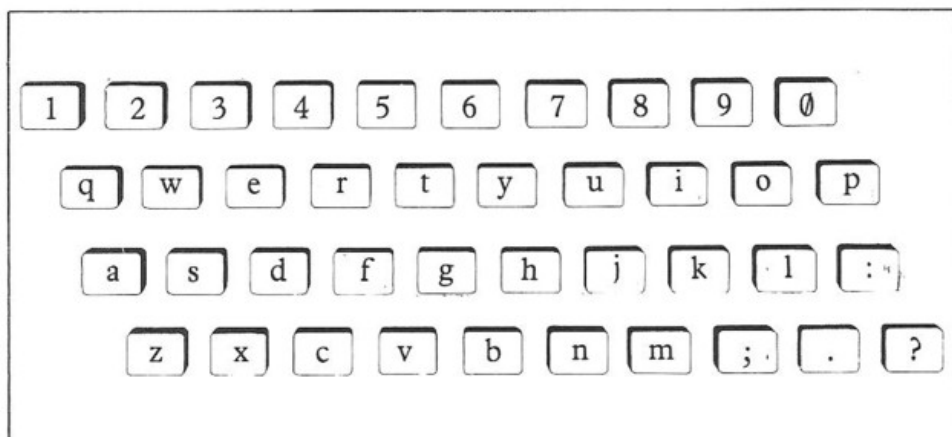
Although the keyboard is complicated, it is quite logically designed. It depends on the idea of 'keyboard shifts' taken to a degree much further than any of us are used to.

To give a full and accurate description of the SPECTRUM keyboard, we'll give you a 'trial run' based on something which nearly everybody must have used - or played with - at one time: a typewriter.

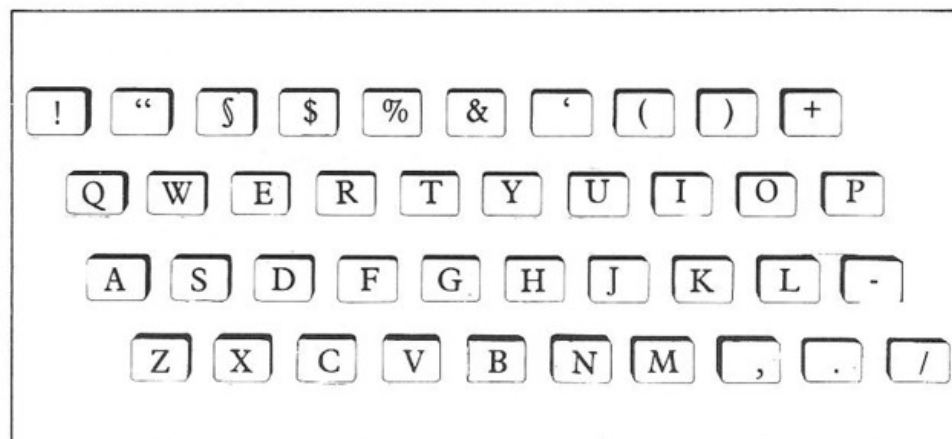
carries two different signs: a capital letter and its lower case equivalent or a number and a symbol such as #, £ or +.

Normally the keys are set up to use the lower case letters and numbers, but if you press the SHIFT key then all the type bars move so that capitals and symbols are used instead.

We can illustrate this point by drawing a 'lower case keyboard':



and an 'upper case keyboard':



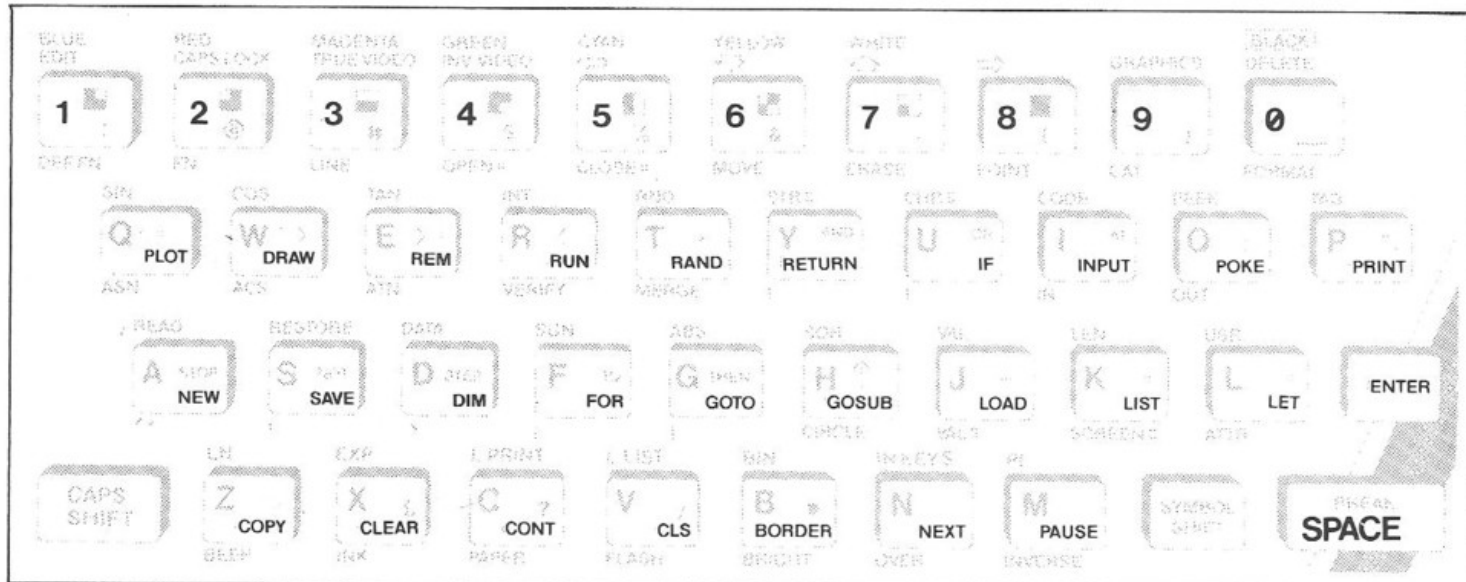
You can imagine (if you like) that the machine has two keyboards and the typist is free to switch between them. In practice it is easier to use only one keyboard where each key is used for two different signs.

It is worth looking at the mechanism by which you can change shifts. If the typewriter is in 'lower case' shift, you can move to upper case either by pressing (and holding) the SHIFT key, or by hitting (and releasing) the SHIFT LOCK. Similarly, you can move from upper to lower case either by releasing SHIFT or by striking SHIFT LOCK. The SHIFT and SHIFT LOCK are really 'operation' keys since they don't result directly in any typewritten characters.

The SPECTRUM keyboard uses no fewer than eight different shifts or 'modes'. You are already familiar with some of them.

11.2 The Keyword Mode

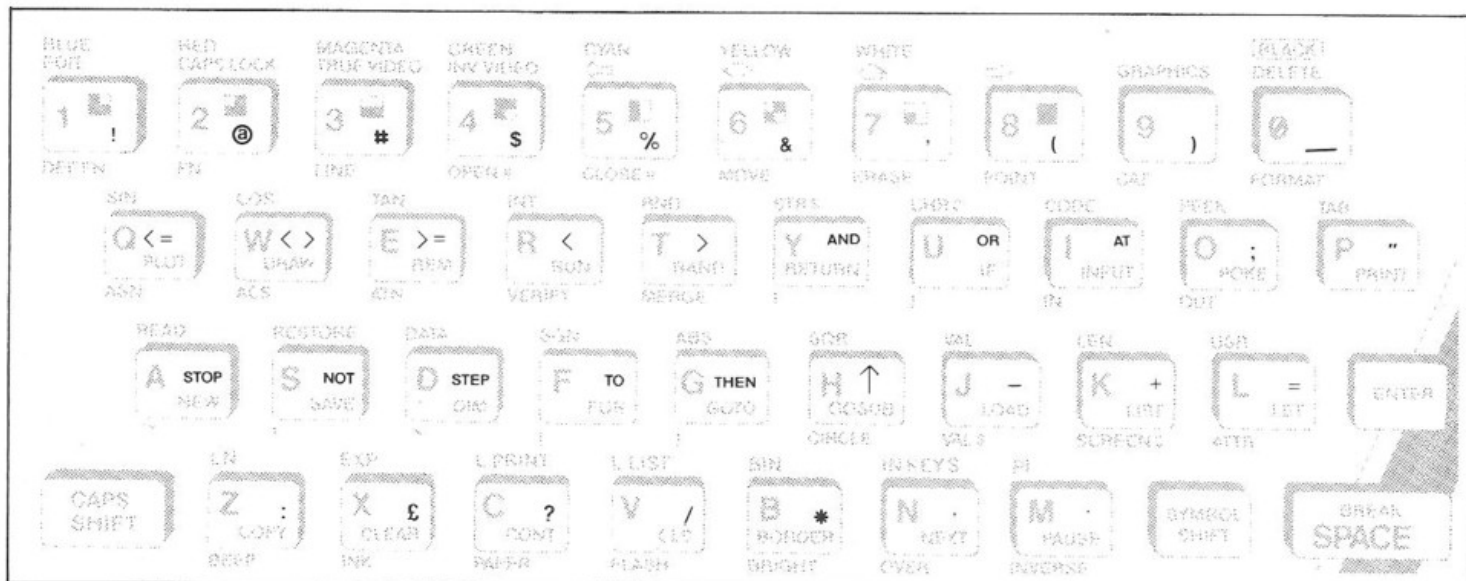
The KEYWORD mode is the first one you meet. It comes up at the beginning of a command (or after THEN or : if these symbols are not inside strings). The Keyword mode is indicated by a flashing K under the cursor, and its keyboard is:



We haven't shown the CAPS SHIFT and SYMBOL SHIFT keys because they are used to change shift. If you hold down CAPS SHIFT when the machine is in KEYWORD mode the top row of keys will give you 'operations' (such as DELETE or cursor movement) instead of numbers. The effect of the other keys remains unchanged.

11.3 The Symbol Shift Mode

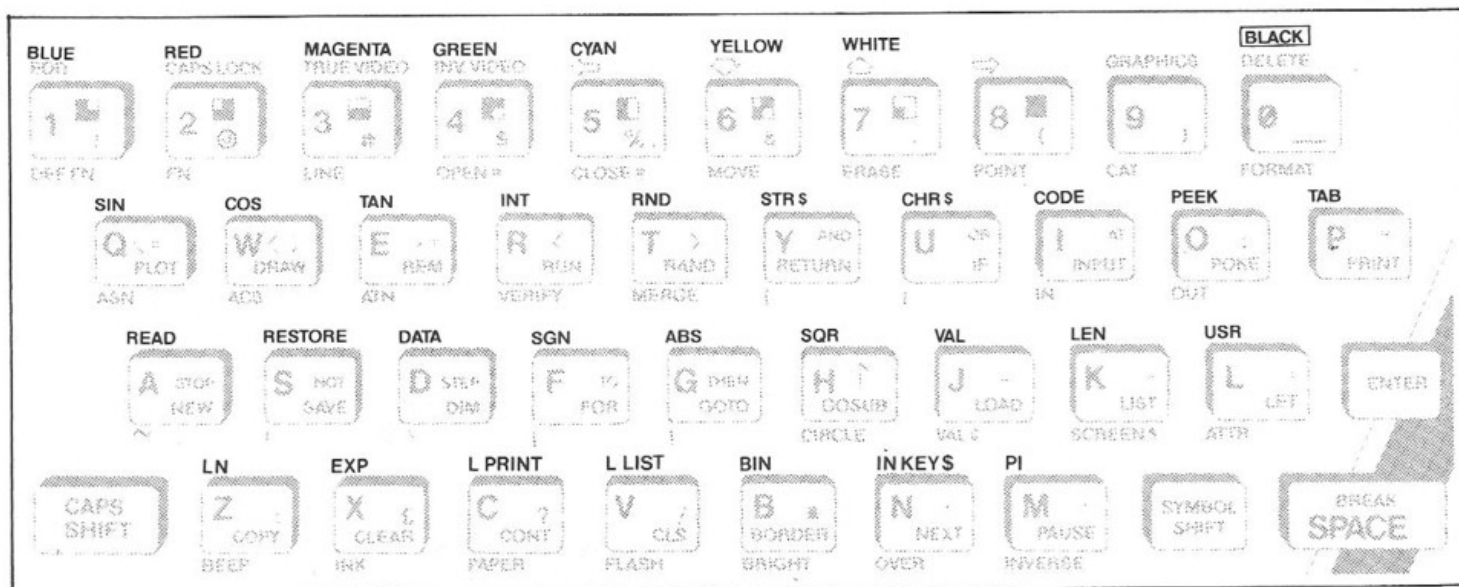
Now suppose that you press (and hold down) the SYMBOL SHIFT key. This moves the SPECTRUM to the SYMBOL SHIFT mode, where the keyboard consists of all the red characters and words printed on the keys themselves. The keyboard effectively is:



11.4 The Extended Mode

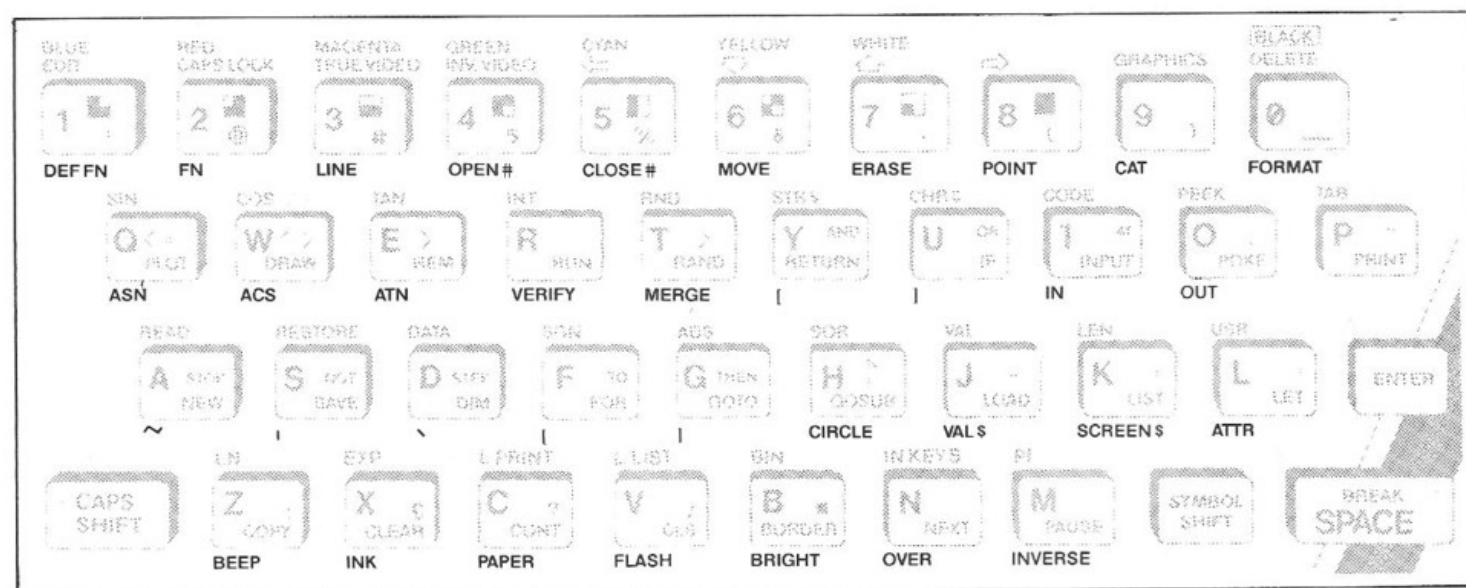
Several times in previous chapters, we've asked you to press **SYMBOL SHIFT** and **CAPS SHIFT** together, so that the cursor changes to a flashing E. This signals that the SPECTRUM keyboard is in the **EXTENDED** mode, and the symbols you get are those written above the keys, in green or other colours (except for the word **WHITE**).

The **EXTENDED** keyboard is:



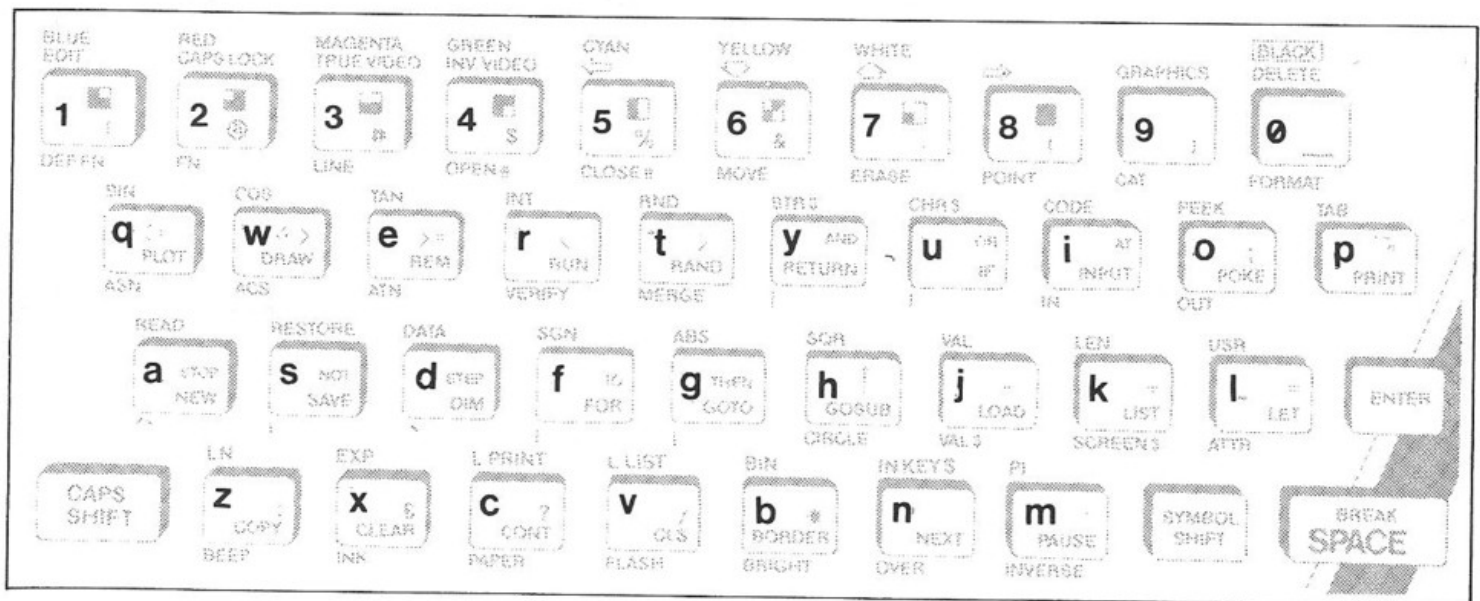
11.5 Extended (Symbol Shift) Mode

The last mode you can get to directly from **KEYWORD** is called **EXTENDED SYMBOL SHIFT**. You move into **SYMBOL SHIFT** by pressing the **CAPS SHIFT** and **SYMBOL SHIFT** keys together, and then holding down the **SYMBOL SHIFT** key. Now the SPECTRUM will give you the words or symbols printed in red below each key, as follows:



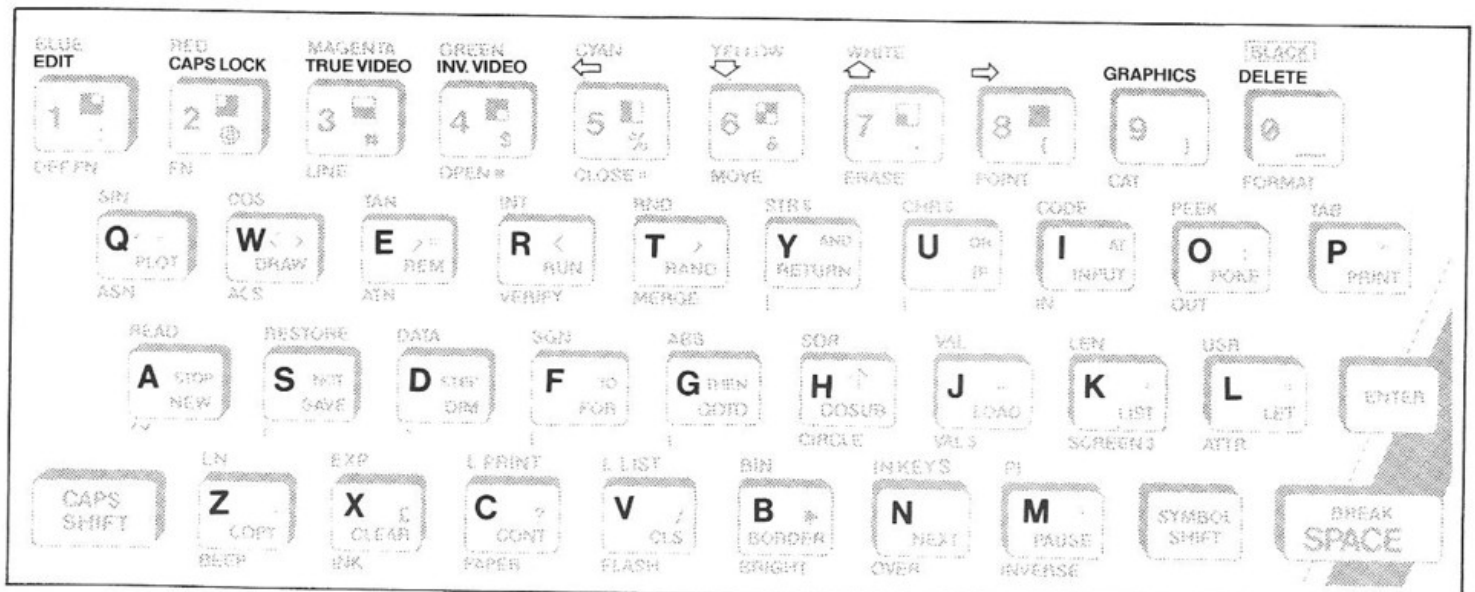
11.6 Lower Case Mode

So far, our discussion has been related to the **KEYWORD** mode, which comes up automatically as you start a command. The **SPECTRUM** will stay in this state as long as you keep typing digits, because it assumes that they must be part of the line number of a numbered command. It also remains in the **KEYWORD** state if you move the program cursor or bring a line down to edit. If you actually type a keyword, the machine normally changes to **LOWER CASE** mode, and shows a flashing **L** accordingly. The **LOWER CASE** mode is the one you use for most of your typing, and its keyboard is very like the one on a typewriter:



11.7 Caps Shift Mode

If you press and hold down the **CAPS SHIFT** key when the **SPECTRUM** is in **LOWER CASE** mode, it will change to **CAPS SHIFT** mode. Here the letters will come up as capitals, and the top row of keys will call up various operations. The corresponding keyboard is:



When you're in the **LOWER CASE** mode, you can call up the **EXTENDED** mode and the **EXTENDED (SYMBOL SHIFT)** mode by using the **CAPS SHIFT** and **SYMBOL SHIFT** keys. You'll be glad to hear that they are exactly the same as they were when you invoked them from the **KEYWORD** mode. The only point worth

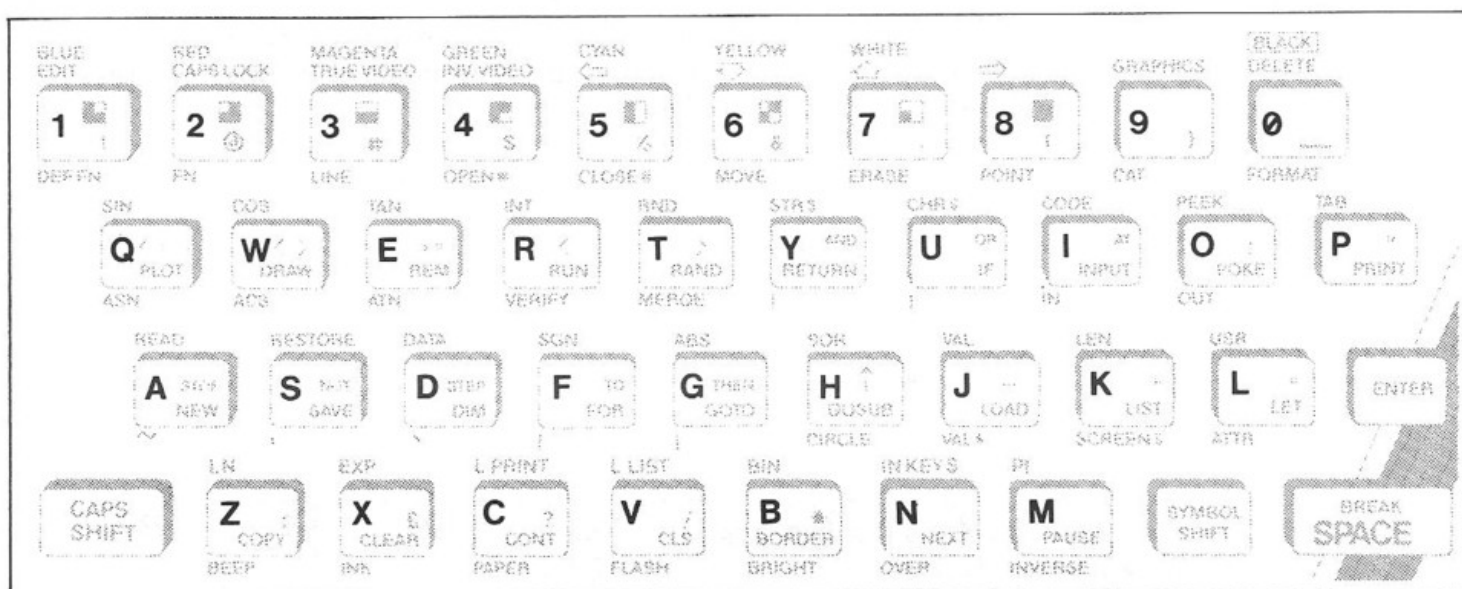
11.8 Caps Lock Mode

An alternative to the LOWER CASE mode is CAPS LOCK, which is signalled by a flashing C. In this mode the numbers are the same, but the letters appear as capitals. The mode is useful if you want everything you type to come out in capitals, and the text you are typing is too long to hold down CAPS SHIFT all the time.

The SPECTRUM switches between the LOWER CASE and CAPS LOCK modes when you hit the CAPS LOCK key (CAPS SHIFT and 2). It remembers which of the two modes it was in, so that after being in - say - K or E modes it returns automatically to the one it was in before.

If you hold down CAPS SHIFT when you are in the CAPS LOCK mode, you'll get operations instead of digits from the top row of keys.

The CAPS LOCK keyboard is:



11.9 Graphics Mode

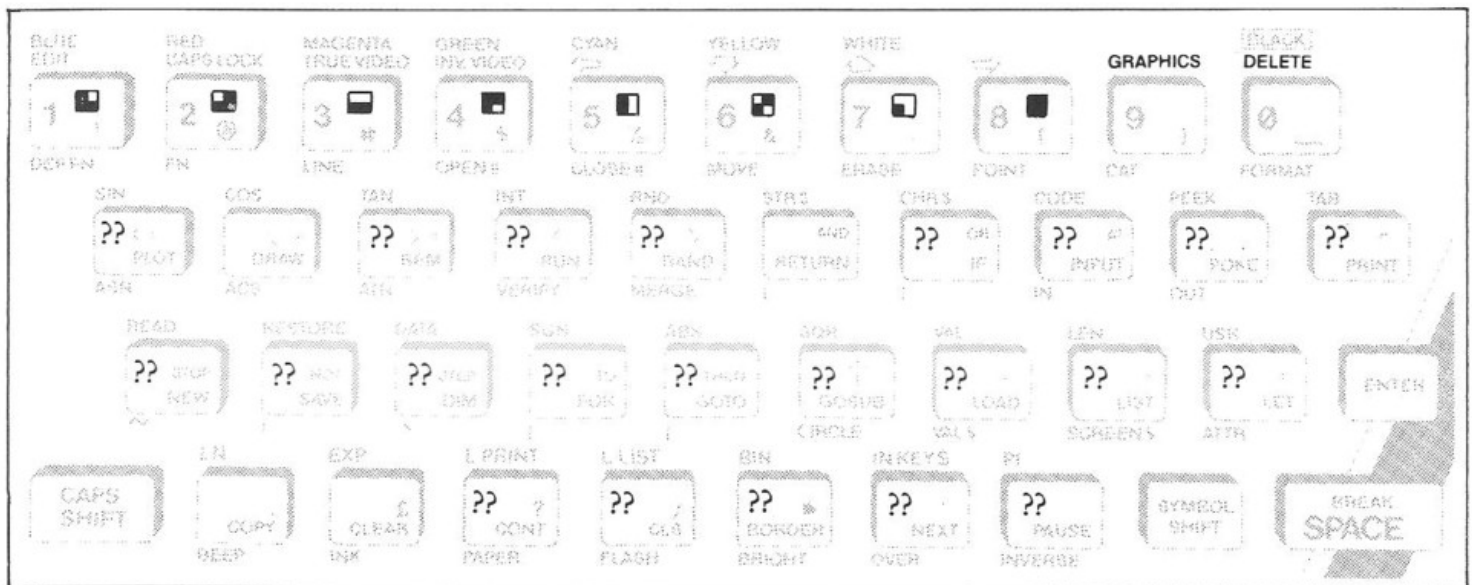
The last of the eight modes of the SPECTRUM is reserved for 'graphics' symbols. A graphic symbol is a sign, such as a cross, a star or a blob, used in drawing diagrams or pictures on the screen. There are eight 'standard' graphics symbols, which are shown on keys 1 to 8, but the programmer is free to design up to 21 symbols of his own and incorporate them in his program. They are called 'user-defined symbols'.

So far this course has not dealt with the graphics symbols, but we shall be considering them in a later chapter.

When the SPECTRUM is in the GRAPHICS mode, most of the keys produce graphics instead of their usual characters and words. Keys 1 to 8 display the standard graphics, whilst the letter keys (except V, W, X, Y and Z) correspond to the user-defined graphic symbols.

You can get into the GRAPHICS mode by holding down CAPS SHIFT and pressing GRAPHICS. Once you are in the graphics mode, you can get back to the previous mode by hitting key 9 (with or without CAPS SHIFT).

The GRAPHICS keyboard is shown below. ?? stands for a user-defined graphic symbol.



If you hold down the CAPS SHIFT key when in GRAPHICS mode, the number keys will produce reversed versions of the standard graphics symbols.

If you've got this far, congratulations! You won't need to spend any effort learning the various keyboards by heart, as the information will simply soak in as you gain practice in programming the SPECTRUM. There follows a short summary of the various modes, and in **Appendix D** you'll find a guide to all the various signs and symbols used by the SPECTRUM.

11.10 Summary

The KEYWORD mode is distinguished by a flashing K. It is entered automatically whenever the SPECTRUM expects a new instruction, and is used to type label numbers and keywords of commands. The mode changes to C or L as soon as you actually type a keyword.

The SYMBOL SHIFT mode is used whenever you hold down the red SYMBOL SHIFT key (unless the machine is already in EXTENDED MODE). This mode gives you all the red words and symbols printed on the keys themselves.

The EXTENDED mode is called up whenever you press CAPS SHIFT and SYMBOL SHIFT together. It gives you the symbol printed above each key, and only lasts for one key depression unless you call it up again. The EXTENDED mode is marked by a flashing E.

The EXTENDED SYMBOL SHIFT mode comes about whenever the SPECTRUM is in EXTENDED mode and you hold down the SYMBOL SHIFT key. This mode gets you the words printed below each key.

The LOWER CASE mode is signalled by a flashing C. It gives you the small letters and the decimal digits.

The CAPS SHIFT mode is used if you hold down CAPS SHIFT in LOWER CASE mode. It gives you the capital letters and the operations on the top row of keys.

CAPS LOCK mode is signalled by a flashing C, and is used to type capital letters and digits. You switch between LOWER CASE and CAPS SHIFT modes by holding down CAPS SHIFT and striking key 2 ('CAPS LOCK').

The GRAPHICS mode is used for getting the graphics characters. It is indicated by a flashing G, and you use key 9 (usually with CAPS SHIFT) to switch between this mode and the other ones.

Chapter 12

REM statements
DATA statements and
READ commands

In all the programs we have studied so far, every line has commanded the SPECTRUM to take some particular action: to PRINT something, to LET a variable take on a new value, or to GO TO a command out of the normal sequence. In this chapter we introduce two new types of line, which start with the keywords REM and DATA. These lines are very different from one another, and the only thing they have in common is that when the computer comes to one of them in the normal sequence of obeying commands, it skips right over without taking any action at all. We therefore talk about REM and DATA **statements** rather than referring to them as 'commands'.

12.1 REM Statements

Programs are designed to be understood by computers. People generally find them obscure and confusing, even though every correct BASIC program is perfectly clear and meaningful to the SPECTRUM itself. One of the most difficult tasks in programming is to take a working program and to modify it, so that it does a job more suited to your special needs. You usually have to deduce, merely from the clues in the program itself, exactly what the original programmer meant by variables with names such as s\$, min or j.

The main point of a REM statement is to be a 'signpost' for any human reader of your program. The REM statement is numbered like any other line, and starts with the keyword REM. Then comes any text you wish to include, such as descriptions of the variables, or of the methods used to work out your results. Here is a typical example:

```
...  
40 PRINT "Welcome to the Monster Quiz"  
50 REM n$ is the pupil's name, and s the score so far  
60 INPUT "What is your name";n$  
70 LET s=0  
...
```

When it obeys this section of code, the machine carries out the PRINT command in line 40, just as normal. It then notes that 50 is a REM statement and skips directly to line 60.

The longer and more complex the program, the more important the REM statements become. You should use a REM statement to describe each section of the program, stating its purpose and the variables it uses. Eventually your programs will reach a size where you can't remember every detail in your head, and the REMs and other design notes you keep become essential; without them you simply can't carry on without getting more and more mixed up.

When you start work on a new program, the first statement should be a REM with a 'title'. The statement should include the purpose of the program and your name, thus:

```
10 REM Football quiz by Q.Swarbrick
```

If you intend to sell your program, we advise you to include a copyright notice like this:

```
20 REM COPYRIGHT © QUENTIN SWARBRICK 1984
```

applies in all countries whose governments have signed the Berne Copyright Convention - that is, most countries of the World.

The SPECTRUM obligingly gives you a copyright symbol. It is under P in EXTENDED SYMBOL SHIFT mode. (See section 11.4).

Although REM statements are completely ignored by the computer, it is possible to GO TO a REM line. The effect is the same as a jump to the next line which is not a REM (or a DATA statement). For example:

```
...
100 REM A comment
110 REM Another comment
120 PRINT "Try Again"
...
160 GO TO 100
...
```

In this sequence,

GO TO 100

has exactly the same effect as

GO TO 120

Short REM statements don't need line numbers of their own; you can tack them on to the ends of other commands and statements using a colon : as glue.

```
50 LET x$ = "summer":REM x$ is the season
```

Finally, be generous with REM statements; one day you'll be glad of them !

12.2 DATA Statements and READ Commands

Some programs need lots of background information to run - timetables, lists of places or books, or sequences of musical notes forming tunes. The DATA statement gives you an easy way to include this information in your programs.

Here is a timetable for trains from Glasgow to Oban (correct for August 1983):

Glasgow Queen St (dep)	Oban (Arr)
0804	1116
1255	1600
1823	2130

Suppose you've been asked to write a program which gives a traveller information about this journey. It asks what time he would like to leave, and tells him the details of the next available train (if there is

When can you leave Glasgow?

0800 ←

User's reply,
copied up from
INPUT area

Your best train leaves Glasgow at 0804,
arriving at Oban 1116.

or:

When can you leave Glasgow?

1930 ←

User's reply

No more trains today

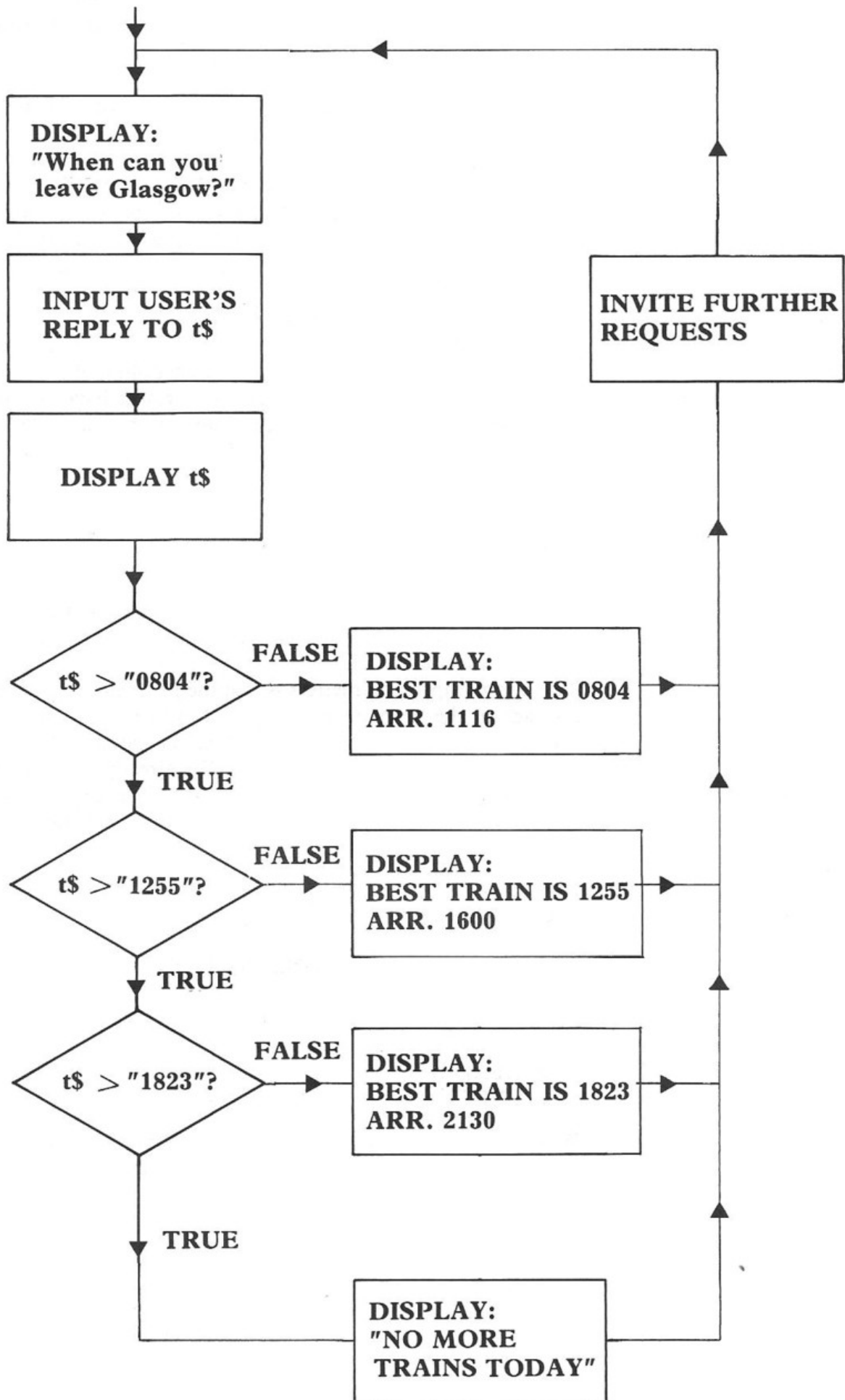
With our existing skills, the program is quite straightforward, even if rather long-winded:

```

10 REM Glasgow-Oban Travel (A.Colin)
20 PRINT "When can you leave Glasgow?"
30 INPUT t$ : REM t$ is time passenger wants to leave
40 PRINT
50 PRINT t$ :REM Copy time to top of screen
60 PRINT
70 IF t$ > "0804" THEN GO TO 110
80 PRINT "Your best train is at 0804"
90 PRINT "Arriving at 1116."
100 GO TO 200
110 IF t$ > "1255" THEN GO TO 150
120 PRINT "Your best train is at 1255"
130 PRINT "Arriving at 1600."
140 GO TO 200
150 IF t$ > "1823" THEN GO TO 190
160 PRINT "Your best train is at 1823"
170 PRINT "Arriving at 2130."
180 GO TO 200
190 PRINT "No more trains today."
200 INPUT "More information?" ;x$
210 CLS
220 GO TO 10
    
```

A flow chart for this program appears in Fig 12.1

(Fig 12.1)



If you key this program into the SPECTRUM, you'll find that it works correctly. We have used strings to represent the times so that quantities like "0804" are properly displayed. Conditions like the one in line 70 still work, because if two strings represent the time of day on the 24-hour system, the relation $>$ means 'after'.

If we had used numerical variables, times before 10 o'clock would have had zeros missing from the front!

The point of the commands at 200-220 is to let the user make another enquiry. The INPUT line at 200 simply makes the computer wait until the user types something - anything will do. Line 210 clears the screen and 220 returns to the beginning of the program.

The program seems somewhat repetitious, but its defects don't really strike you until you try to adapt it to different circumstances.

The simplest change possible is the rescheduling of a train. If the first train is timed to run 2 minutes later, you'll have to change three statements in your program - 70, 80 and 90, because "0804" becomes "0806" and "1116" becomes "1118".

Now suppose that British Rail put on some extra trains in the evening. Each new train will need four lines of program. Soon you'll run out of line numbers between 180 and 190 and start having to change the numbering in other parts of the program.

If you try to adapt the program for a really busy route (say with 30 trains a day) the work soon becomes intolerably tedious - more like a school punishment than a creative effort.

The key to improving matters is that the program does the same action for each of the trains in the timetable: it uses a condition ($t\$ > \text{train time}$) to work out whether the traveller can catch that train, and if so, it gives the appropriate instructions. We already know how to control repetition with loops, but in this case the difficulty is that two of the variables in the loop - the departure and arrival times of the trains - vary without any regular pattern.

DATA statements and the READ command are designed just to solve this problem. To see how, examine this simple program:

```
10 REM DATA DEMONSTRATION
20 DATA 7
30 DATA 19
40 READ a
50 READ b
60 PRINT a,b
```

When this program is started, it begins by sorting the DATA statements out and putting them in a separate pile, thus:

```
10 REM ...      20 DATA 7
40 READ a       30 DATA 19
50 READ b
60 PRINT a,b
```

Now it starts obeying the commands which are left. The REM is skipped, so that the first 'real' command is:

```
40 READ a
```

next value is the 7 in statement 20, so this is the one to be used. Once a value has been read, the top DATA statement is thrown away (like a discard in a game of cards) so that the top 'visible' statement is the one labelled 30.

The next command is at line 50. It reads the 19 from statement 30, and puts it into variable b. Finally, the PRINT command displays:

7 19

The READ command is very like INPUT. It can also be used to set up values for any number of variables of either type - numbers or strings. The main difference is that the READ command fetches its values from DATA statements rather than asking the user to type anything. It therefore doesn't need prompts and runs at the full speed of the machine.

A DATA statement can contain any number of values, separated by commas. They are peeled off as they are needed by READ commands, and a DATA statement is not discarded until all the values in it have been read. The values can be numbers or strings, but the strings have to be enclosed in double quote marks, like this:

```
200 DATA "Mozart", "Haydn", "Schubert"
```

Experiment 12a

To see how well you have understood DATA statements, study the following program and try to predict the display it produces:

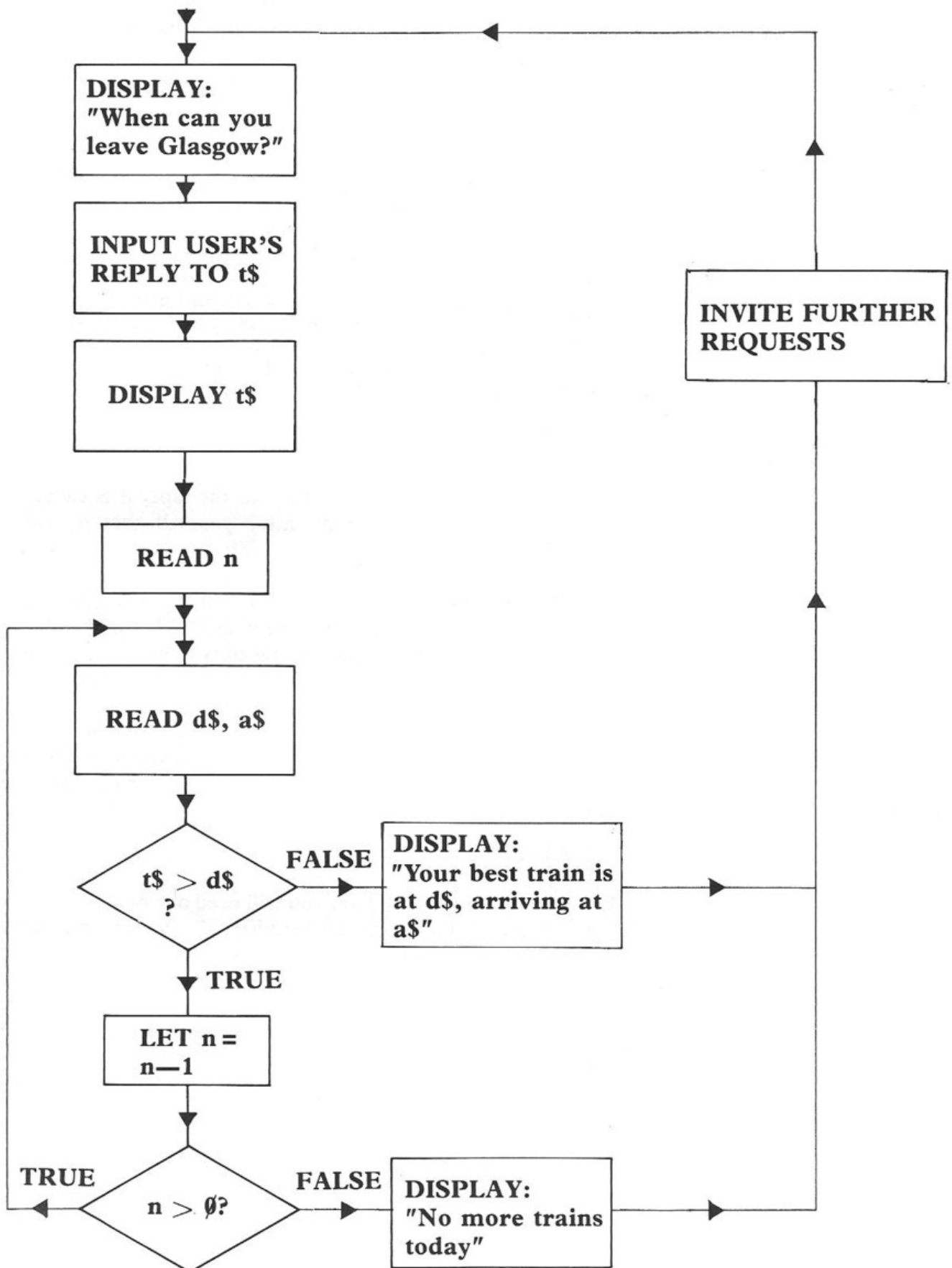
```
10 REM Musical History
20 READ n: REM n is number of composers
30 FOR j = 1 TO n
40 READ c$,b,d
50 PRINT c$;" lived for ";d-b;" years"
60 NEXT j
70 STOP
100 DATA 4: REM Number of composers
110 DATA "Mozart",1756,1791
120 DATA "Bach",1685,1750
130 DATA "Richard Strauss",1864,1949
140 DATA "Sibelius",1865,1957
```

(Clue: the first time round the loop, n,c\$,b and d will be set to 4, "Mozart",1756 and 1791 respectively).

Experiment 12a completed.	
---------------------------	--

Now we can return to the timetable program. Here is a version which uses DATA statements and READ commands:

(Fig 12.2)



```
10 REM Glasgow-Oban Travel
20 PRINT "When can you leave Glasgow?"
30 INPUT t$: REM t$ is time passenger wants to leave
40 PRINT
50 PRINT t$
60 PRINT
70 READ n : REM n is number of trains
80 READ d$,a$ : REM d$ and a$ are departure and arrival
  times
90 IF t$ > d$ THEN GO TO 130
100 PRINT "Your best train is at ";d$
110 PRINT "Arriving at ";a$
120 GO TO 160
130 LET n = n-1
140 IF n > 0 THEN GO TO 80
150 PRINT "No more trains today."
160 INPUT "More information ?";x$
170 CLS
180 RESTORE
190 GO TO 10
200 DATA 3 : REM number of trains
210 DATA "0804","1116"
220 DATA "1255","1600 "
230 DATA "1823","2130 "
```

You'll find a copy of this program on the tape; it is called "trains". Load it and try it out, and satisfy yourself that it really works correctly.

As before, the commands from 160 to 190 are concerned with letting the user make another enquiry. The RESTORE command collects up the discarded DATA statements and puts them in order, so that the next READ command can start at the beginning again.

Study the program carefully, and find out for yourself how it works. Its advantages really begin to show up when you start changing it in various ways. If a train is rescheduled, you only have to change one DATA statement; for example:

```
210 DATA "0806","1118"
```

If any extra trains are laid on, you will need one new DATA statement for each one, and of course you will have to alter line 200 which gives the number of trains.

Experiment 12b

As a practical exercise, we'd like you to edit the "trains" program in two ways: First, modify it to allow for the following extra trains:

1015 (dep)	1515 (arr)
1710 (dep)	2039 (arr)
1937 (dep)	2346 (arr)

Remember to put the trains in the right order! You'll find the READ, DATA and RESTORE keywords in EXTENDED mode.

Second, change the program so that it isn't limited to the Glasgow-Oban line, but can work with any origin and destination. You'll have to include the actual origin and destination as strings in a DATA statement, and to use variables (perhaps o\$ and p\$) to represent them in your program. Try out the program on the following timetable:

Florence (Dep)	Perugia (Arr)
(Correct until Sept 23rd. 1983)	
0311	0609
0436	0704
0531	0807
0700	0928
0802	1040
0948	1212
1140	1408
1330	1614
1454	1742
1643	1914
1728	2029
1850	2117
1950	2205
2042	2336

Experiment 12b completed.	
---------------------------	--

To end this section we'll just repeat a few of the important points about DATA statements and READ commands:

A. DATA statements are separated out whenever a program is started. They can be placed anywhere in the program: the beginning, the middle, or the end, or they can be scattered at random. It is best to keep them together so that you can find them.

B. DATA statements can hold numbers or strings. The strings must be enclosed in double quotes. If a DATA statement contains two or more values, they must be separated by commas.

C. DATA statements are used up in the order of their line numbers. The RESTORE command takes the machine 'back' to the first DATA statement.

D. A READ command can read any number of values from DATA statements. There is no need for all the values read by a READ command to be in a single DATA statement, nor for a READ command to 'use up' all the values in any one DATA statement. In short, the SPECTRUM behaves as if it had one very long DATA statement instead of many shorter ones.

E. The type of a variable mentioned in a READ command must correspond to the type in the DATA statement. For example:

READ a

expects a number and would make the SPECTRUM give a fault warning if the next value was a string. The same would be true the other way round.

F. If a READ command is given when all the DATA statements have been used up (and not RESTORED) the SPECTRUM will report a fault:

Out of DATA

and the line number of the offending command.

Experiment 12c

The shopping list program you wrote in Chapter 8 is to be generalised so that it becomes more widely useful. It is now to include a number of DATA statements, each one giving the name of a grocery item and its price in pence per pound. The first DATA line just gives the number of grocery items included.

A sample set of data is:

```
100 DATA 5: REM 5 items
110 DATA "Haggis",102
120 DATA "Caviare (Beluga)",6500
130 DATA "Quails' Eggs",2550
140 DATA "Salsify",40
150 DATA "Truffles",1540
```

Write a program which will ask the user how many pounds they would like to buy of each type of food, and then displays the total bill.

<i>Experiment 12c completed.</i>	
----------------------------------	--

Chapter 13

Beeps

13.1 Beeps : How to Play Music

To introduce this section, load and run program "tune".

Whilst the music making capacities of the SPECTRUM are modest, they can be used to add interest to programs whose impact would otherwise be purely visual, and to give the user warnings and information where necessary.

Sound is produced by the aptly-named BEEP command. You'll find the keyword for this command in EXTENDED SYMBOL SHIFT mode, on key Z.

The BEEP command plays a single note on the loudspeaker built into the SPECTRUM. The keyword is followed by two numbers or expressions, which control the length and pitch of the note respectively. For example:

BEEP 2, 0

will play a long, low note, whilst

BEEP 0.1, 25

will produce a short, squeaky one. Try both these commands on the SPECTRUM.

To fill in some detail, the time a BEEP takes is exactly the same, in seconds, as the value of the length number. For example,

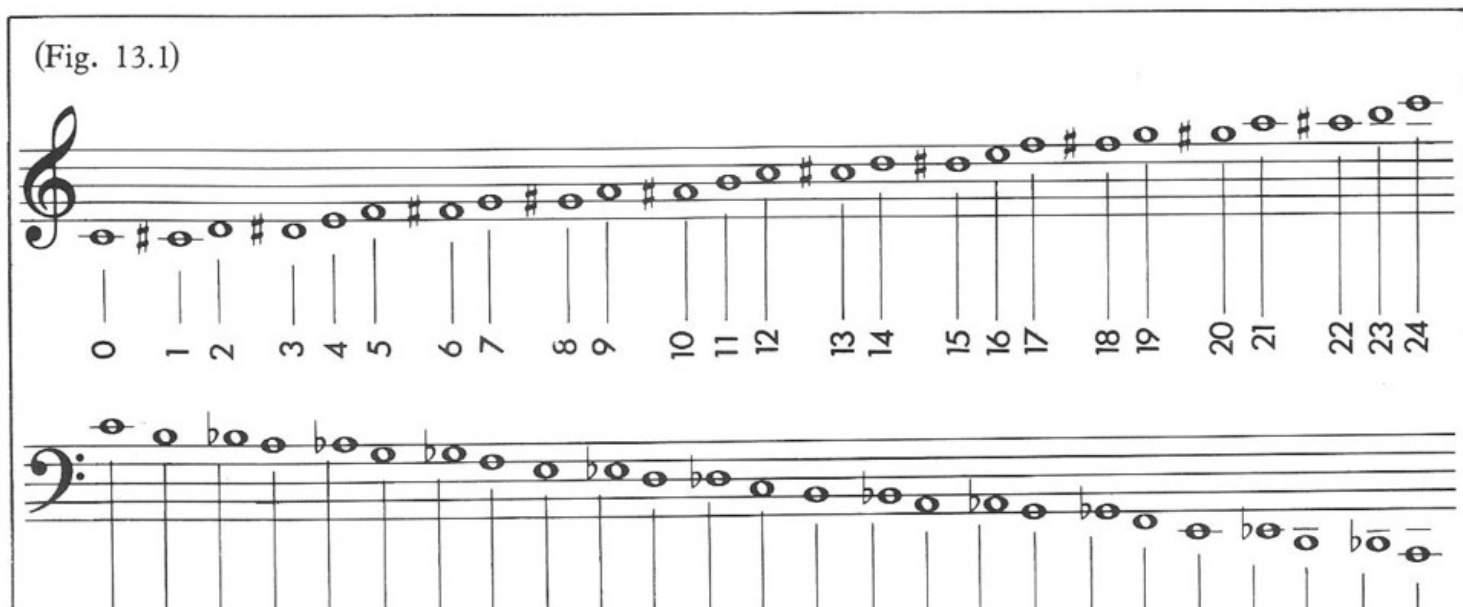
BEEP 5, 12

will last exactly five seconds.

The number which controls the pitch of the BEEP is related to the keyboard of a piano. 0 gives 'Middle C', and every step of one unit up raises the pitch by a semitone. A pitch number of 12 gives the C exactly one octave above Middle C.

The notes below Middle C are obtained in the same way, using negative numbers.

The SPECTRUM will play a note of any pitch between -60 and +69. Unfortunately the notes at the extremes of this range do not sound musical, and you are recommended to keep to the notes shown in this diagram :



You can use the BEEP command to make the computer give different kinds of sound, depending on whether the user's action is sensible or not. For example:

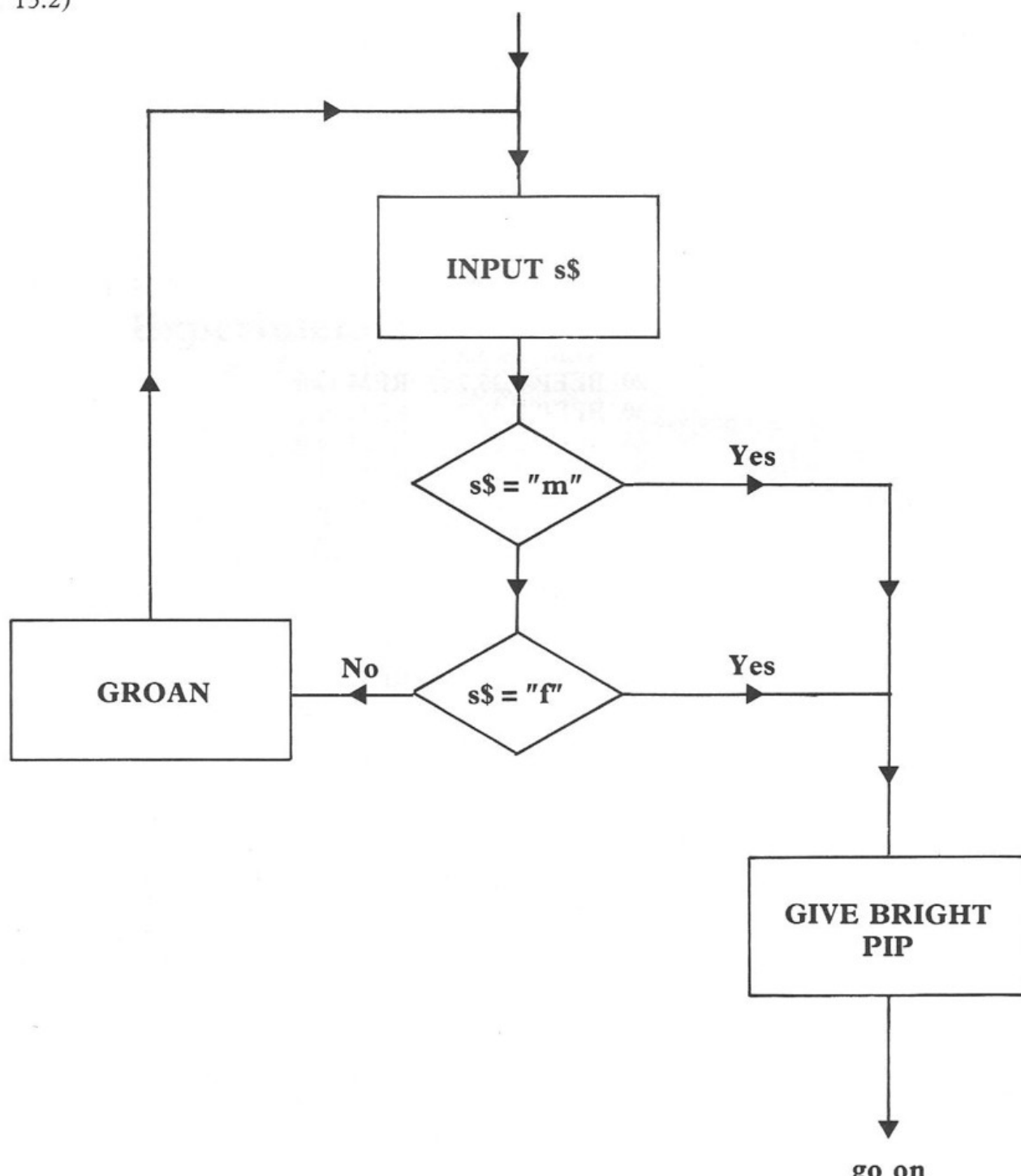
```

...
100 INPUT "Are you male (m) or female (f) ?";s$
110 IF s$="m" THEN GO TO 160
120 IF s$="f" THEN GO TO 160
130 BEEP 3,-10:REM groan
140 PRINT "Answer m or f!"
150 GO TO 100
160 BEEP 0.2,9:REM bright pip
...

```

The flow chart below may help you follow this fragment of code.

(Fig. 13.2)



To play a tune, all you need do is to arrange for the machine to sound the right sequence of notes. Each note in the tune must be 'coded' into a length and a pitch, and then included in a BEEP command. This is laborious, but quite simple. Consider the first 8 bars of 'Twinkle twinkle little Star' (which Mozart thought good enough to write a set of variations on).

(Fig. 13.3)



We shall make each quaver in the tune last 0.25 seconds. The first note is a G, which (according to the diagram) has a pitch number of 7. The right command to play it is therefore:

BEEP 0.25,7

We can work through the tune in this way, coding each note. We end up with:

```

10 REM Twinkle
20 BEEP 0.25,7 : REM twin-
30 BEEP 0.25,7 : REM kle
40 BEEP 0.25,14: REM twin-
50 BEEP 0.25,14: REM kle
60 BEEP 0.25,16: REM lit-
70 BEEP 0.25,16: REM tle
80 BEEP 0.5,14 : REM star
90 BEEP 0.25,12: REM how
100 BEEP 0.25,12: REM I
110 BEEP 0.25,11: REM won-
120 BEEP 0.25,11: REM der
130 BEEP 0.25,9 : REM what
140 BEEP 0.25,9 : REM you
150 BEEP 0.5,7 : REM are
  
```

This approach is hard work and gives a very rigid result. For example, if you wanted the tune to go faster you'd have to change the length number in every command, or if you preferred it in a different key, you would be obliged to alter all the pitch numbers.

Fortunately there is a good deal of regularity about both the length and the pitch numbers. For example, to make the tune go twice as fast, you would simply divide each length number by two. For each semitone that you wanted it played higher, you would add 1 to the pitch number.

In the example that follows, all the length and pitch numbers for the notes of the tune are supplied as DATA statements. To make matters easier, we have fixed the length of each quaver as 'one unit' instead of putting down its actual duration in seconds. If the tune was played

of its written value) and adding a fixed number to the pitch of each note. By changing the dividing factor, you can run the tune at any speed you like, and the same goes for the pitch. In the version of the program below, the duration is set in line 20 and the pitch in line 30:

```
10 REM Tune player
20 LET d = 4: REM set speed
30 LET p = 5: REM set pitch
40 READ du,pt
50 IF du = 0 THEN GO TO 80
60 BEEP du/d,p+pt
70 GO TO 40
100 REM Twinkle Twinkle Little Star
110 DATA 1,7,1,7,1,14,1,14,1,15,1,15,2,14
120 DATA 1,12,1,12,1,11,1,11,1,9,1,9,2,7
130 DATA 0,0
```

If you list program "tune" (which you loaded from the tape at the beginning of this chapter) you will see that it is controlled in the same way. Try altering its speed and key (by editing lines 20 and 30) and observe the effect.

Experiment 13a

Write a program to play your favourite tune, over and over, with the pitch rising a semitone each time.

<i>Experiment 13a completed.</i>	
----------------------------------	--

Chapter 14

The Print position
The Graphics symbols
User-defined graphics

The TV set connected to your SPECTRUM is a vital part of the whole system. Apart from a few BEEP's, all the information the computer needs to convey to you must be presented in the form of a television picture.

In this chapter we'll look at some of the ways you can use SPECTRUM BASIC commands to produce interesting and well-designed displays.

When you're reading through the chapter and working on the exercises, you'll find a supply of squared paper essential. We suggest you get some now.

14.1 The Print Position

When you are designing a program, one of the most important aspects is the way the results are arranged on the screen. For example:

```

After 10 years the
Pharaoh will have ← professional
used 328350 blocks.

is better than
328350 ← amateur
  
```

To produce such an output, you'll have to take a little trouble, such as writing:

```

...
90 CLS
100 PRINT "After ";n;" years the"
110 PRINT "Pharaoh will have"
120 PRINT "used "; s ; " blocks."
...
  
```

A good deal of screen control can be done with the commands you already know about:

CLS To clear the screen (and start printing at the top)

PRINT (by itself): to display a blank line

PRINT followed by one or more items:

The items will be displayed in the order they are given. If they are separated by commas, they will be arranged in two columns, but with semicolons the items will simply be run together without any space in between.

Useful as they are, these facilities are not powerful enough to do everything you might need. For example, consider the problem of displaying a form with blanks for the answers, and then filling them in, on the screen, as the user replies to the various questions. You might begin by showing something like this:

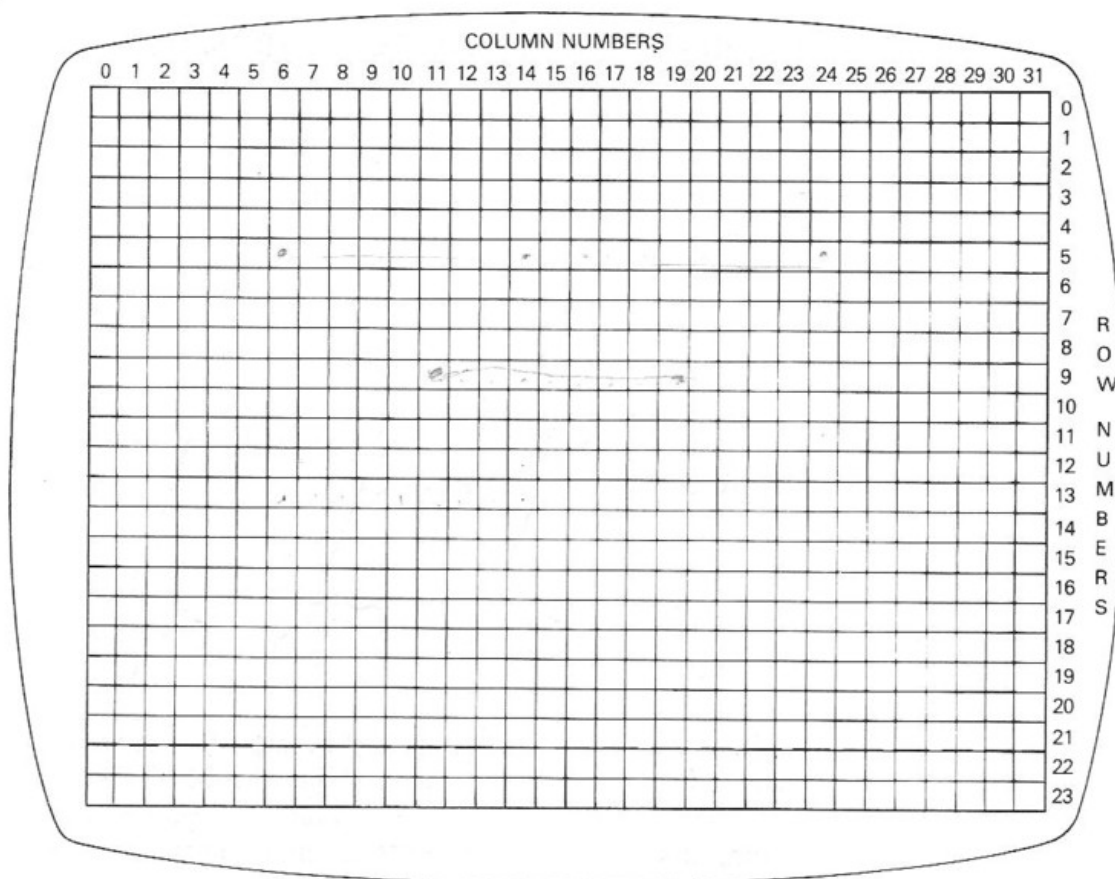
```

Name .....
Age .....
Type of Compute
Was it a gift?
How long hav
Has it bro
  
```

must appear on the screen in the right place. Clearly you can't use CLS to get back to the top of the screen because that would wipe out the form and all the other answers. It won't do just to PRINT the answers as they come, because they will appear in the wrong place. We need a way of telling the computer exactly where to make the PRINT command put its results.

To find its way round the screen, the SPECTRUM uses a simple 'navigational' system. Every position at which a character can be displayed is identified by its row number (so many rows down from the top) and its column number (so many in from the left edge). The screen map is like this:

(Fig 14.1)



When the SPECTRUM is running a BASIC program, it keeps track of the place where the next printed item is to go. This place is called the 'current print position', and the computer remembers both the row number and the column number concerned.

Most of the commands which put something on the screen affect the print position. For example:

```
PRINT "HELLO"
```

will advance the row number by one and reset the column number to zero (so that the next item will come at the beginning of the next line) whilst:

```
PRINT "HELLO";
```

simply advances the column number five places

A PRINT command can include certain special items which do nothing at all except change the current print position. The three most useful ones are:

TAB n; (where n is a number, a variable or a formula). This moves the print position on to column n.

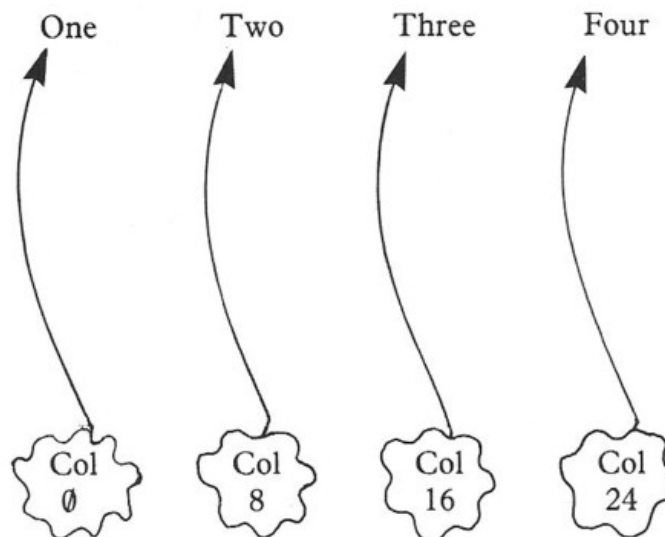
AT c,r; (where c and r are both numbers, variables or formulas). This takes the print position directly to column c and row r, no matter where it was before.

CHR\$8; This moves the print position back one space.

TAB is the simplest of these items to use. If you put

```
PRINT "One";TAB 8;"Two";TAB 16;"Three";TAB 24;"Four"
```

the machine will reply:



The word 'Two' is at column 8 because that is where the TAB item has just put the print position. Note that all the items in the PRINT command are separated by semicolons. If you had used commas the computer would have tried to arrange the items in two columns and would have ruined your intended layout.

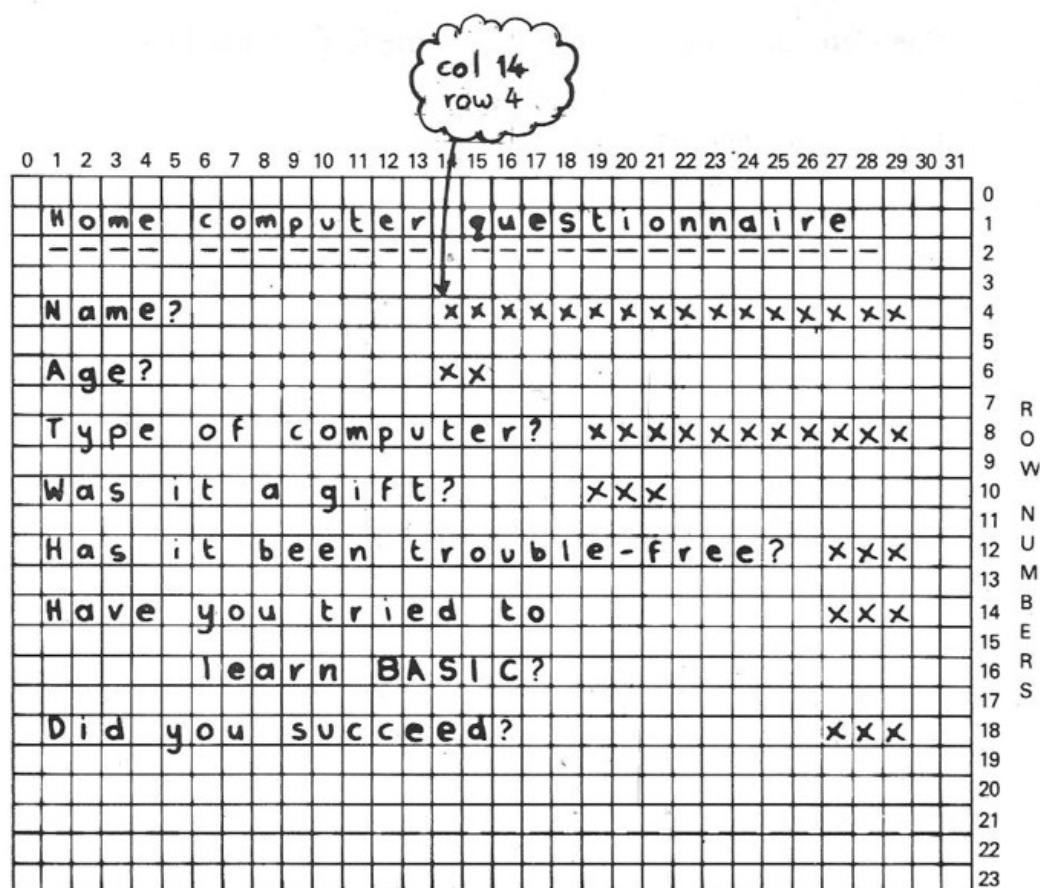
If you follow the TAB symbol with a variable or expression, you can get interesting patterns. Try the following:

```
10 FOR j= 0 TO 20
20 PRINT TAB j;"*"
30 NEXT j
40 STOP
```

and

```
10 FOR k=0 TO 15
20 PRINT TAB k; " + "; TAB (31-k);" + "
30 NEXT k
40 STOP
```

If you tell the SPECTRUM to TAB to a certain column, and the current print position is already past that column, the machine won't go



Use X's to show where the answers are going to appear. Once you've drawn the diagram, you can see exactly where each item is to go. In the example, the title goes in column 1 row 1, whilst the name of the person answering starts in column 14, row 4.

The program will use AT to put everything in the right place. The part which displays the blank form begins:

```
10 CLS
20 PRINT AT 1,1;"Home Computer Questionnaire"
30 PRINT AT 2,1;"  _____  "
...
```

and the part which inputs and displays the answers might go:

```
150 INPUT "What is your name? "; n$
160 PRINT AT 4,14;n$
170 INPUT "How old are you? "; age
180 PRINT AT 6,14; age
...
```

Experiment 14a

Complete this program, according to the form in Fig 14.2

<i>Experiment 14a completed.</i>	
----------------------------------	--

One of the most useful items in a PRINT command is CHR\$ 8, which, as we have said, simply moves the print position back one place. The significance is that you can rub out the last symbol you displayed, by moving back and printing a space instead:

```
PRINT CHR$ 8;"  ";
```

This feature is a great advantage if you need to show a moving object. Here, for instance, is a program which shows a falling star:

```
10 FOR f = 0 TO 21
20 PRINT AT f,10;"*";
30 FOR g = 0 TO 20
40 NEXT g
50 PRINT CHR$ 8;"  ";
```

```
60 NEXT f
70 STOP
```

Enter this program and run it. Don't miss out the semicolon in line 20. The basic idea is to display the star in row f (command 20), wait for a moment (commands 30 and 40) and then rub it out (command 50). The next time round the main loop, the star is displayed one row lower, until it reaches the bottom row.

As a more complex example, here's a program which shows a 'billiard ball' bouncing around the screen.

```
10 CLS
20 PRINT "  ";
30 LET x = 0
40 LET y = 0
50 LET dx = 1
60 LET dy = 1
70 LET x = x + dx
80 LET y = y + dy
90 IF x = 31 THEN LET dx = -1
100 IF x = 0 THEN LET dx = 1
110 IF y = 20 THEN LET dy = -1
120 IF y = 0 THEN LET dy = 1
130 PRINT CHR$ 8;"  ";;REM Rub out old ball
140 PRINT AT y,x;"O";;REM Display new ball
150 GO TO 70
```

Enter this program and run it. When you've seen what it does, study it carefully and decide how it works.

Here are some hints:

x and y are the present column and row positions of the ball. The column number must lie between 0 and 31, and the row number, between 0 and 20.

dx and dy show which way the ball is moving. When dx = 1 it's going to the right, and when dx is -1 it is moving left. Similarly, dy is 1 for 'down' and -1 for 'up'. dx and dy have to be taken together, so that if dx = 1 and dy = 1, the ball is moving diagonally downwards and to the right.

Commands 70 and 80 move the ball one square in the direction it is going. Commands 90 to 120 detect whether it has hit one of the four edges of the screen and change the direction number appropriately. For example, if it hits the bottom of the screen (y = 20) the vertical

only if the condition is true. Thus:

```
90 IF x = 31 THEN LET dx = -1
```

is exactly equivalent to:

```
90 IF x < > 31 THEN GO TO 100
```

```
95 LET dx = -1
```

```
100 ....
```

but the first alternative is shorter and easier to read.

The space provided in line 20 is to ensure that the first CHR\$ 8 doesn't try to go off the screen. Do you see why it is needed? The authors didn't until they tried to run the program.

Experiment 14b

Alter the behaviour of the bouncing ball program in the following ways:


- 1) Make it run slower.
- 2) Make it use a smaller 'billiard table'.
- 3) Make it leave a continuous trail of X's.

















<i>Experiment 14b completed.</i>	
----------------------------------	--

14.2 The Graphics Symbols

The output of a good program can often be made even better by underlining certain words or putting frames round key numbers. Charts and diagrams are also useful; they can give the user a much clearer picture of the results of a program than plain lists of figures. To help produce this kind of display the SPECTRUM offers you 16 'graphics' characters - shapes which can be built into diagrams, frames or pictures.

To use a graphics character you must first put the computer into GRAPHICS MODE by holding down CAPS SHIFT and hitting the GRAPHICS key (9). The cursor changes to a flashing G. If you now type any of the keys 1 to 8 you'll get the graphics character drawn on the key itself. If you hold down CAPS SHIFT while hitting one of the numbered keys, the character comes out reversed - that is, like a photographic negative.













The 16 graphics characters are shown below, together with the keys you need to get them. Note that they represent all possible ways of shading the four quarters of a character space, and that  (unshifted 8) is the same as a space.

Key	unshifted GRAPHIC	shifted GRAPHIC
1		
2		
3		
4		
5		
6		
7		
8		

To get out of the GRAPHICS mode, you must strike key 9 (with or without CAPS SHIFT). DELETE still works in the GRAPHICS mode but the other keys don't, so if you've been typing a string of graphic characters you must hit the '9' before you can type the closing double quotes.

To show you graphics in action, try typing the following program:

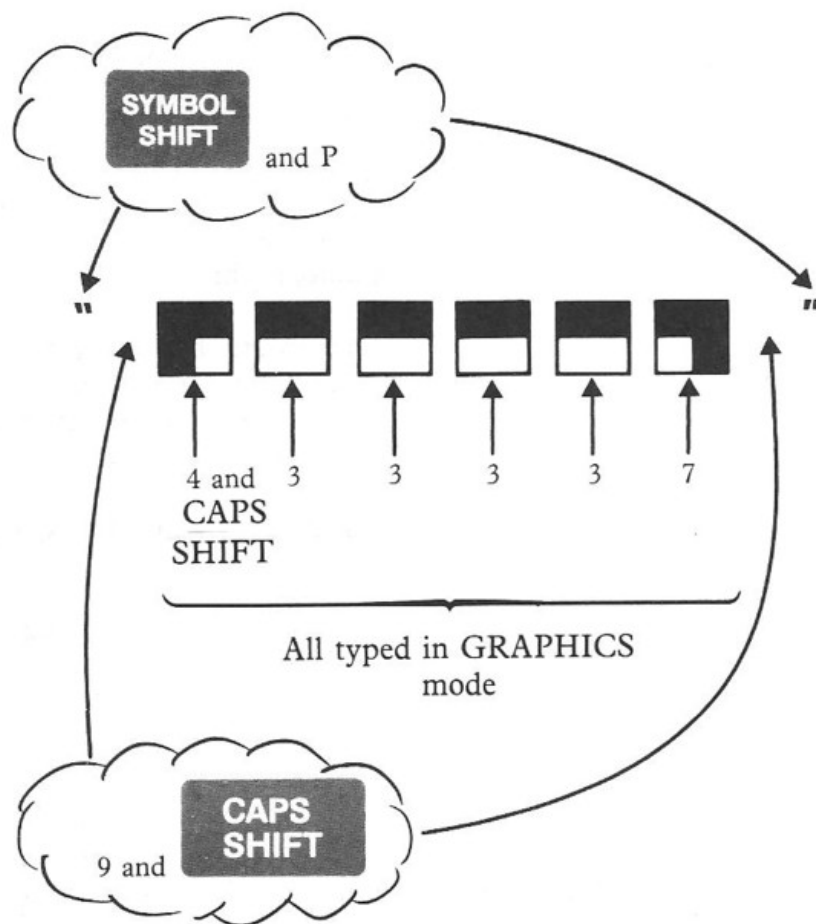
```

10 PRINT "       "
20 FOR j= 1 TO 4
30 PRINT "       "
40 NEXT j

```


At first, you'll find the lines with graphics hard to type correctly. Use the DELETE key as much as you like to correct your mistakes.

To type the string in line 10 you'll need to do the following:



If you've typed the program correctly you will get a framed chessboard when you give the RUN command.

Graphics symbols can be placed anywhere in the screen by using AT. This allows you to construct and display charts of every different sort. To give an example, here is a program which might be used to follow the election results in Transylvania - which is, as you doubtless know, a small country with only 31 parliamentary seats. There are three Political parties - the Teetotal Tories, the Anti-Blood Sports Federation and the Moral Anarchists. Any party which gains 16 seats in the election wins an outright majority and seizes absolute power until the next election. As the election results come in over the radio, you sit by the computer. Whenever a Teetotal Tory is elected, you type "tt". Likewise you type "ab" for an Anti-Blood Sports victory or "ma" for a Moral Anarchist win. The computer shows the state of the parties in vivid graphical form, and informs you immediately if any of the three parties should secure an outright win.

Load and run the program called "trs", getting the latest results from the BBC World Service. (If this happens to be a day when there is no election in Transylvania, you can make up your own). When you've

(Fig. 14.3) (listing of "trs")

```

10 REM Transylvanian Election Results
20 CLS
30 PRINT AT 2,0;" Absolute Majority"
40 PRINT AT 3,0;"-----"
50 PRINT AT 19,0;" Teetotal Anti-blood Moral"
60 PRINT AT 20,0;" Tories Sports Party Anarchists"
70 PRINT AT 18,0;"-----"
80 LET tt = 0
90 LET ab = 0
100 LET ma = 0
110 INPUT "Party of next winning candidate (tt, ab or ma)"; w$
120 IF w$ <> "tt" THEN GO TO 180
130 LET tt = tt + 1
140 PRINT AT 19 - tt,3;"███"
150 IF tt <> 16 THEN GO TO 300
160 PRINT AT 1,0; "Victory for the Teetotal Tories!"
170 STOP
180 IF w$ <> "ab" THEN GO TO 240
190 LET ab = ab + 1
200 PRINT AT 19-ab,13;"███"
210 IF ab <> 16 THEN GO TO 300
220 PRINT AT 1,1; "Anti-blood sports win."
230 STOP
240 IF w$ <> "ma" THEN GO TO 350
250 LET ma = ma + 1
260 PRINT AT 19-ma, 24;"███"
270 IF ma <> 16 THEN GO TO 300
280 PRINT AT 1,1; "Moral Anarchists sweep the field."
290 STOP
300 IF tt + ab + ma < 31 THEN GO TO 110
310 BEEP 1,5
320 PRINT AT 0,0; "No overall majority."
330 STOP
350 PRINT AT 0,1; "NO SUCH PARTY"
360 BEEP 2,18
370 PRINT AT 0,1;"      "
380 GO TO 110

```

14.3 User-defined Graphics

As well as providing you with 16 fixed graphical characters for drawing charts and frames, the SPECTRUM lets you design and use up to 21 symbols of your own. The 'user-defined Graphics' (they ought really to be called 'programmer-defined') give your program its own unique personal quality and help to distinguish it from the work of other people. In this area, you will find an artistic sense of shape and form to be a distinct advantage.

Every character the SPECTRUM can display is built up of a square grid of 8×8 or 64 points. When a character is displayed some of these points are bright, whilst others are dark, and the points are so close together that to the human eye they seem to merge and form a continuous shape. For example, the pattern of points which forms a 'd' is:

```

      O
      O
      O
      O
    OOOO
  O   O
    OOOO
  
```

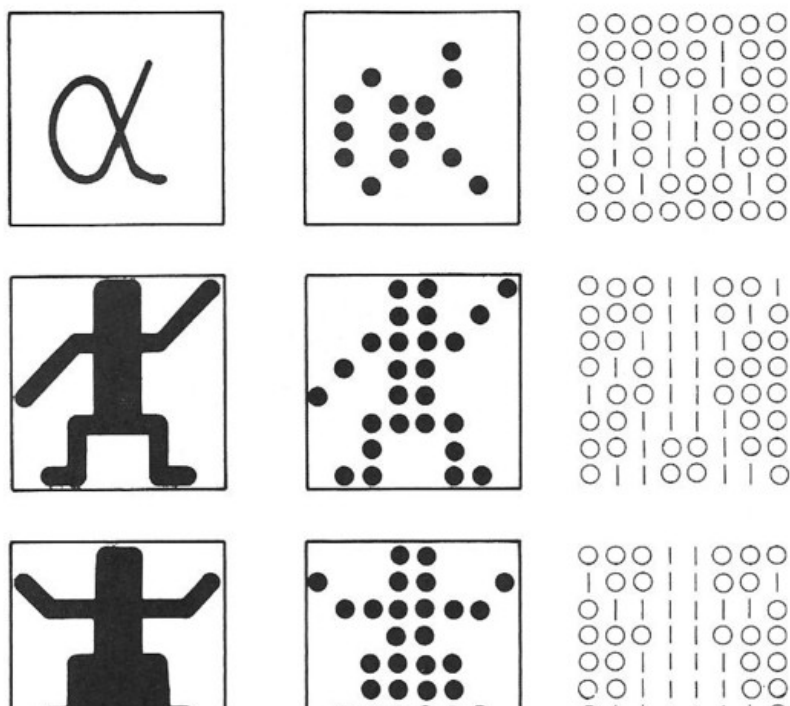
The characters you design can fill the gaps in the set provided by the SPECTRUM. They could, for instance, be letters in a foreign language, or shapes for drawing sloping lines, or tokens for use in games.

Here's how you set about designing your own characters:

First, rule an 8×8 square box on your squared paper. Then draw your character into the box, using thick lines or solid areas as far as possible.

Next, mark all the squares which are completely or at least mainly included in your character. You may find it convenient to use a separate box, as we have done in Figure 14.4.

(Fig. 14.4)



Finally, work down the character row by row, writing each row as a line - 1's for the points you have marked and 0's for the others. You should end up with 8 rows of 8 0's or 1's each.

Once your character is designed, you can build it into your program. First you decide which of the 21 letters a to u (but not v - z) you will choose to stand for the character. You can usually find a suitable letter like 's' to show a star or 'm' to show a man.

Then you must write down a DATA statement. It will have 9 items in it. The first is a string with the letter you want to stand for your character. The next 8 items are the eight rows of the character. Each item starts with the symbol BIN (B in Extended mode) and is followed by the row of 1's and 0's taken from the design of the character. The items correspond to the rows from the top down.

To give an example, here is the DATA statement you would write for the Greek symbol α in Figure 14.4, shown on several lines for clarity.

```
10 DATA "a",  
BIN 00000000,  
BIN 00000100,  
BIN 00100100,  
BIN 01011000,  
BIN 01011000,  
BIN 01010100,  
BIN 00100010,  
BIN 00000000
```

You put down a similar DATA statement for each of the characters you wish to define. Naturally they must all have different letters to represent them.

Next, you include a piece of program which reads the DATA statements and stores details of the characters in the memory of the SPECTRUM. This is a somewhat involved process, and we don't propose to explain it in detail until later in the course. At this point, just copy it down, putting in suitable line numbers:

```
READ q$  
FOR j = 0 TO 7  
READ r  
POKE USR q$ + j, r  
NEXT j
```

The keyword POKE is on key O, and USR is on L in Extended mode.

If you have more than one character to define, you should arrange to repeat this piece of code the right number of times - preferably by enclosing it in an outer FOR loop. Also put the DATA statements in the right place so that they don't get read and misunderstood by any other part of the program.

You can use your new characters as soon as their definitions have been read. Suppose you've defined a new character and identified it by the letter 'a'. Then whenever you put an 'a' in graphics mode your character will be displayed instead.

To illustrate, here is a program which displays groups of people. We

(Fig. 14.5)

```

10 REM People
20 DATA "m", BIN 00011001, BIN 00011010, BIN 00111100,
  BIN 01011000, BIN 10011000, BIN 00111100,
  BIN 00100100, BIN 01100110
30 DATA "f", BIN 00011000, BIN 10011001, BIN 01111110,
  BIN 00011000, BIN 00111100, BIN 00111100,
  BIN 01111110, BIN 00101000
100 FOR n = 1 TO 2
110 READ a$
120 FOR j = 0 TO 7
130 READ r
140 POKE USR a$ + j, r
150 NEXT j
160 NEXT n
200 CLS
210 FOR j = 1 to 234
220 PRINT "mf"; (type mf in graphics mode)
230 NEXT j

```

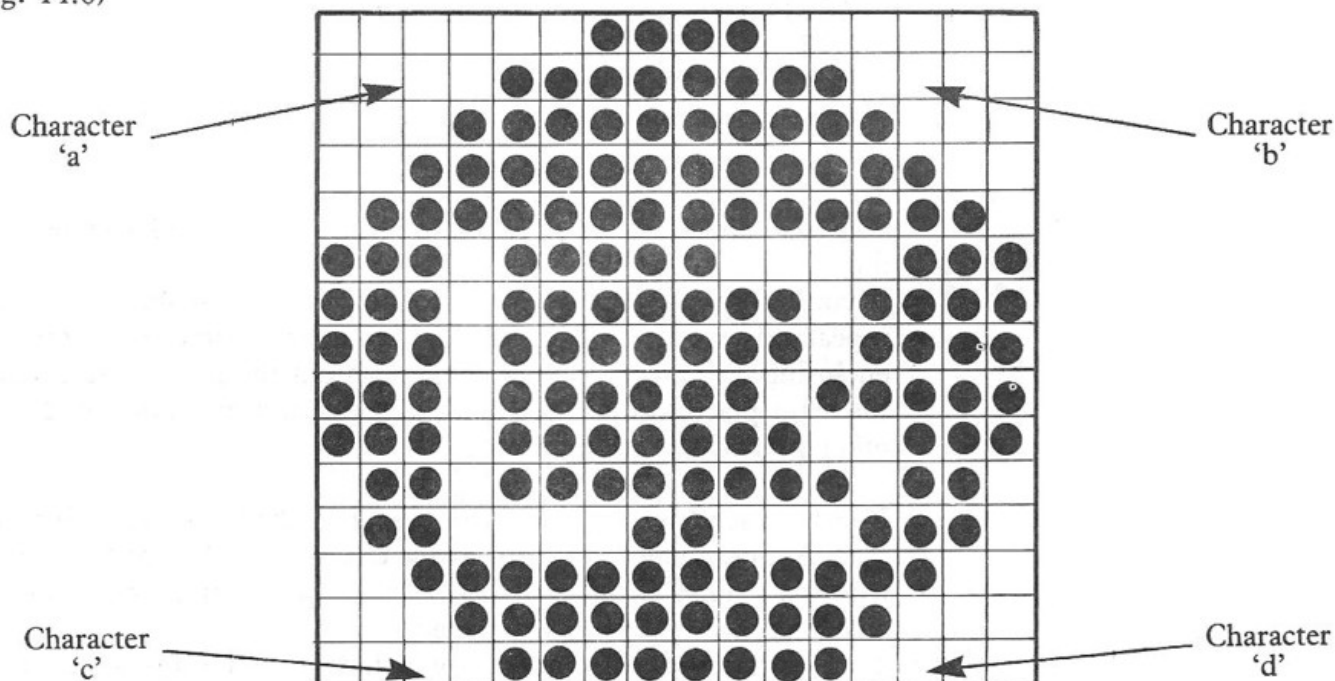
When you stop the program, the new characters will still be defined. Each one will remain in the SPECTRUM'S memory until you turn off the power or define a different character with the same name. Thus if you now type 'm' or 'f' in graphics mode, (these being the letters which represent the figures), you will see that character appear as part of your command. If you list the program, PRINT "mf"; (where the m and f were originally typed in GRAPHICS mode) will now appear as

PRINT "   ";

When designing characters, remember that the 8 * 8 grids of all the screen positions actually touch, and if you want a margin round your character it is best not to use the points on the outside. On the other hand, it is often useful to run individual characters together to make bigger and more complex shapes.

Here is an example, which uses a 'ball'.

(Fig. 14.6)



We divide this shape into four characters and name them:

a b
c d

To display the ball at any screen position (say column y and row x) we would put:

```
PRINT AT y,x;"ab"  
PRINT AT y+1,x;"cd"
```

GRAPHICS MODE



The program below is a modified version of the one used to draw a bouncing ball. Load it from the tape (it is called "bounce") and run it to see the general effect.

(Fig 14.7)

```
10 REM Logic 3 Bouncing Ball
20 DATA "a", BIN 000000011, BIN 00001111, BIN 00011111,
  BIN 00111111, BIN 01111111, BIN 11101111, BIN 11101111,
  BIN 11101111
30 DATA "b", BIN 11000000, BIN 11110000, BIN 11111000,
  BIN 11111100, BIN 11111110, BIN 10000111, BIN 11101111,
  BIN 11101111
40 DATA "c", BIN 11101111, BIN 11101111, BIN 01101111,
  BIN 01100001, BIN 00111111, BIN 00011111,
  BIN 00001111, BIN 00000011
50 DATA "d", BIN 11011111, BIN 11100111, BIN 11110110,
  BIN 10001110, BIN 11111100, BIN 11111000, BIN 11110000,
  BIN 11000000
100 FOR n = 1 TO 4
110 READ q$
120 FOR j = 0 TO 7
130 READ r
140 POKE USR q$ + j,r
150 NEXT j
160 NEXT n
180 CLS
190 PRINT "    "
200 LET x = 0
210 LET y = 0
220 LET dx = 1
230 LET dy = 1
240 LET xo = x
250 LET yo = y
260 LET x = x + dx
270 LET y = y + dy
280 IF x = 30 THEN LET dx = -1: BEEP 0.03, 11
290 IF x = 0 THEN LET dx = 1: BEEP 0.03, 7
300 IF y = 19 THEN LET dy = -1: BEEP 0.03, 14
310 IF y = 0 THEN LET dy = 1: BEEP 0.03, 17
320 PRINT AT yo, xo; "    ": REM Rub out old ball
330 PRINT AT yo + 1, xo; "    "
340 PRINT AT y, x; "ab"
350 PRINT AT y + 1, x; "cd" (type ab and cd in graphics mode)
```


Experiment 14c

A. Modify program "bounce" so that it moves round a different shape (perhaps an egg) with your own initials on it.

B. Define characters with lines (-|/\) and use them to draw various geometric figures.

C. Design a monster, a shark or a space-ship (taking a number of characters in a group) and write a program to make it move about the screen.

<i>Experiment 14c completed.</i>	
----------------------------------	--

Chapter 15

The Colour Code

The Border

Ink and Paper

Random numbers

User-defined characters in colour

Flash

This chapter introduces you to the use of colour on the SPECTRUM. The colour facilities on this machine are among the best you'll find on any home micro-computer, and to see them in their full splendour you will need a good colour television set, well adjusted. If you have only a black and white receiver, all you will get is various shades of grey.

15.1 The Colour Code

The SPECTRUM can show 8 different colours. Each colour has its own number, which you will soon remember:

BLACK	BLUE	RED	PURPLE	GREEN	CYAN	YELLOW	WHITE
0	1	2	3	4	5	6	7

15.2 The Border

The easiest part of the screen to control is the border which surrounds the display. To change the border to any of the eight colours, just give the command:

BORDER n

where n is a colour number in the range 0 to 7. For example:

BORDER 2

will turn the border red.

The border colour also fills in the lower part of the screen, where the INPUT prompt normally appears.

Try the following program, which will show you all the border colours one after another:

```
10 FOR j = 0 TO 7
20 BORDER j
30 FOR w = 1 TO 500
40 NEXT w
50 NEXT j
60 GO TO 10
```

15.3 Ink and Paper

We now turn our attention to the main display. To understand what happens when the SPECTRUM shows you a character, a useful analogy is that the symbol is drawn with 'ink' of one colour on 'paper' of a different (or possibly the same) colour. When the machine is started, it uses black ink on white paper, so you see black characters on a white background.

You can change the colours of both ink and paper as often as you like. The command

INK n

(where n is a colour number) will select the new colour immediately. For instance:

```
10 PRINT "SINCLAIR"
20 INK 4
30 PRINT "SPECTRUM"
```

will display SINCLAIR in black, but SPECTRUM in green.

The paper colour can be changed, just as easily, by the command

If you change the ink or paper colours, the change affects all those characters (and only those characters) which are displayed after the change. When a program ends, the current colours are not reset, so a program listing often looks odd and is sometimes unreadable. A quick cure is to type the command line:

INK 0 :PAPER 7: CLS


The CLS command will clear the whole screen to the current paper colour. Ordinary PRINT commands use the PAPER colour as background for all the characters they display.

You can actually put INK and PAPER commands as items into PRINT commands. They work just as before, except that their effect lasts only as far as the end of the command. In this context, each INK and PAPER command will always be followed by a semicolon. Type NEW and try this short program:










```
10 PRINT "This line is"  
20 PRINT PAPER 0 ;INK 7;"emphasised"  
30 PRINT "but this one is not."  
40 STOP
```

There are many ways of painting pictures in colour. One of the easiest is to use character-wide 'blobs' of paint and to build them up into larger areas of colour. The 'space' character will produce a blob of the current paper colour, and □(shifted graphic 8) will be the colour of the ink. Here, for instance, is a program to display the Italian flag (vertical stripes of green, white and red). The program doesn't use INK at all, but relies on using paper of different colours to build up the flag.

```
10 REM Italian Flag  
20 BORDER 0  
30 FOR j=0 TO 21  
40 PRINT PAPER 4;"          ";;REM eleven green spaces  
50 PRINT PAPER 7;"          ";;REM ten white spaces  
60 PRINT PAPER 2;"          ";;REM eleven red spaces  
70 NEXT j  
80 STOP
```

The program below uses a slightly different method to draw the Swiss flag. It defines red paper and white ink, and draws the central white cross using the  graphics character.

```

10 REM Swiss flag
20 BORDER 0
30 PAPER 2:REM red paper
40 INK 7:REM white ink
50 FOR j = 0 TO 4
60 PRINT " <----- 32 spaces -----> ":REM 32 red spaces
70 NEXT j
80 FOR j=5 TO 8:REM upper arm of cross
90 PRINT " <----- 14 spaces ----->     <----- 14 spaces -----> "
100 NEXT j
110 FOR j = 9 TO 12:REM centre of cross
120 PRINT " <----- 9 spaces -----> 14  's <----- 9 spaces -----> "
130 NEXT j
140 FOR j=13 TO 16:REM lower arm of cross
150 PRINT " <----- 14 spaces ----->     <----- 14 spaces -----> "
160 NEXT j
170 FOR j=17 TO 21
180 PRINT " <----- 32 spaces -----> "
190 NEXT j
200 STOP

```

15.4 RND

One useful feature in making drawings (and indeed in many other fields) is RND. Whenever you put this word into a command the computer produces a decimal fraction selected at random between 0 and 1. For example, if you type:

```

10 FOR j = 0 TO 20
20 LET r = RND
30 PRINT r
40 NEXT j
50 STOP

```

you will get 20 numbers, scattered evenly between 0 and 1. If you run the program again you'll get a different set (this may surprise you, since up to this point the same programs, if they had no INPUT commands, always produced the same result!). Furthermore, the numbers will be different every time, and there is no easy way to predict what they will be. They are called 'random numbers'. (A better name is 'pseudo-random', since if you know how the machine calculates them you can in principle predict each one from the previous one; but the computation is highly involved).

In many applications you'll want random numbers which aren't decimal fractions but whole numbers in a given range. For example, if you are making the machine pretend it's throwing dice, you will need numbers in the range 1 to 6 (and equally likely to have any of these values). If you are painting an abstract picture and want to colour part of it at random, you will need a colour number in the range 0 to 7. A 'distribution' of this kind is quite easy to arrange. You use the expression:

```
face = INT (1 + RND * 6)
```

or

```
colour = INT (0 + RND * 8)
(in practice you might leave out the '0+')
```

Here is an example which gives you 72 throws of a single die. Run it and then examine the result to see if the number of one's, two's, doubles (sequence of two or more numbers the same) and so on tallies roughly with what you expect. It is most unlikely to tally exactly!

```
10 FOR j = 1 TO 72
20 PRINT INT(1 + RND * 6);" ";
30 NEXT j
40 STOP
```

The next program plays 'random music' on the SPECTRUM. Each note has a length of 0 to 0.4 seconds, and the pitch is somewhere in the range -15 to +25.

```
10 REM Random music
20 BEEP 0.4*RND, INT(-15+RND * 40)
30 GO TO 20
```

The program below paints a never-ending series of rectangles of random size and colour. The main loop starts at line 40 and begins with the selection of the top and bottom rows of the rectangle (t and b, whole numbers in the range 0 to 21). If the bottom row turns out not to be below the top row (as it must be to draw a sensible rectangle) the pair of values is simply discarded and the machine picks another pair instead. The test is in line 60.

The left and right-hand columns are selected in a similar way in lines 70-90, and the colour of the rectangle is set in line 100. Finally, the rectangle is painted by the commands in lines 110 to 150.

Type the program in and watch the patterns it makes on the screen.

```
10 BORDER 0
20 PAPER 0
30 CLS
40 LET t= INT(21*RND):REM set top row
50 LET b= INT(21*RND):REM set bottom row
60 IF t >= b THEN GO TO 40:REM check values
70 LET a= INT(31*RND):REM set left column
80 LET r= INT(31*RND):REM set right column
90 IF a >= r THEN GO TO 70:REM check values
100 INK INT(8*RND):REM set colour
110 FOR j = t TO b: REM paint rectangle
120 FOR k = a TO r
130 PRINT AT j,k;"■";
140 NEXT k
150 NEXT j
160 GO TO 40
```

15.5 User-Defined Characters in Colour

If you define your own characters, you can make pictures with a great deal of fine detail. The main restriction is that the colour of each character space on the screen is governed by the paper and ink colours when the character is displayed. There is no way to get more than two

user-defined graphics. Load and run it, and see if you can spot the non-standard graphics for yourself. They are of course the triangle needed for the side of the roof and the crosses used to draw the windows.

To write this program a sketch of the picture was first drawn on squared paper. The exact positions of the chimney, the door, etc. were then found by counting squares.

A full listing of the program is given below. It is long but completely straightforward. It may help you to know that *j* is always the row number, and *k* (when it is used) is the column number.

A feature of the program which may be new to you is that several of the lines contain two or more commands, separated by colons. This is quite acceptable, provided that your program does not need to GO TO one of the later (unnumbered) commands in the line.

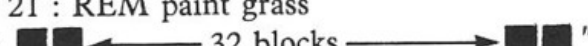
Another unfamiliar element of programming style appears in line 120. Here the numerical variables 'black', 'blue' etc. are given the values which correspond to the code numbers of those colours. The point of this line is that we can now write commands like:

```
BORDER red
INK yellow
or PAPER green
```

instead of

```
BORDER 2
INK 6
and PAPER 4
```

This (we believe) makes the program more readable.

```
10 REM The Wee Hoose
20 DATA "a",BIN 00000001, BIN 00000011, BIN 00000111,
  BIN 00001111, BIN 00011111, BIN 00111111,
  BIN 01111111, BIN 01111111
30 DATA "b",BIN 10000000, BIN 11000000, BIN
  11100000, BIN 11110000, BIN 11111000, BIN 11111100,
  BIN 11111110, BIN 11111111
40 DATA "c",BIN 00011000, BIN 00011000,
  BIN 00011000, BIN 11111111, BIN 11111111, 00011000,
  BIN 00011000, BIN 00011000
50 FOR j = 0 TO 2: REM set up user-defined graphics
60 READ q$
70 FOR k = 0 TO 7
80 READ r
90 POKE USR q$ + k,r
100 NEXT k
110 NEXT j
120 LET black = 0: LET blue = 1: LET red = 2: LET
  purple = 3: LET green = 4: LET cyan = 5:
  LET yellow = 6: LET white = 7
130 BORDER black
140 PAPER blue: INK green
150 CLS: REM paint sky
160 FOR j = 12 TO 21: REM paint grass
170 PRINT AT j, 0: 
```

```

210 PRINT AT j,4; " ■■ ←———— 24 blocks —————→ ■■ "
220 NEXT j
230 INK black: REM paint roof
240 FOR j = 10 TO 5 STEP -1
250 PRINT AT j, 13 - j; "a";
260 FOR k = 14 - j TO 17 + j: PRINT " ■ ";: NEXT k
270 PRINT "b"
280 NEXT j
290 REM chimney
300 INK red
310 PRINT AT 4, 10; "■■■■■"
320 PRINT AT 3, 10; "■ ■ ■"
330 PRINT AT 2, 10; "■ ■ ■"
340 INK white
350 PRINT AT 1, 10; " O o O"
360 PRINT AT 0, 10; " o o"
370 REM Windows
380 INK green: PAPER white
390 FOR k = 6 TO 21 STEP 5
400 FOR j = 12 TO 15
410 PRINT AT j, k; "cc"
420 NEXT j
430 NEXT k
440 REM Door
450 PAPER black
460 FOR j = 13 TO 18
470 PRINT AT j, 24; " "
480 NEXT j
490 INK yellow: REM Doorknob
500 PRINT AT 16, 26; "o"
510 GO TO 510: REM loop stop

```

15.6 Flash

There is yet another way you can affect the appearance of characters on the screen. The command:

FLASH 1

will make all the characters you display from that point flash by switching the ink and paper colours backwards and forwards. The effect lasts until you give the command:

FLASH 0

Try typing the following program:

```

10 PRINT "Bad brakes spell"
20 FLASH 1
30 PRINT "DANGER!"

```

When you have run this program, list it. Then clear the screen. You will see that the FLASH command continues to affect all characters even after the end of the program. To get rid of it, type:

Experiment 15a

Use the computer to paint a picture of a lighthouse, with a flying flag and a flashing lantern. You might include some moving ships or birds.

<i>Experiment 15a completed.</i>	
----------------------------------	--

Chapter 16

INKEY\$
Simple games
Afterword

This is the last chapter of Part 1. In it we shall introduce two simple new concepts, and then show how all the ideas and facilities in the book can be combined in a single program.

16.1 INKEY\$

The main use of the keyboard on your SPECTRUM is to type programs and data. However, the keyboard also provides you with an extremely effective set of 'buttons' for programs where the user has to respond to a changing situation, such as a test of speed and co-ordination or a game of skill. The INPUT command is not suitable for this kind of application because

- a) Every item typed would have to be followed by ENTER
- b) While the machine is waiting for the user's response, everything else stops. This means that you can't organise a moving picture.

These drawbacks are overcome by the INKEY\$ facility, which provides exactly what is needed. The word INKEY\$ (on N in extended mode) can be used wherever you would normally write a string or a string variable. When the SPECTRUM works out an expression with INKEY\$, it supplies a string which is derived from the keyboard in the following way:

If exactly one key is being held down at that moment, INKEY\$ gives a string with a single character - the one being pressed.

If no keys are being pressed, or if more than one is held down, INKEY\$ delivers the null string, which contains no characters and is often written "".

In neither case does the machine wait for the user to do anything; it just looks at the keyboard and takes its state at that moment.

To get a feel for INKEY\$, type in the following program and try it out. Find out what happens when you type both ordinary and shifted characters.

```
10 LET a$ = INKEY$
20 PRINT a$;
30 GO TO 10
```

The simplest application of INKEY\$ is to make the machine wait until any key is pressed. This can be done by a single command:

```
100 IF INKEY$ = "" THEN GO TO 100
```

Clearly any other line number would work just as well.

As you will have noticed, INKEY\$ will keep on repeating its information until a key is released. In many cases, therefore, it is best to check that a key has been pressed and released before going on. The release can be checked by

```
110 IF INKEY$ <> "" THEN GO TO 110
```

Sometimes you will want the computer to wait for a particular character, such as 'W'. This can also be done in one command:

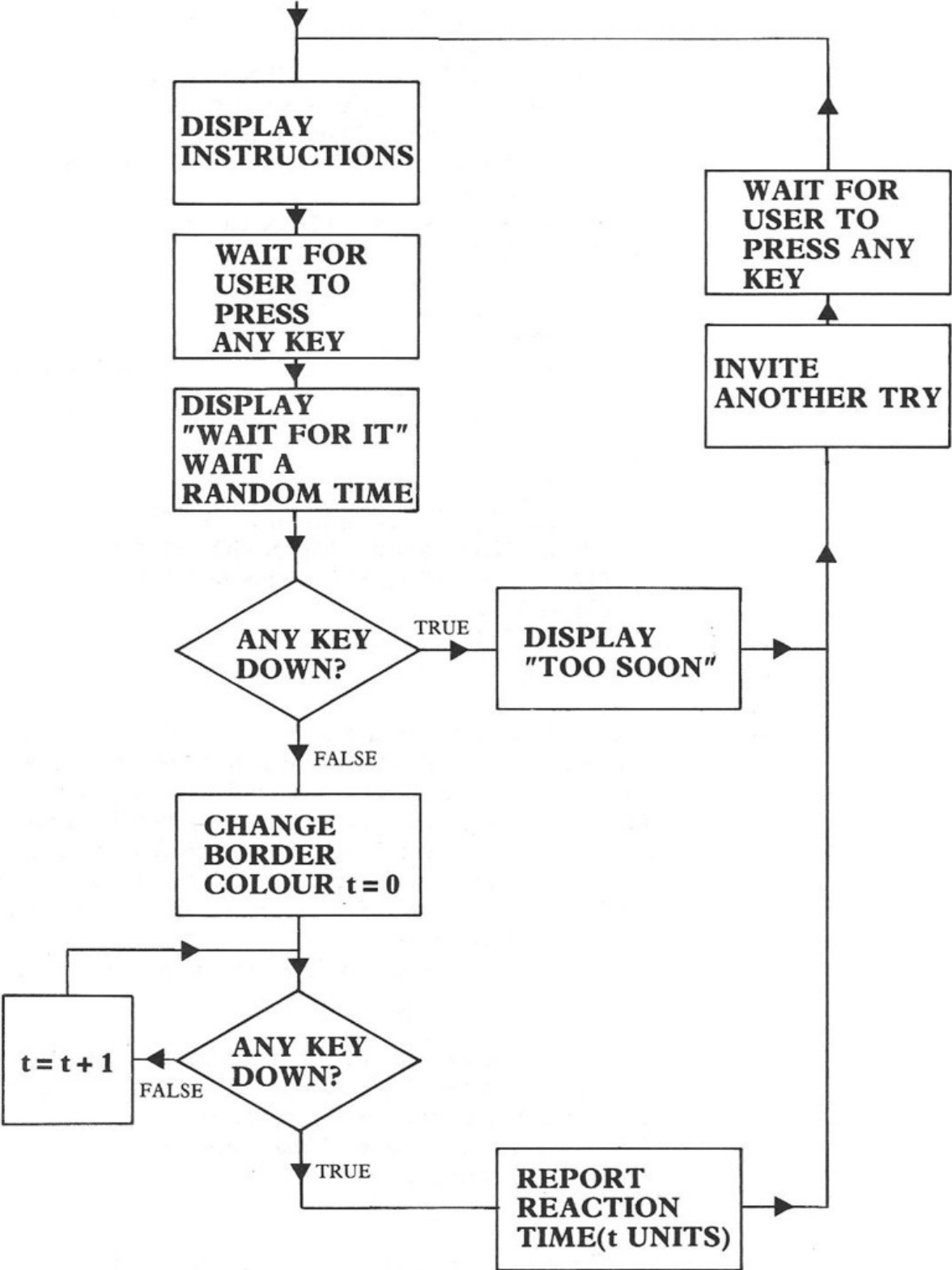
```
120 IF INKEY$ <> "W" THEN GO TO 120
```

These methods can be used in a program which measures your

as quickly as you can, by hitting any key. The program then tells you how long you took, in hundredths of a second. If you're reasonably wide awake you should have a reaction time of less than 0.25 seconds. The program is provided on the tape, under the name "reaction". Load it and try it out a few times. Now study the flow chart and program below, to see how it works.

Fig. 16.1

(Flow chart for reaction time)




```
10 REM Reaction time
15 INK 0: PAPER 7
20 BORDER 0: INK 0: PAPER 7: CLS
30 PRINT "Reaction Time Program"
40 PRINT
50 "Press any key as soon as you"
60 PRINT "are ready to start. Then wait"
70 PRINT "for the border to change colour"
80 PRINT "and hit any key as fast as you"
90 PRINT "can."
100 IF INKEY$ = "" THEN GO TO 100
110 BEEP 0.5,8
120 IF INKEY$ <> "" THEN GO TO 120
130 LET j = INT (200 + RND * 1000)
140 PRINT "WAIT FOR IT!"
150 FOR k = 1 TO j: NEXT k: REM wait a random time
160 LET t = 0
165 IF INKEY$ <> "" THEN GO TO 300
170 BORDER 7
180 LET t = t + 1
190 IF INKEY$ = "" THEN GO TO 180
200 BEEP 0.1, 18
210 PRINT 220 PRINT "Your reaction time was about"
230 PRINT t*4000/4071;" hundredths of a second."
240 IF INKEY$ <> "" THEN GO TO 240: REM wait for
    key to be released.
250 PRINT
260 PRINT "To try again hit any key"
270 IF INKEY$ = "" THEN GO TO 270
280 IF INKEY$ <> "" THEN GO TO 280
290 GO TO 20
300 PRINT "Too soon !"
310 GO TO 240
```

There is little to be explained. Line 165 (which we confess, is a late addition) serves to catch those who hope to get a short reaction time by pressing down a key before the stimulus is presented.

The main timing loop is in lines 180 and 190. Variable t counts the number of times that the loop is obeyed before the user responds.

There is, of course, no reason to suppose that the loop will count exact hundredths of a second. The program had to be calibrated, and this was done as follows:

First, a temporary version of line 230 was put in. It read:

```
230 PRINT t; "units"
```

Next, the program was run with a measured delay of 40 seconds (although any other period long enough to measure accurately would have served). The program reported a reaction time of 4071 units.

Since the exact response would have been 4000, the correction factor is 4000/4071, and this has been incorporated in the final version of line 230.

INKEY\$ can be used to control a moving object such as a 'man' in a video game. Here, for example, is a short program which moves a character backwards and forwards along the bottom row. 'm' means

```

10 LET x = 15 :REM set starting position
20 PRINT AT 21,x;"o";:REM display character
30 IF INKEY$ <> "n" THEN GO TO 80
40 IF x=0 THEN GO TO 30 :REM check range
50 LET x=x-1:REM move left
60 PRINT CHR$ 8;" ";: REM erase old character
70 GO TO 20
80 IF INKEY$ <> "m" THEN GO TO 30
90 IF x=31 THEN GO TO 30 :REM check range
100 LET x=x+1:REM move right
110 PRINT CHR$ 8;" ";:REM erase old character
120 GO TO 20

```

Key this program and try it out. It will give you some idea of the speed at which the machine can react.

16.2 Simple Games

To end the book, we will describe a very simple video game. Please note that for the sake of simplicity and brevity the game has been reduced to the barest essentials. There are no elaborate displays, no user-defined characters, or sophisticated musical accompaniments. All these features could easily be added using the various commands we have already studied. They would make the program look much more like a commercial product, but they would tend to obscure the basic principle of the game itself.

The subject of our discussion is a program called "game". Load it and play it through several times, in order to get a feel for its main features.

How would you set about designing a program for such a game? A good place to start is to write down a 'model' for the situation which the game represents at any moment. A model, in this context, is a set of variables which, when taken together, allow the picture on the screen to be reconstructed and keep track of any other salient points. In this case, the model contains a number of variables, which can be arranged in groups:

a) The man on the bottom line: the only thing the program needs to know about the man is his position. It is kept in variable x.

b) The missiles: at any moment the program needs to know whether a missile is on its way up the screen or not. If there is no missile flying, variable b is set to zero. Otherwise, b is set to 1 and variables bx and by indicate the current position of the missile as column and row numbers. Note that the column number is not necessarily the same as that of the man, since he can move away after firing. The number of missiles currently left is stored in n.

c) The target: at any given moment, the target may or may not exist. If it does not, then variable te is set to 0 and tx and ty are both set to -1. If there is a target, te becomes 1, and tx and ty record the position of the target on the screen as row and column numbers. Every target has a limited life, and ex holds the time at which it is due to disappear.

d) Time: the time since the game was started (in arbitrary units) is

To summarise, we have planned for a number of variables, which, between them, completely describe the situation of the game. They are:

Variable	Purpose
x	Position of man
b	1 if missile in flight, else 0
bx,by	Position of missile (if any)
n	Number of missiles left
te	1 if target showing
tx,ty	Position of target (if any); else -1
ex	Expiry time of target
t	Time
s	Score

Now we consider the overall structure of the program. It has a beginning, where the instructions are displayed, a middle during which the game is played, and an end, where the final score is displayed and the player given the chance to have another try. The beginning is entirely straightforward. Apart from putting instructions up on the screen (which any game should do) it sets up suitable starting values for the variables. Thus n, the number of missiles, is set to 10. b and te are both set to zero (since at the beginning there is no missile or target) and the man is put in the middle of his patch by setting x=15.

The middle part is one large loop, which continues to run until there are no missiles left. t (the 'time' variable) simply counts the number of times round the loop.

During the execution of the loop, the computer has to fulfil a number of jobs. These are listed below:

1. (Line 170). The computer displays the current score and missile stock, every time either of these changes .
2. (Lines 180-280). The machine displays the man, and then checks the keyboard. If necessary, it moves the man left or right as the player indicates (provided that this does not take him off the screen).
3. (Lines 290 to 345). Next, the machine again checks the keyboard to see if the user has pressed "z" to fire a missile. If so, a missile is started on its way by setting variables b,bx and by. This whole step is left out if there is already a missile in the air.
4. (Lines 350 to 420). This section moves the missile up one row, and works out the consequences. The section is omitted if there is no missile in the air at the moment. The basic movement is produced in lines 360 which erases the current missile, and line 370, which moves its position up one row. Now there are three possibilities:
 - a) The missile hits the top of the screen (by=0). In this case the program jumps to 600.
 - b) The missile hits the target (this occurs when bx=tx and by=ty). On a direct hit the machine jumps to line 700.
 - c) The missile doesn't hit anything. In this case it is displayed in its new position and the program continues in sequence.

If there is no target, the program makes a random decision. The condition:

$RND > 0.1$

in line 460 will be true about nine out of every ten times through the section. When it is false, the computer selects a random position for the target and displays it on the screen. It sets variable *te* to show that there is a target, and decides on a lifetime for it, also at random, in line 480. The information is actually recorded as a 'date of death' in variable *ex*. To make the game easier or more difficult, you can adjust the numbers in this command to give you longer or shorter target lifetimes.

6. (Lines 500 to 530). Here the program checks whether the lifetime of the current target is up, by testing whether (*ex* = *t*). If so, the current target is removed from the screen and the 'target' group of variables is reset accordingly.

7. (Lines 550, 560). Here the time variable *t* is advanced by one and the loop jumps back to the start.

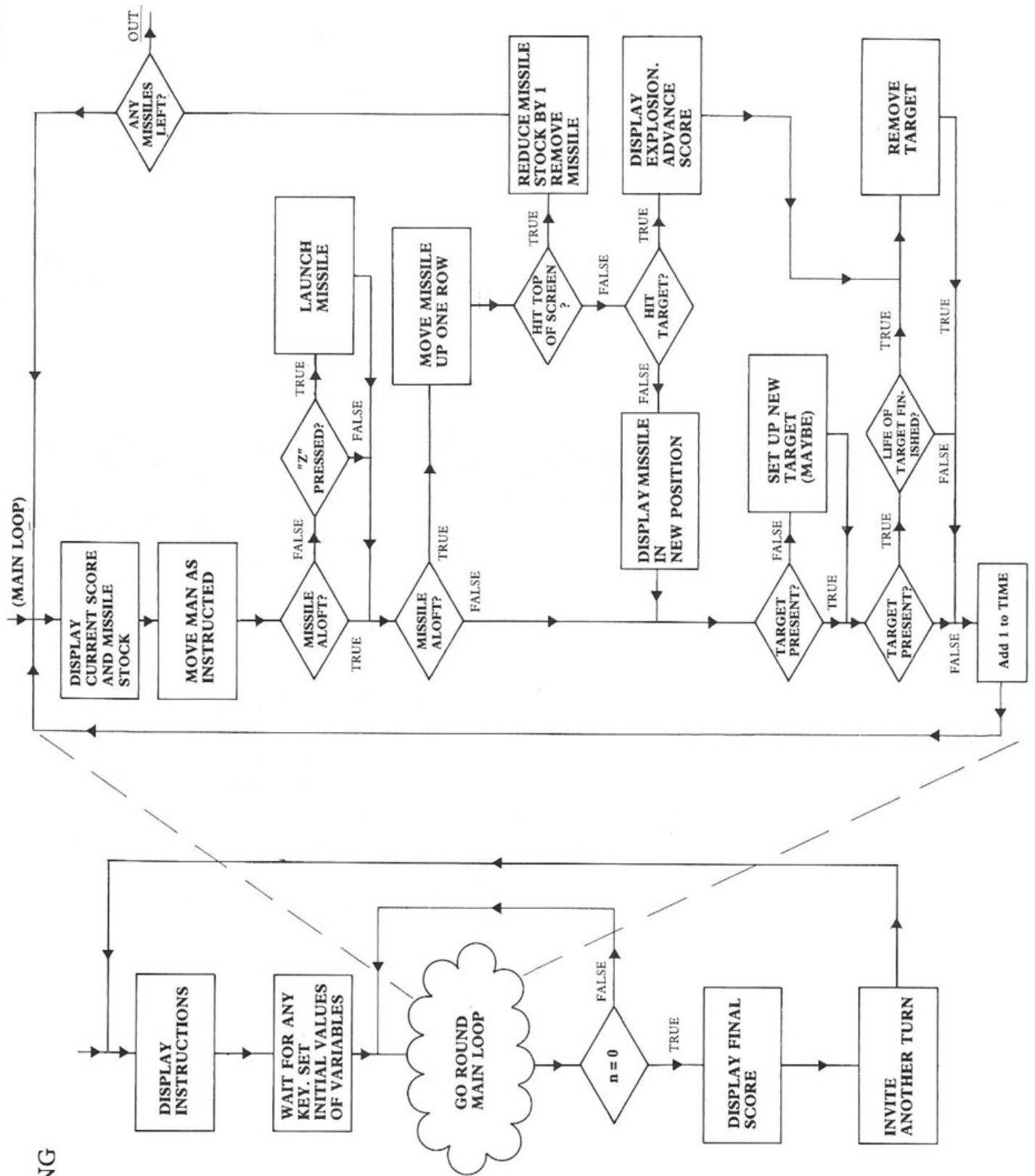
8. (Lines 600-640). This section is only used when a missile hits the top of the screen. The missile is removed by setting *b* = 0, and *n*, the number of available missiles, is reduced by 1. If there are any missiles left the program returns to the main loop to continue the game. Otherwise it goes on to:

9. (Lines 650 to 680). This is the 'end' of the program. The machine displays the final score and waits for the user to hit a key if he wants to repeat the game.

10. (Lines 700 to 780). The player has scored a hit and deserves a show. He gets a cheerful BEEP and the target flashes a few times before disappearing. The player earns a score which is equal to the unexpired life of the target, on the grounds that the quicker you shoot it down the more it's worth. The player also miraculously gets his missile back, because the program omits to decrease the missile count.

A flow chart for the game is shown on page 16.8

Fig 16.2) TARGET SHOOTING



Here is the actual code:

```

10 REM Target shooter
20 CLS
30 PRINT: PRINT "Shoot down as many targets as"
40 PRINT "you can. Press m to go right, "
50 PRINT "n to go left, and z to shoot."
60 PRINT: PRINT "Now hit any key to start"
70 IF INKEY$ = "" THEN GO TO 70
80 IF INKEY$ <> "" THEN GO TO 80
100 LET t = 0
110 LET s = 0
120 LET n = 10
130 LET b = 0
140 LET ex = -1: LET te = 0: LET tx = -1: LET ty = -1
150 LET x = 15
160 CLS
170 PRINT AT 0,0; "SCORE  ";s;"MISSILES LEFT
    ";n; "  "
180 PRINT AT 21,x; " ■ ";
190 IF INKEY$ <> "n" THEN GO TO 240
200 IF x = 0 THEN GO TO 240
210 PRINT AT 21,x;" ";
220 LET x = x - 1
230 PRINT AT 21,x; " ■ ";
240 IF INKEY$ <> "m" THEN GO TO 290
250 IF x = 31 THEN GO TO 290
260 PRINT AT 21,x; " ";
270 LET x = x + 1
280 PRINT AT 21,x; " ■ "
290 IF b = 1 THEN GO TO 350: REM jump if bullet already
    flying
300 IF INKEY$ <> "z" THEN GO TO 350
310 REM start bullet
320 LET b = 1
330 LET bx = x
340 LET by = 20
345 PRINT AT by, bx; "!";
350 IF b = 0 THEN GO TO 450
360 PRINT AT by, bx; " ";
370 LET by = by - 1
380 IF by = 0 THEN GO TO 600
390 IF bx <> tx THEN GO TO 420
400 IF by <> ty THEN GO TO 420
410 GO TO 700: REM direct hit
420 PRINT AT by, bx; "!";
450 IF te <> 0 THEN GO TO 500
460 IF RND > 0.1 THEN GO TO 500
470 LET te = 1: LET tx = INT (32 * RND): LET ty = INT
    (2 + 16 * RND)
480 LET ex = t + INT (10 + 40 * RND)
490 PRINT AT ty,tx; "◎";
500 REM check if target life up
510 IF ex <> t THEN GO TO 550
520 PRINT AT ty,tx; " ";
530 LET ty = -1: LET tx = -1: LET te = 0

```



```
610 LET b = 0
620 LET n = n - 1
630 BEEP 0.2, 3
640 IF n > 0 THEN GO TO 170
650 CLS
660 PRINT AT 4,4: "FINAL SCORE WAS "; s
670 FOR g = 1 TO 1000: NEXT g
680 GO TO 10
700 REM Direct hit
710 FLASH 1
720 BEEP 0.05, 27
730 PRINT AT ty, tx; " © "
740 LET s = s + ex - t
750 FOR g = 1 TO 100: NEXT g
760 FLASH 0
770 LET b = 0
780 GO TO 520
```

Experiment 16a

Design and write a video game which includes sound, user-defined characters, plenty of motion and excitement.

<i>Experiment 16a completed.</i>	
----------------------------------	--

Afterword

Congratulations! If you have worked steadily through the book and have done all the experiments, you will have an excellent grasp of the principles of programming. You'll be able to design and write a huge range of interesting and useful programs.

There is, of course, much more to programming than we have been able to cover in Part 1. You should try to expand your knowledge by reading as widely as possible. If you have enjoyed the book, we recommend **ADVANCED BASIC** for the SPECTRUM, which uses the same methods to cover such important topics as subroutines, string handling, arrays, graphics and program structure. You should also read other books about the SPECTRUM and about its microprocessor, the Z80. In particular, reference books - which are written for the knowledgeable programmer - should now begin to make sense to you. Other sources of interesting information are the computer magazines, of which there are at least 30 titles in Great Britain. You should look both at the ones dedicated to the SPECTRUM and the general ones which have articles about all kinds of micro-computers.

Finally, if you can possibly afford it, we suggest you get a printer for your SPECTRUM. We found one extremely useful in writing the book, and we are sure that you will find it equally helpful for designing accurate and effective programs.

Appendix A

Expressions

Number Theory

Algebraic Functions

Evaluating Series

Trigonometric Functions

This appendix explains the mathematical functions of the SPECTRUM and describes some of its possible applications in Mathematics and Science. You may not find the Appendix easy to follow: it all depends on how much mathematics you know. Read as much as you like, and stop when the going gets too hard. If your maths is rudimentary or even non-existent, don't worry! You only need the SPECTRUM's mathematical facilities to solve certain kinds of problem, and there are plenty of other things to do with a computer.

A.1. Expressions

The way expressions are written in BASIC is designed to be as close as possible to the conventional mathematical notation. The differences are mainly due to the fact that you can't write superscripts (such as powers) or fractions on a computer keyboard. So, instead of e^x you have to put EXP(x); instead of a^b , $a \uparrow b$; and instead of $\frac{x}{y}$, x/y.

In BASIC, (or at least that part of the language covered in this book) mathematical expressions are written with four types of element:

numbers (such as 3 , 4.7 or PI)

variables (like j , x or re)

the arithmetic operators, including +, -, *, / and \uparrow

The brackets (and)

As far as possible, expressions are worked out in exactly the same way as in ordinary algebra. Division (/) always gives an answer which is as accurate as possible and may have a decimal part (so that, for example, 8/7 gives 1.142857). The sign \uparrow means 'raised to the power of', so that $5 \uparrow 2$ is a way of writing 'five squared'. The power needn't be a whole number, since the machine does the calculation by logs. You should note that the \uparrow operator won't accept a negative number as its first operand, even though the power is a small whole number. For example,

$(-2) \uparrow 2$

does not give the value 4 (as it should) but generates the fault message

A Invalid argument

This is due to a software bug - that is, an error in the manufacturer's software. It may well be fixed in later versions of the SPECTRUM.

When an expression has several operators, the order they are used depends on the 'rules of precedence', which are as follows:

Highest precedence : \uparrow

Next : - (when used at the beginning of an expression or after a left bracket)

Next : * and /

Lowest precedence : + and -

To show how this works, consider the expression

$$a + b * c - d$$

The operator with the highest precedence in this expression is the $*$, so the product $b * c$ is worked out first. $+$ and $-$ have the same precedence, so the next operator to be used is the $+$, because it is on the left. The result at this stage is $a + b * c$. Finally, the $-$ is used and the d is subtracted.

Brackets change the rules of precedence entirely. The rule is that anything enclosed in brackets is worked out first, notwithstanding the ordinary precedence of the operators involved. For example, in

$$(a + b) * (c - d)$$

the $+$ and the $-$ are used before the $*$, because they are enclosed in brackets and the $*$ is not. A further rule is that a $-$ sign immediately following a left bracket is always taken as if it were at the beginning of an expression and given a high precedence, even inside the brackets.

Brackets are cheap, and should be used whenever you are in any doubt about the meaning of your expression. A common source of error is the coding of an expression like

$$\frac{x + y}{2 * a}$$

which should be written either as

$$(x + y) / (2 * a)$$

or as

$$(x + y) / 2 / a$$

The form $(x + y) / 2 * a$ (which people often write) means

$$\frac{(x + y) * a}{2}$$

Finally, it is worth saying that if an expression seems to be growing so complicated that it is getting out of hand, try splitting it up. For example:

$$\text{LET } a = 2 * (q \uparrow 4 - p \uparrow 4) + 5 / (q \uparrow 4 - p \uparrow 4) + 10 * (q \uparrow 4 - p \uparrow 4) \uparrow 2$$

could become

$$\begin{aligned} \text{LET } j &= q \uparrow 4 - p \uparrow 4 \\ \text{LET } a &= 2 * j + 5 / j + 10 * j \uparrow 2 \end{aligned}$$

A.2. Number Theory

The SPECTRUM has several functions which do simple but useful things with numbers. In the definitions which follow, x always stands for a number or expression:

INT (x)

holds for negative numbers, so that $\text{INT}(-3.5)$ is -4 .

$\text{SGN}(x)$

gives you 1 if x is positive, 0 if $x=0$, and -1 if x is negative.

$\text{ABS}(x)$

produces x if x is positive, or $-x$ if x is negative. For example, $\text{ABS}(5.5)$ is 5.5, and $\text{ABS}(-4.6)$ is 4.6.

$\text{SQR}(x)$ (here x is a number or expression which must be positive or zero)

gives the square root of x . Thus $\text{SQR}(9)$ is 3, and $\text{SQR}(5)$ is 2.236068 (which is correct to six places of decimals). If you ask the machine to find the square root of a negative number, it will object and come up with the message 'Invalid Argument'.

Lots of elementary number theory (such as the theory of Prime Numbers) is about exact divisors - whether one number can divide another number exactly, without leaving any remainder. Here is a BASIC expression which gives the remainder when you divide a by b (both being whole numbers or 'integers'):

$a - \text{INT}(a / b) * b$

On the SPECTRUM this expression will give correct result for numbers up to 1000000000 or so.

To give you a very simple example of the way this expression can be used, here is a program which inputs a number n from the user and tells him whether it is a Prime Number. You'll remember that a Prime Number is one which has no exact divisors except for itself and 1. The program works by trying all the numbers between 2 and $(n-1)$, and reporting that n is Prime only if none of these numbers is an exact divisor.

```
10 REM Test for Primality
20 INPUT "What is your number? ";n
30 FOR j = 2 TO n-1
40 IF n - INT(n / j) * j = 0 THEN GO TO 80
50 NEXT j
60 PRINT n;" is a Prime Number"
70 GO TO 90
80 PRINT n;" is not Prime. A factor is ";j
90 STOP
```

Of course this program is in its simplest form, and is extremely slow and inefficient. As an exercise, try speeding it up by:

- Using only odd trial divisors (apart from 2).
- Using trial divisors only up to the square root of the number being tested. (Why? Because if a number has a factor greater than its square root, it must also have one smaller).

as a prime). For example:

$$\begin{aligned}2 &= 1 + 1 \\4 &= 2 + 2 = 3 + 1 \\6 &= 3 + 3 = 5 + 1 \\8 &= 5 + 3 = 7 + 1 \\10 &= 7 + 3 = 5 + 5 \\12 &= 11 + 1 = 7 + 5 \\14 &= 13 + 1 = 11 + 3 = 7 + 7 \\16 &= 13 + 3 = 11 + 5 \\18 &= 17 + 1 = 13 + 5 = 11 + 7\end{aligned}$$

This table shows that the conjecture is true for even numbers up to 18. Your program should examine further even numbers, and try to find one which cannot be expressed as the sum of two primes. If there is a number which doesn't obey the conjecture (and nobody knows if this is so) it is bound to be quite large since most of the smaller numbers have already been tested. Write your program so that it can start the search at any number, and can tell you how far it has got when you stop it. Once you get the program running, leave it to churn away over nights, weekends, holidays, and at any other time the machine isn't doing anything else. You may (just) be the first to disprove Goldbach's conjecture!

A.3. Algebraic Functions

The SPECTRUM offers you a group of functions which are normally called algebraic. The functions work on numbers, which can be written as expressions, and include the following:

LOG (x)

gives you the natural logarithm of x (that is, the log of x to the base e). x must be positive or the machine will report a fault.

EXP (x)

gives you e raised to the power of x. Note that $\text{EXP}(\text{LOG}(x)) = \text{LOG}(\text{EXP}(x)) = x$.

EXP and LOG find considerable use in electrical calculations, and in those dealing with nuclear energy. For example, if the 'half-life' of a radio-active source is h years, the time it takes to decay to a fraction, f, of its original strength is given by the formula

$$h * \text{LOG}(f) / \text{LOG}(0.5)$$

Thus, if the half-life of a waste product is 37 years, the time it will take for the radio-activity to drop to 1% of its original value will be

$$37 * \text{LOG}(0.01) / \text{LOG}(0.5) \text{ or } 246 \text{ years.}$$

A useful way of solving certain types of equation is Newton's method. It is assumed that the equation, in one unknown called x, can be written as

$$f(x) = 0$$

The method depends on guessing an approximate answer, and then improving the guess, over and over again, until the equation is satisfied as accurately as need be. It can be proved that if the initial guess x is near enough the correct value, a better guess x^1 is given by the formula

$$x^1 = x - f(x)/f'(x)$$

where $f'(x)$ is the derivative of $f(x)$ with respect to x .

Let's take the example just quoted. $f'(x)$ will be

$$f'(x) = e^x - 9x^2 + 8x + 7$$

Here is a program to solve the equation:

```

10 REM Solve exp(x)-3x↑3+4x↑2+7x-5=0
20 INPUT "Initial guess? ";x
30 LET f = EXP(x)-3*x*x*x+4*x*x+7*x-5
35 REM ↑ would be unsafe because x may be < 0
40 PRINT "x = ";x;" f(x) = ";f
50 IF ABS(f) < 0.000001 THEN GO TO 90
60 LET fd = EXP(x) - 9*x*x + 8*x + 7
70 LET x = x - f/fd
80 GO TO 30
90 PRINT "Finally. x = ";x
100 STOP

```

The program asks the user for the initial guess, and stops when $f(x)$ is within 0.000001 of zero. The ABS function is used to ensure that a band either side of zero is used: an alternative, but more clumsy way of achieving the same would be

```

50 IF (x < 0.000001) AND (x > -0.000001)
THEN GO TO 90

```

Key this program in, and run it with different starting values. See how it behaves, and how quickly it converges to an answer. How many answers can you find? Are they valid?

A.4. Evaluating Series

Mathematicians, physicists and engineers often need to work out series of the general form

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots + a_nx^n$$

where the a 's are 'coefficients' taken from a set of published tables and x is a value specific to the current problem. The most obvious way to code the evaluation in BASIC is to use the \uparrow operator for the powers of x : for example, a series with four terms could be written as

```

LET y = 2.3412 + 0.9542 * x + 0.4213 * x↑2 + 0.1034 *
x↑3 + 0.0023 * x↑4

```

If this statement is used for different values of x , the program will run very slowly, and will not work at all if x is negative. This is because

You can increase the speed of the program perhaps 100 times and avoid the problem with negative values of x by rewriting the expression so that it only uses multiplication and addition. Our example would become:

```
LET y = (((0.0023 * x + 0.1034) * x + 0.4213) * x  
+ 0.9542) * x + 2.3412
```

where the changes necessary are clear.

A.5. Trigonometric Functions

The SPECTRUM gives you a set of 'trig' functions : SIN, COS, and TAN. The use of these functions is quite straightforward provided you remember that the arguments (or numbers supplied to the functions) must be in radians, not degrees. One radian is $180/\pi$ or 57.295779 degrees.

Here is a simple program which plots the position of a ship sailing various distances and courses one after the other. The ship is presumed to start at the 'origin' of a 1-km grid, and the total distance moved is taken to be too small to be affected by the curvature of the earth. The program inputs each new course and the distance travelled, and displays the ship's coordinates at the end of that 'leg' of the journey. The course is entered as a bearing East of North, and can range between 0 and 360 degrees.

```
10 REM Ship's Course  
20 REM e is Easting: n is Northing.  
30 REM c is current course, and d is distance in km.  
40 LET e = 0  
50 LET n = 0  
60 PRINT "Current coordinates are ";e;" , ";n  
70 INPUT "Course ?";c  
80 INPUT "Distance on this course ?";d  
90 LET n = n + d * SIN (c * PI / 180)  
100 LET e = e + d * COS (c * PI / 180)  
110 GO TO 60
```

The reciprocal functions cotangent, secant and cosecant are not provided but can easily be synthesised by the relationships

$$\begin{aligned}\cot(x) &= 1/\tan(x) \\ \sec(x) &= 1/\cos(x) \\ \operatorname{cosec}(x) &= 1/\sin(x)\end{aligned}$$

The SPECTRUM does provide the inverse trigonometric functions ARCSIN, ARCCOS and ARCTAN (they use the names ASN, ACS and ATN respectively). As you would expect, the results of these functions are delivered in radians.

As a final illustration we'll give you a program to find the Great Circle Distance (the distance over the earth's surface) between any two places in the world. You have to supply the latitude and longitude of each of the places, which you can easily look up in an atlas. To avoid messing about with degrees, minutes, seconds and such symbols as 'N', 'W', etc., the program uses two simple conventions:

The program gives its result to the nearest mile. It uses the formula from spherical trigonometry that gives the angle a subtended at the centre of the sphere by an arc bounded by two points, in terms of their latitudes $la1$ and $la2$, and their difference in longitude d . The formula is:

$$a = \arccos(\sin(la1) * \sin(la2) + \cos(la1) * \cos(la2) * \cos(d))$$

When this angle has been calculated, the Great Circle Distance is simply the product of the angle and the radius of the earth.

The program assumes that the earth is a perfect sphere, and therefore gives slight errors (up to 10 miles) over long distances. It is not accurate enough for pigeon racers, who need distances correct to 10 metres, but it will serve most other purposes.

The program is provided on the tape under the title "gcd". Here is its text in full for you to study:

```

10 REM Great Circle Distance
20 CLS
30 PRINT
40 PRINT "The program gives you the Great"
50 PRINT "Circle Distance between any two"
60 PRINT "places on the Earth."
70 PRINT "You will be asked for the"
80 PRINT "Latitude and Longitude of each"
90 PRINT "place in Degrees. Use minus"
100 PRINT "values for places SOUTH of"
110 PRINT "the Equator or EAST of Green-"
120 PRINT "wich. For example, the position"
130 PRINT "of GENEVA (45 degrees 12' N,"
140 PRINT "6 degrees 7' E ) would be given"
150 PRINT "as : 45.2, -6.11667 and SYDNEY"
160 PRINT "would be (roughly) at -32, -141."
170 LET r=3980:REM Mean radius of the earth
180 INPUT "Name of first place ? ";a$
190 INPUT "Lat. and Long. of first place ? ";la1,lo1
200 INPUT "Name of second place ? ";b$
210 INPUT "Lat. and Long. of second place ? ";la2,lo2
220 LET d=ABS ((lo1-lo2)*PI/180): REM difference of
    longitude in radians
230 LET la1=la1*PI/180: REM convert latitudes to radians
240 LET la2=la2*PI/180
250 LET a=ACS (SIN (la1)*SIN (la2)+COS (la1)*COS
    (la2)*COS(d))
260 PRINT
270 PRINT "The Great Circle Distance"
280 PRINT "between ";a$
290 PRINT "and ";b$
300 PRINT
310 PRINT "is ";INT (a*r+0.5);" miles"
320 PRINT
330 PRINT "Hit any key to run the program"
340 PRINT "again."
350 LET x$=INKEY$
360 IF x$ = "" THEN GO TO 350

```

Appendix B

Sample Solutions to the Experiments

Experiment 6a

a\$ <> "B"	T
c <> j	T
a\$ < "ANDY"	F
a\$ > "CHRIS"	F
j = 10	F
a\$ + "TOPHER" <> "CHRISTOPHER"	F
4 > c	F
c + j <= 13	T
c + 5 = j	T

Experiment 6b

	Control variable	Starting value	Final value	Increment	No of times round loop
A.	a\$	"X"	"XQQQQ"	"Q"	5
B.	x	0	10	+1	11
C.	z\$	"A"	"ABCBCBCBC"	"BC"	5
D.	m	3	19	2	9
E.	p	13	3	-2	6

Experiment 6c

- a. 10 LET a\$ = "+"
20 PRINT a\$
30 LET a\$ = a\$ + "+"
40 IF a\$ <= "++++++++++" THEN GO TO 20
50 STOP
- b. 10 PRINT "£", "LIRE"
20 PRINT
30 LET li = 10
40 PRINT li, 2320 * li
50 LET li = li + 1
60 IF li <= 30 THEN GO TO 40
70 STOP

Experiment 7a

A.

PROGRAM	DISPLAY
<pre> 10 LET a = 2 20 LET b = 5 30 LET c = a + b 40 LET d = b - a 50 PRINT a, b, c, d 60 STOP </pre>	<pre> 2 5 7 3 </pre>
PROGRAM POINTER: 10 20 30 40 50 60 VARIABLES: a: 2 b: 5 c: 7 d: 3	

B.

PROGRAM	DISPLAY
<pre> 10 LET s = 1 20 PRINT "Tinker" 30 PRINT "Tailor" 40 PRINT "Soldier" 50 PRINT "Sailor" 60 LET s = s + 1 70 IF s < 3 THEN GO TO 40 80 STOP </pre>	<pre> Tinker Tailor Soldier Sailor Soldier Sailor </pre>
PROGRAM POINTER: 10 20 30 40 50 60 70 40 50 60 70 80 VARIABLES: s = 1 2 3	

Experiment 7b

Line 70 should be:

```
70 IF c < 31 THEN GO TO 40
```

Line 30 should be:

```
30 LET a$ = a$ + "*"
```

Experiment 7c

There are two mistakes in the program. The correct version is

```
10 PRINT "Nine times table"  
20 LET multiplicand = 1  
30 PRINT multiplicand ; " times 9 = "; multiplicand * 9  
40 LET multiplicand = multiplicand + 1  
50 IF multiplicand <= 12 THEN GO TO 30  
60 STOP
```

If you can't find the second mistake, here is a coded hint:

```
DIFDL UIF TQFMMJOH PG UIF MPOHFTU XPSE
```


Experiment 8a

```

10 INPUT " Type any number ";m
20 LET j = 1
30 PRINT j;" times "; m; " = "; j * m
40 LET j = j + 1
50 IF j <= 12 THEN GO TO 30
60 STOP

```

Experiment 8b

```

10 LET t = 0
20 INPUT "How many pounds of apples? "; a
30 LET t = t + 37 * a
40 INPUT "How many pounds of grapes? "; g
50 LET t = t + 54 * g
60 PRINT "Total to pay is £"; t/100
70 STOP

```

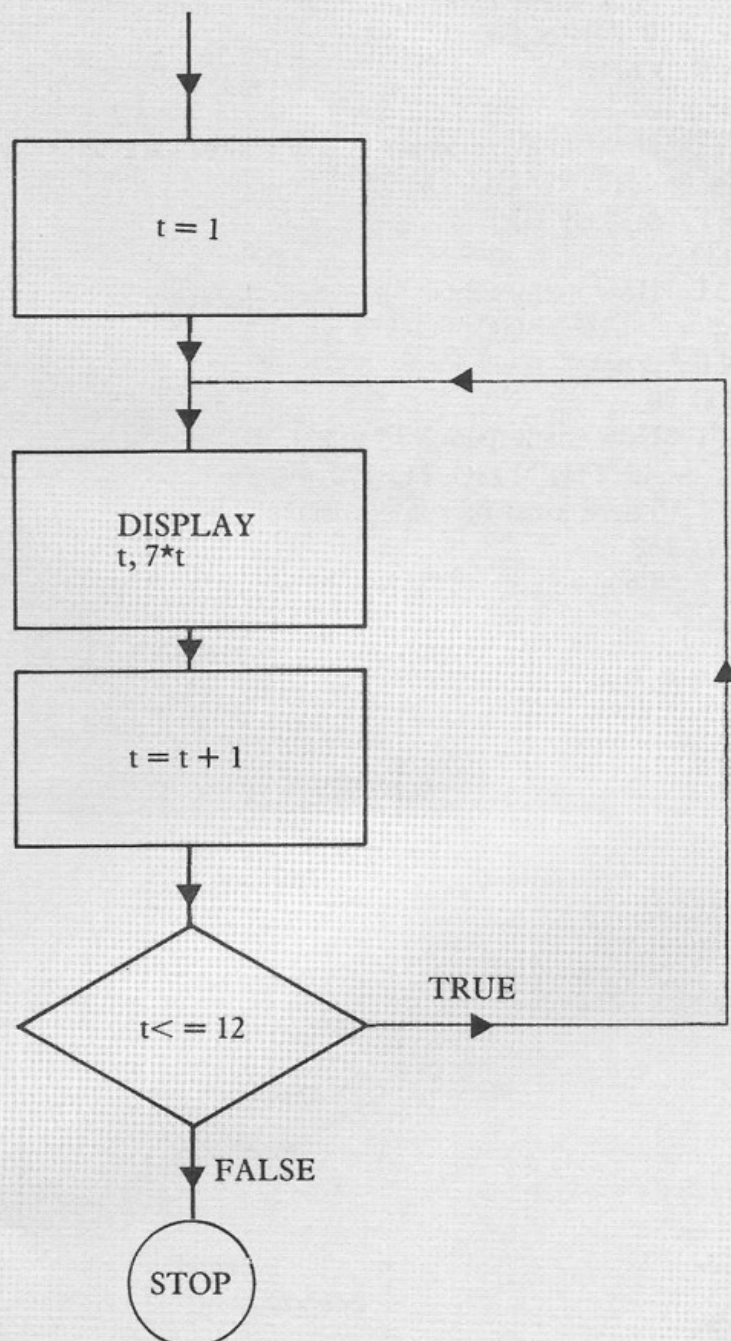
Experiment 8c

```

10 INPUT "What is her name? ";w$
20 INPUT "How old is she? "; wa
30 INPUT "What is his name? "; h$
40 INPUT "How old is he? ";ha
50 IF ha <= wa THEN GO TO 80
60 PRINT h$; " is "; ha - wa; " years older than "; w$
70 STOP
80 IF ha < wa THEN GO TO 110
90 PRINT w$; " and "; h$; " are the same age."
100 STOP
110 PRINT w$; " is "; wa - ha; " years older than "; h$

```

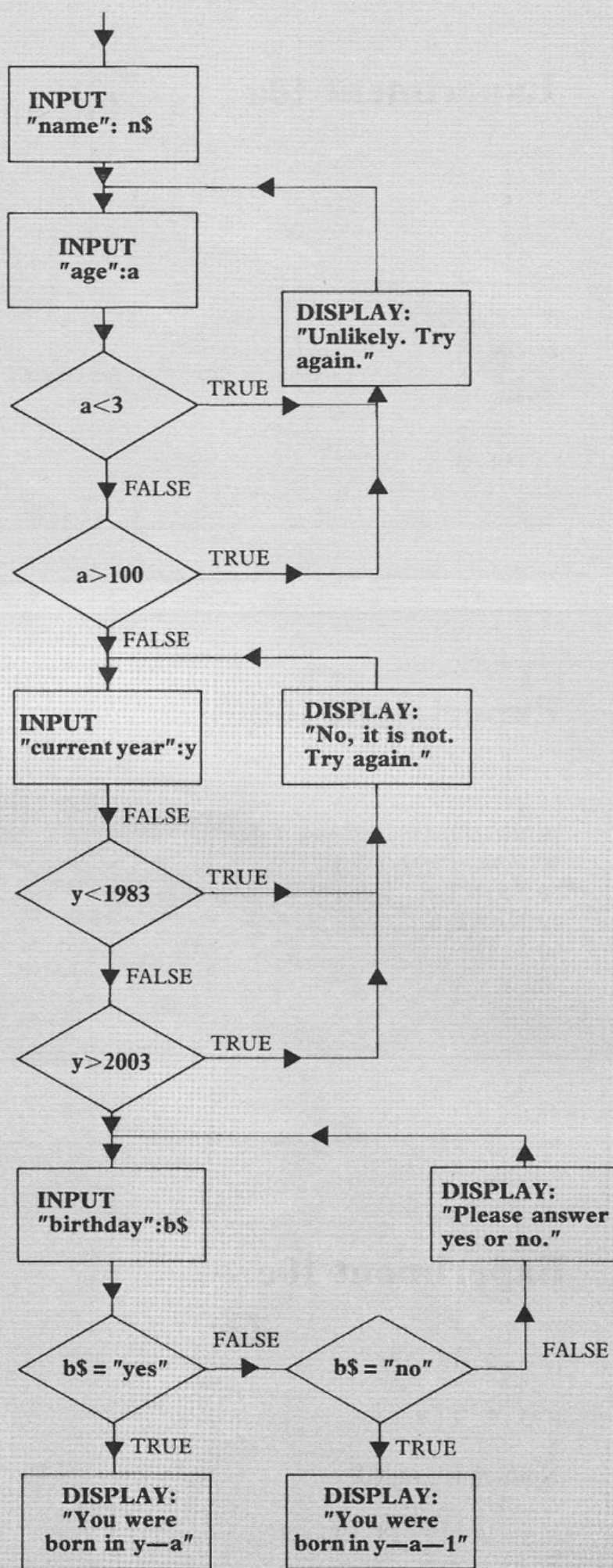
Experiment 9a



Experiment 9b

```
10 INPUT "How many cars? ";c
20 IF c <= 5 THEN GO TO 50
30 PRINT "Eh?"
40 GO TO 10
50 INPUT "How many caravans and/or motor caravans? ";v
60 IF v <= 4 THEN GO TO 90
70 PRINT "A likely tale!"
80 GO TO 50
90 INPUT "How many tents? ";t
100 IF t <= 5 THEN GO TO 130
110 PRINT "What?"
120 GO TO 90
130 INPUT "How many people? ";p
140 IF p <= 40 THEN GO TO 170
150 PRINT "There must be some mistake!"
160 GO TO 130
170 PRINT "Total to pay = ";5*c+25*v+10*t+3*p;
    " Francs"
180 STOP
```

Experiment 9c



Experiment 10a

```
1: 2
    5
    8
    11
    14
```

```
2: 24 8
    26 6
    28 4
    30 2
    32 0
```

Experiment 10b

```
10 PRINT "Five Times Table"
20 FOR n = 1 TO 12
30 PRINT n;" times 5 = ";5*n
40 NEXT n
50 STOP
```

Experiment 10c

```
1 4 9 16 25
```

```
4 3 2 1 0
```

(Nothing is printed)

```
2 6 10 14 18 22
```


Experiment 10d

```
10 LET rm = 0
20 INPUT "How many judges? ";j
30 FOR p = 1 TO j
40 INPUT "Marks ";m
50 LET rm=rm+m
60 NEXT p
70 PRINT "The average mark is ";rm/j
80 STOP
(Note : lines 10 and 20 may be interchanged.)
```

Experiment 10e

```
10 INPUT "First name? ";f$
20 FOR k=2 TO 10
30 INPUT "Next name? ";n$
40 IF f$ < n$ THEN GO TO 60
50 LET f$ = n$
60 NEXT k
70 PRINT "The top name is ";f$
80 STOP
```

Experiment 10f

```
10 INPUT "Give a number (up to 12) ";w
20 FOR t=1 TO w
30 FOR j=1 TO t
40 PRINT j;" ";
50 NEXT j
60 PRINT
70 NEXT t
80 STOP
```

Experiment 12a

Mozart lived for 35 years
Bach lived for 65 years
Richard Strauss lived for 85 years
Sibelius lived for 92 years

Experiment 12b

First problem - alter DATA statements to read

```
DATA 6 : REM number of trains
DATA "0804","1116"
DATA "1015","1515"
DATA "1255","1600"
DATA "1710","2039"
DATA "1823","2130"
DATA "1937","2346"
```

Second problem: the modified program is

```
10 REM General timetable program
20 READ o$,p$: REM read origin and destination
30 PRINT "When can you leave ";o$
40 INPUT t$: REM t$ is time passenger wants to leave
50 PRINT
60 PRINT t$
70 PRINT
80 READ n: REM n is number of trains
90 READ d$,a$: REM d$ and a$ are departure and arrival
   times
100 IF t$ > d$ THEN GO TO 150
110 PRINT "Your best train from ";o$
120 PRINT "leaves at ";d$;" and arrives"
130 PRINT "at ";p$;" at ";a$;". "
140 GO TO 180
150 LET n = n-1
160 IF n > 0 THEN GO TO 90
170 PRINT "No more trains today"
180 INPUT "More information ? ";x$
190 CLS
200 RESTORE
210 GO TO 10
220 DATA "Florence","Perugia"
225 DATA 14
230 DATA "0311","0609"
240 DATA "0436","0704"
250 DATA "0531","0807"
260 DATA "0700","0928"
270 DATA "0802","1040"
280 DATA "0948","1212"
290 DATA "1140","1408"
300 DATA "1330","1614"
310 DATA "1454","1742"
320 DATA "1643","1914"
330 DATA "1728","2029"
340 DATA "1850","2117"
350 DATA "1950","2205"
360 DATA "2042","2336"
```


Experiment 12c

```

100 DATA 5
110 DATA "Haggis",102
120 DATA "Caviare (Beluga)",6500
130 DATA "Quails' eggs",2550
140 DATA "Salsify",40
150 DATA "Truffles",1540
200 READ n: REM number of items
210 LET t=0:REM total to pay
220 READ c$,p: REM item and price
230 PRINT "How many pounds of ";c$; " would you like?"
240 INPUT q
250 PRINT q;" lbs. please."
260 LET t=t+p*q
270 LET n=n-1
280 IF n > 0 THEN GO TO 220
290 PRINT "Your total bill is ";t/100
300 STOP

```

Experiment 13a

```

10 REM rising tune
20 FOR p=-10 TO 20
25 LET d=4+p/6
30 READ du,pt
40 IF du=0 THEN GO TO 70
50 BEEP du/d,p+pt
60 GO TO 30
70 RESTORE
80 NEXT p
90 STOP
100 REM Col. Bogey
110 DATA 1,7,4,4,1,4,1,5,1,7,2,16,2,16,4,12
120 DATA 1,7,4,4,1,4,1,5,1,4,2,7,2,7,4,5
130 DATA 1,5,4,2,1,9,1,11,1,9,1,12,4,7
140 DATA 1,7,1,5,1,4,1,2,2,9,1,0,1,-1,2,7,1,7,4,0
150 DATA 0,0

```

Experiment 14b

1. To make the ball move slower, put a null loop into the program.
You might put

```
125 FOR z=1 TO 100  
126 NEXT z
```

2. To reduce the size of the billiard table, alter the numbers 0, 1 and 31 in commands 90 to 120.

3. To make the ball leave a trail of X's, alter line 130 to read

```
130 PRINT CHR$ 8;"X"
```

Experiments 14c, 14d and 14e

No standard solutions are included, since your answers will probably be better and more impressive than ours!

No sample solutions are given for the experiments in Chapters 15 and 16, since these problems need a good deal of creative design work.

Appendix C

Words and Symbols

Table of the words and symbols used in SPECTRUM BASIC

SYMBOL	KEYWORD MODE	SYMBOL SHIFT MODE	EXTENDED MODE	EXTENDED SYMBOL SHIFT MODE
ACS		Y		W
AND				Q
ASN		I		E
AT				L
ATN				Z
ATTR				
BEEP			B	B
BIN				9
BORDER	B			H
BRIGHT				5
CAT			U	
CHR\$				
CIRCLE				
CLEAR	X			
CLOSE #				
CLS	V		I	
CODE				
CONTINUE	C			
COPY	Z		W	
COS			D	
DATA				1
DEF FN				
DIM	D			
DRAW	W			7
ERASE			X	
EXP				V
FLASH				2
FN				0
FOR	F			
FORMAT				
GO SUB	H			
GO TO	G			
IF	U			I
IN				X
INK			N	
INKEY\$				
INPUT	I		R	
INT				M
INVERSE				
LEN			K	
LET	L			3
LINE				
LIST	K			
LLIST			V	
LN			Z	

SYMBOL	KEYWORD MODE	SYMBOL SHIFT MODE	EXTENDED MODE	EXTENDED SYMBOL SHIFT MODE
MERGE MOVE				T 6
NEW NEXT NOT	A N	S		
OPEN # OR OUT OVER		U		4 O N C
PAPER PAUSE PEEK PI PLOT POINT POKE PRINT	M Q O P		O M	8
RANDOM- ISE READ REM RESTORE RETURN RND RUN	T E Y R		A S T	
SAVE SCREEN\$ SGN SIN SQR STEP STOP STR\$	S	D A	F Q H Y P E	K
TAB TAN THEN TO		G F		
USR			L	
VAL VAL\$ VERIFY			J R	J
! " # \$ %		1 P 3 4 5		

Appendix D

Error Reports

This appendix describes the error reports you are likely to get when you run BASIC programs. This is not the complete list of error messages which the SPECTRUM can give you under all possible circumstances, but the ones we have omitted could only arise through using commands and facilities which we have not yet described.

Whenever the computer stops obeying a BASIC program, it gives you a code, a brief explanation, and the place where the error occurred. The place is given as a line number and a command number within the line. The codes are as follows:

CODE MESSAGE		MEANING
0	OK	Your program has ended. The last instruction has been obeyed.
1	NEXT without FOR	Your FOR and NEXT commands don't match up properly.
2	Variable not found	A command refers to a variable which has not yet been given a value
4	Out of memory	Your program is so long that it won't fit into the computer. This error is extremely unlikely at this point in the course.
6	Number too big	A calculation has led to a number larger than 10^{38} . The most likely reason is division by zero, but other causes can be the incorrect use of the mathematical functions, such as evaluating $\tan(\pi/2)$.
9	STOP statement.	The computer has come to a STOP statement. This is not a fault!
A	Invalid Argument	The number supplied to a mathematical function is not valid. For example, you cannot ask for the square root of a negative number or the log of a number of zero or less.
B	Integer out of range	A number used in the command is not within the acceptable limits. This could happen (for example) with a GO TO followed by a very large number.
E	Out of DATA	You have tried to read past the end of the DATA statements.

CODE MESSAGE		MEANING
H	STOP in INPUT	Data typed by the user began with the word STOP.
I	FOR without NEXT	Your FOR and NEXT commands don't match up.
K	Invalid colour	The colour number is out of the correct range.
L	BREAK into program	BREAK has been pressed.
R	Tape loading error	An error has been found when loading or verifying a program.

Index

Algebraic functions	A.5
Altering programs	5.2 - 5.5
Arithmetic operators	3.9
AT	14.4, 14.5
BASIC	3.2, 6.2, 7.10
BEEP command	13.2 - 13.5
BIN	14.13
BORDER	15.2 - 15.3, 15.6
BREAK SPACE	2.3, 2.4, 4.4, 4.10, 6.5
CAPS LOCK mode	11.7, 11.9
CAPS SHIFT	1.4, 2.3, 2.3, 4.4, 4.10, 11.6, 14.9
Cassette recorder	1.2, 1.6, 5.6
Cassette tape	1.5 - 1.7, 5.6, 5.7, 5.9
CHR\$	14.4, 14.7
CLS	1.6, 4.2, 9.9, 14.2, 14.3, 15.3
Colour	1.3, 1.7, 15.2, 15.3, 15.5, 15.6
Comma	3.4, 8.4
Concatenation	3.9
Conditions	6.2, 6.5, 6.11, 9.4
CONT	2.4
Control variable	6.7, 6.9, 10.5
Copyright	5.9, 12.3
Correcting typing mistakes	1.4, 2.3, 4.3, 5.2 - 5.5
Cursor	1.4, 4.2, 5.3, 5.4, 8.4
Cursor control key	5.3
DATA statements	12.3, 12.6, 12.7 - 12.10, 12.13, 14.13
DELETE	1.4, 2.3, 3.2, 5.4
Division	3.5
EDIT	5.4, 5.5
ENTER	1.5, 3.2, 4.2, 4.3, 8.3, 8.5
Error report	1.5, 3.4, 3.5, 7.2, 8.4, Appendix D
Evaluating series	A.6
Expression	3.4, 3.5, 6.4, A.2, A.3
Extended mode	11.5, 11.9
Extended (Symbol Shift) mode	11.5, 11.9
FLASH	15.7
Flexible programs	8.2
Flow charts	9.2
FOR command	10.2, 10.5, 10.7, 10.8
Game	16.5
GO TO command	4.2, 4.3 - 4.5, 4.8, 6.5, 7.3, 7.10, 9.4
Grammatical errors	7.2
Graphics symbols	14.9
Graphics mode	11.8, 11.9, 14.9
IF command	6.2, 6.5
Increment	6.7, 6.9, 10.2
INK	5.5, 15.2, 15.3, 15.6
INKEY\$	16.2 - 16.4
INPUT command	8.2 - 8.5
Kemeny and Kurtz	3.2
Keyboard	2.2, 2.3, 11.1, 11.2
Keyword	3.2
Keyword mode	11.4, 11.9
Labelled command	4.2, 4.3
Label numbers	4.2

LOAD command	1.5 - 1.7
Loop	4.7, 4.8, 6.7, 6.9, 7.6, 7.10, 10.2, 10.5
Loop body	6.7, 6.9, 10.2
Lower case	2.3
Lower case mode	11.6, 11.9
Machine breakdown	7.10
Memory	4.2, 7.3
Modes	11.3 - 11.8
Multiplication	3.5
Music	13.2, 15.5
NEW command	4.2, 4.4, 5.9
Nested loop	10.11
NEXT command	10.2
Null loops	10.11
Null string	16.2
Number theory	A.3, A.4
Numbers	3.3
Numeric variable	3.7, 3.8, 6.2, 8.4
PAPER	5.5, 15.2, 15.3
Pitch	13.2, 13.4
PRINT command	1.4, 3.2 - 3.5, 3.7, 3.10, 4.2, 6.6, 6.7 6.9, 7.3
Print position	14.2, 14.3
Program cursor	5.3, 7.3
Program design	6.9, 7.2, 9.12
Program glossary	9.11
Program pointer	7.3
Program robustness	9.6
Program tracing	7.3 - 7.10
Prompts	8.3
Pyramid	10.7
Quote symbols	1.6, 3.4, 8.4
Random (RND) function	15.4, 15.5, 16.7
Reaction time	16.2
READ command	12.3, 12.6, 12.10
Relationships	6.2, 6.3, 6.11
REM statements	12.2, 12.3
RESTORE command	12.10, 12.12
RUN command	4.5, 5.10
SAVE command	5.6, 5.9, 5.10
Screen control	14.2
Scroll	4.4, 6.5
Security	5.9, 5.10
Semicolon	3.4, 4.6, 8.4, 14.7
Spaces	3.4
Stored command	4.2, 4.3
String	3.4, 4.8, 4.9
String variable	3.7, 3.8, 6.7, 8.4
STOP command	6.6, 7.3, 8.5, 9.4
SYMBOL SHIFT	1.4, 2.3, 6.6
Symbol Shift mode	11.4, 11.9
TAB	14.4
Transylvania	9.1, 9.6, 14.10, 14.11
THEN	6.2, 6.5, 14.7
Trigonometric functions	A.7
Tuning TV	1.3
TV set	1.2, 14.2
User (of a program)	8.5, 9.6
User-defined graphics	14.12, 15.5
USR	14.13
Variables	3.6 - 3.9
VERIFY command	5.7

Postscript

We hope you have enjoyed "Learn BASIC programming on the Sinclair Spectrum". If you have followed the course thoroughly you will, by now, have learnt to write useful programs, but we should make clear that the course does **not** cover the whole of Spectrum BASIC. The second part of this course: "Advanced BASIC Programming on the Sinclair Spectrum" covers in detail the more advanced and difficult features.

Logic 3 is constantly striving to maintain its reputation as a supplier of top quality computer programs. If you have any suggestions or ideas regarding our products we would be delighted to hear from you. Please write to us at the address below.

