

Easy Programming for the **ZX SPECTRUM**

Ian Stewart and
Robin Jones

SHIVA's
friendly
micro
series



Easy Programming for the ZX SPECTRUM

Ian Stewart

Mathematics Institute, University of Warwick

Robin Jones

Computer Unit, South Kent College of Technology



Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
4 Church Lane, Nantwich, Cheshire CW5 5RQ, England

©Ian Stewart and Robin Jones, 1982

Reprinted 1983

ISBN 0 906812 23 2

Cover photograph of the ZX Spectrum kindly supplied by Sinclair Research Ltd.

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

This book is sold subject to the Standard Conditions of Sale of Net Books and may not be resold in the UK below the net price given by the Publishers in their current price list.

Typeset and printed by Devon Print Group, Exeter

Contents

Preface	1
1 Basic BASIC (How to talk to your Spectrum)	3
2 Arithmetic in BASIC (+, -, *, /, , LET, INPUT, PRINT)	8
3 The Keyboard (shifts, modes, how to type what you want)	13
4 Heeeeellllppp!	15
5 Input/Output (INPUT "Message", PRINT AT, TAB)	16
6 Looping (FOR, NEXT)	18
7 Debugging I (Types of error)	22
8 Random Numbers (RND)	26
9 Branching (IF, THEN, GO TO, AND, OR, NOT)	27
10 Plotting (PLOT, DRAW, CIRCLE, OVER, SCREEN\$)	31
11 Debugging II (Dry running)	39
12 Graphics (PRINT AT, INKEY\$, PAUSE)	44
13 User-defined Characters (POKE USR, BIN, cats)	49
14 Subroutines (GO SUB, RETURN)	52
15 Son et Lumière (INK, PAPER, BORDER, BLEEP, FLASH)	56
16 Debugging III (Tracing)	63
17 Strings (\$, CODE, VAL, LEN, TO)	70
18 Data (DATA, READ, RESTORE)	74
19 Debugging IV (Dormant bugs)	77
20 Curve Plotting (For the mathematically minded)	79
21 Debugging V (Round-off errors)	85
22 Programming Style (and an anti-missile game)	87
23 Peek and Poke (PEEK, POKE)	93
24 Tips (Snags and tricks)	98
25 What I haven't told you about	101
26 What Next?	102
Prepacked Programs (Copy them and RUN)	103

Preface

Clive Sinclair has done it again, and so have we.

Not content with selling half a million ZX81 microcomputers, he's now come up with the ZX Spectrum. Not only can you now get home computing at a reasonable price: you can have sound, colour, and high-resolution graphics. (And an acceptable keyboard!)

When the ZX81 came out, we wrote an introductory book for it, called *PEEK, POKE, BYTE, AND RAM!*—an obscure title foisted on us by our publicity man. We thought of calling this one *SON OF PEEK POKE* but that seemed even more obscure. But, in a sense, that's what it is. It's the Spectrum version of *PEEK POKE*.

The Spectrum has several features in common with the ZX81, and where possible we've treated them the same way. So, if you bought the first book and are upgrading, you'll recognize about a quarter to a third of it. But, even there, we've rewritten to take advantage of the Spectrum's superior features. The rest is new, but written in a similar style.

The aim is straightforward: it is to describe, in simple and comprehensible terms, those features of the ZX Spectrum that a first-time user should know about. In addition, we've included about 30 "prepacked" program listings, to be copied and run as you wish; and another 30 or so programs in the text. The advantage that we have over the *Manual* is that we can be selective, and concentrate on the less-esoteric features: otherwise, the trees tend to obscure the wood.

So what do you get? You get a gentle guide to BASIC programming, programming style, colour, sound, moving graphics, high-resolution (fancy) graphics, debugging, number-crunching, and string-handling. When you've read this, the *Manual* will be easy meat. Everything is broken down into manageable chunks, so that you'll be able to spend a couple of hours with the Spectrum and come away having achieved something definite. *Plus* a variety of programs for your delectation, which make good use of the Spectrum's remarkable range of facilities; plus "De Bugs" to amuse you when attention starts to flag.

Lots of computer books give the impression they're trying to prove how clever their authors are. We're not doing that: everything here is very straightforward and basic. We think that's what an *introduction* to the world of the computer ought to look like. The hardest part is getting off the ground.

So far, we've referred to ourselves collectively as "we", but this won't work very well later. Personal experiences relayed by a "we" take on a curious air. Since any given section was written by only one of us, we have plumped for the word "I" to describe us both. If that disturbs you, consider how often a singular author refers to himself as "we"!

ROMs, RAMs, and REMs . . . Computer jargon isn't really confusing. It just looks like it is.

I Basic BASIC

There is nothing fundamentally mysterious about computers. They're just machines which will carry out series of instructions. Of course the actual *way* they carry them out—now that is truly mysterious, at least if you don't have a degree in solid state physics. But, just as you can drive a car without being able to build one, so you can program a computer without knowing how to design one. *Some* understanding of the actual mechanics of the thing—what computer people call the *hardware*—is useful though, just as it helps to know how a clutch works when you're learning how to change gear, or indeed wondering why it is necessary to change gear at all. But the main principles of programming don't depend much on the hardware, as far as the ordinary user is concerned.

Machines that carry out instructions have been with us for a long time now. Blaise Pascal, a French mathematician-cum-philosopher, built a calculating machine in 1642. Charles Babbage, an Englishman, designed an ambitious "Analytical engine" in 1835, and the British government was persuaded to invest in it. Like other government projects, it turned out not to be feasible with the technology of the time; but the underlying ideas were original and sound. The Frenchman Joseph-Marie Jacquard, at the beginning of the nineteenth century, developed a system using holes punched in cards to control the weave produced by a loom. Fairground steam-organs use a similar principle. Indeed you can think of a trained orchestra as a machine to produce music, with the written sheet music acting as a "program". It's not a bad analogy.

It's no good walking up to a Jacquard loom, saying "seven yards of blue and green check tweed, please," and expecting much useful to happen. It doesn't talk that language. All it understands is a pattern of punched holes; and there are all sorts of restrictions on the size, number, and arrangement of those holes. "Auld Lang Syne" on a steam-organ won't run on a loom—and even if it did, it would produce a result that bore little resemblance to the original tune. You have to know *which* pattern of holes is required to produce the desired pattern of threads in the cloth. So the pattern of holes is a kind of *coded* version of the pattern of threads, with the mechanism of the loom acting as a go-between.

Similarly the written music used by an orchestra is a coded version of the intended musical sounds, with the orchestra itself as an intermediary. (There is a minor difference: orchestras are not as precise as a machine, and a conductor has some freedom to interpret the music. But it's close.)

It is the same with computers, only more so. A given machine must be talked to in a language that it understands.

In the early days of computing, this meant that you would have to type out long lists looking something like this: 01000111010011010111111100010110001010011 . . . which is kind of hard on the eye. Computers still, "deep down", *think* this way (with "0" meaning "no electrical current" and "1" meaning "some electrical current"); this rather rudimentary language is the *machine code* for the chosen type of computer. Miraculously, it lets you do everything you want to; and it generally has the advantage of being *fast* to run . . . but it's less than straightforward to write programs in it.

Fortunately, these days you don't have to.

This is because a computer can be *taught* to accept commands in a language other than its “native tongue”. What actually happens is that some diligent programmer works out a kind of “dictionary” to translate the new language into machine code, and feeds it into the computer (either from some outside source, or built into the hardware). The dictionary is known as a *compiler*, or *interpreter* (depending on just how it works).

The Spectrum has a built-in interpreter for a language known as BASIC, which stands for “Beginners’ All-purpose Symbolic Instruction Code”. This language was developed in the USA in the mid-1960s and is very widely used. It has the advantage of being relatively straightforward, like a cross between ordinary English and school algebra. As far as the beginning programmer is concerned, the Spectrum speaks only BASIC. (But Z80 machine language is accessible via the *USR* key—and if you don’t know *why* you might want to use it, you *don’t*. Not yet, anyway.)

A BASIC PROGRAM

Rather than starting with the “grammar” of BASIC, let’s take a look at a simple BASIC program and see what it does and how it does it. The advantage of this is that you’ll see right away how easy it is. Grammatical fine points will come later.

```
10 PRINT "Doubling"  
20 INPUT x  
30 LET y = x + x  
40 PRINT x, y  
50 STOP
```

You can probably guess what this does, just by looking at it. But first, there are some points to notice about the form that it takes.

- (a) It consists of a series of *lines*.
- (b) Each line is a “legal” (that is, logically sensible) BASIC *instruction* or *command*, or *statement*. (These three words all mean the same thing in practice.)
- (c) Each line starts with a number, known (not surprisingly) as the *line number*. (Computers often use 0 for zero to distinguish from the letter O.)

To these rather obvious remarks I must add some clarifications. Some of these apply to *all* BASIC interpreters, as used on other machines; some only to the Spectrum. To save breath I’m not going to worry which is which: if you graduate to a more fancy machine you’ll soon find out.

There’s nothing worth saying about (a) except that on the Spectrum a program line can be more than one line long when written on the TV screen—the machine doesn’t mind.

Details for (b) depend on the “grammar” of BASIC, to which we shall come.

The reason for numbering lines (c) is that, early in your programming career, you are going to want to tell the machine to obey specific lines in a program. An instruction “GO TO 730” will tell it to do whatever it says on line 730 . . . but it needs to know which line that is. There are advantages in *not* just counting “1, 2, 3, . . .”, as we shall see. The line number must be a whole number between 1 and 9999 inclusive.

When the machine works its way through a program, it moves from a given line to the next one (except when told to go elsewhere, as part of the program). So it needs those line numbers, even if the programmer himself doesn’t want to refer to them. The Spectrum will not accept program lines without a number (although it may instead carry out the command for that line, as if it were a pocket calculator). *Don’t forget the line numbers!*

WHAT DOES IT DO?

Type the above program into your Spectrum. I'm going to explain the full intricacies of the keyboard a little later: you'll certainly have noticed that each key has an awful lot of writing on it, and in fact some keys can produce eight different effects, depending on what "mode" and "shift" they are in! But you won't need to know the gory truth all at once.

I'm going to assume you've just switched the machine on, and got the "© 1982 Sinclair Research Ltd." message. (If not, type NEW, or pull out the power supply plug for a second.)

Let's take line 10 as an example. Hit key 1 (top row, on the left) and then 0 (top row, on the right). You'll see these numbers come up at the bottom of the screen, with a flashing "K", called the *cursor*. Next, hit key P: the entire word "PRINT" appears. This is a very clever feature of the Spectrum, and it saves a lot of time: whole BASIC words can be typed in using a single key. (You will have noticed that the word "PRINT" is written on key P too.)


Now for the quotation mark ". That's on key P too! But in *red*. That means you must hold the SYMBOL SHIFT key down while pressing key P, to get the quotes. Now release the SYMBOL SHIFT. To get a capital D, hold down the CAPS SHIFT key while pressing key D. Now let go of the CAPS SHIFT and hit, in turn, keys O, U, B, L, I, N, G. These will be printed in lower case letters—"oubling". Now the quotes again: SYMBOL SHIFT and P as before.

Line 10 is now sitting at the bottom of the screen. (The cursor has become a flashing "L", too.) To put it into the program memory, press the key marked ENTER. Up it goes!

Line 20 works the same way: press 2, 0, I (which has "INPUT" written on it), and X: then ENTER. Lines 30–50 are similar: but note that you get the "=" sign by holding down SYMBOL SHIFT while pressing L; the "+" sign using SYMBOL SHIFT and K; the comma by SYMBOL SHIFT and N; and STOP is SYMBOL SHIFT plus A. Don't forget to ENTER once a line is typed out correctly.

If you've followed these instructions, you'll have the whole program listed at the top of the screen. And it will sit there forever unless you do something to prevent it. This is because computers not only obey instructions, but they are fundamentally as thick as two short planks: even when it's obvious what you want them to do, you have to tell them anyway. So press key R (which comes out as RUN) and then ENTER so that it knows you've finished the command.

What you see on the screen is something like RUN  with a flashing L. This disappears when you hit ENTER.

Across the screen appears the message "Doubling". This is the computer's response to line 10: PRINT "Doubling". It's done this pretty quickly, and now it's waiting for *you* to do *your* bit of line 20: INPUT x. To remind you, there's an  at lower left of the screen. It wants you to tell it what x is.

To do this, type out a number, followed by ENTER. Try

```
2      (ENTER)
```

Quick as a flash, the computer types out

```
2      4
```

(and, in the lower corner, a message "9 STOP Statement, 50: 1" which we won't worry about—it just means it's finished the job correctly.)

Try it again. To do this, just type RUN followed by ENTER and it's off again. When it asks for x, try something else: if you try 756.2912 and hit ENTER you should get

```
756.2912      1512.5824
```

RUN it a few more times, trying different INPUTs x. Try some at random. Try 0, 1, 2, 3, 4.

This ought to convince you that whatever you input as x , the machine PRINTs out two numbers: first x , and then the double of x .

Experiment. You should get things like:

17	34
21	42
-5	-10
6	12

HOW DOES IT DO IT?

I don't mean in detail—that would get us into the hardware, machine language, and such complexities. But how should you, the programmer, think of the machine, as it runs your program? What's going on inside its tiny silicon-chip brain? Indulging in a little anthropomorphism, it's something like this.

```
10 PRINT "Doubling"
20 INPUT x
30 LET y = x + x
40 PRINT x, y
50 STOP
```

RUN ENTER

Doubling



2 ENTER

2 4

9 STOP Statement, 50: 1

Well, gosh, here come some instructions for me to remember. Better do it.

I wonder if he's finished yet?

Yep. Off we go. What's the first line? Get it from memory: 10 PRINT "Doubling". I've got to PRINT something. There's a quotation mark "; then I copy out what follows . . .

until I reach another". Which tells me to stop PRINTing.

Nothing else. On to next line: 20 INPUT x. He's going to tell me a number, and I've got to call it x . Give him an . . .

cursor to remind him.

OK, I'm waiting.

Dead slow, these humans.

Ah, x is 2. Now what?

30 LET $y = x + x$. That means I must add x to x , that is, work out $2 + 2$. And call the result y . So y is $2 + 2 = 4$. Next instruction? 40 PRINT x, y . I have to PRINT x and y , which are . . .

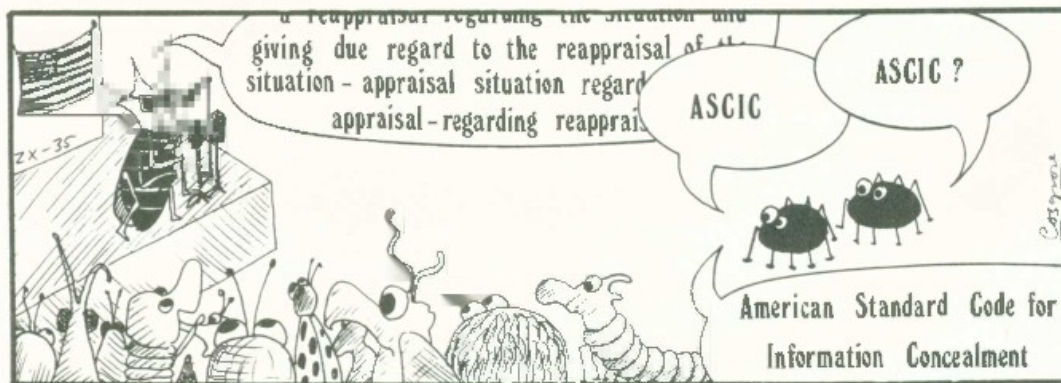
Next line? 50 STOP. That's the lot, then. Finish with a report . . .

Ignore the whimsy: do you see how the machine is just running through the list of commands, and obeying them as they come? It doesn't even "know" what the program is about. But you, the programmer, know—it doubles numbers.

The task of the programmer is now clear. Given a job that you want the machine to do, you must assemble a series of instructions which, when carried out, achieve this objective.

To do this, you'll need to know BASIC in more detail. Your Spectrum will do some pretty clever things if you know how to talk to it. There are all sorts of refinements, of course—a *good* program should be quick, efficient, and clear, as well as getting the job done. But refinements can wait: the main thing is to write programs that *work*.

So let's get started.



Addition, subtraction, multiplication, and division form the basis of all mathematics. The same goes for computing. So the sensible place to start is by explaining how to do:

2 Arithmetic in BASIC

Actually that's a lie—it's algebra. But I didn't want to scare you off. Algebraic calculations are involved in almost any program you care to write, even if only for book-keeping purposes, so they're a reasonable place to start.

Just like ordinary algebra, BASIC uses letters to stand for "general" numbers. In the jargon, these are called *variables*—indeed, *numeric variables*—but really all this means is that they are things like x , y , z , a , b , c and so on which can be used to build up algebraic expressions like $x + y - z$, which the computer can then work out if you tell it, via the program, what values x , y , and z take. (For instance if $x = 14$, $y = 3$, $z = 9$ then $x + y - z = 14 + 3 - 9 = 8$.) On the Spectrum you can use more complicated symbols for variables than just single letters, but long names waste memory.

There are some small but important differences between BASIC algebra and ordinary algebra. The signs $+$ (plus), $-$ (minus), and $/$ (divided by) are just as usual. (You can't use \div .) But multiplication is written as an asterisk $*$, so that $5 * 7$ means 5×7 , which is 35. The up-arrow \uparrow means "raised to the power". For instance, $2 \uparrow 3$ means what an algebraist would write as 2^3 , which is 2 to the power of 3 (or *2 cubed*) and has value $2 \times 2 \times 2 = 8$.

You can combine numbers, variables, and these arithmetical signs to produce more complicated expressions. For instance

$$a * x \uparrow 2 + b * x + c$$

is the algebraist's favourite $ax^2 + bx + c$.

Chapter 3 of the Introductory booklet explains about these, and it makes a very important point about the order in which the machine makes the calculations. Suppose, in the above expression, the machine knows that $a = 4$, $x = 5$, $b = 3$, $c = 7$. You might think that it will work out the sum as follows:

$$a * x = 4 * 5 = 20.$$

$$a * x \uparrow 2 = 20 \uparrow 2 = 400,$$

$$a * x \uparrow 2 + b = 400 + b = 403,$$

$$a * x \uparrow 2 + b * x = 403 * x = 2015,$$

$$a * x \uparrow 2 + b * x + c = 2015 + c = 2022.$$

That's what you get if you work along the expression from left to right. But this will not correspond to the algebraist's $ax^2 + bx + c$, which would take the value

$$4 \times 5^2 + 3 \times 5 + 7 = 100 + 15 + 7 = 122.$$

So what's gone wrong?

The point is that ordinary algebra is full of rules, about which operations you do first. (Anyone remember a thing called BODMAS? Never mind . . .) And the language BASIC has similar rules. In fact it works out all \uparrow 's *first*, then all $*$'s and $/$'s, and finally the $+$'s and $-$'s. If in doubt, it does each of these in order from left to right. So it actually works out $a * x \uparrow 2 + b * x + c$ this way:

First do the \uparrow 's: $x \uparrow 2 = 5 \uparrow 2 = 25$.

Now the $*$'s: $a * x \uparrow 2 = 4 * 25 = 100$
 $b * x = 3 * 5 = 15$.

Now the $+$'s: $a * x \uparrow 2 + b * x = 100 + 15 = 115$
 $a * x \uparrow 2 + b * x + c = 115 + c = 115 + 7 = 122$.

You can see how this resembles ordinary algebra.

Just as in algebra, you sometimes want to work out an expression which does *not* follow these rules naturally. The solution is the same: you use *brackets* (). Any expression inside brackets is worked out as a whole first. So

$(a * x) \uparrow 2$

would be worked out as $(4 * 5) \uparrow 2$, which is $(20) \uparrow 2$ if you do the bracketed bit first, and this is 400 .

There is only one kind of bracket (not like $|$ or $\{$ in algebra). The other brackets are on the keyboard but can't be used in expressions. So you have to be especially careful when putting brackets inside other brackets, writing things like $(3 + 5 * (a - b)) \uparrow 4$ where an algebraist would write $[3 + 5(a - b)]^4$.

The main thing to understand is that it really is just like ordinary algebra with slightly unfamiliar symbols.

ASSIGNING A VALUE TO A VARIABLE

This is done by the LET statement (key L). If you use it where it ought to be used, the clever Spectrum automatically recognizes it as LET and not L (by a process known as "automatic syntax-checking" which just means it keeps an eye on what makes sense as you type it in). A program line such as

```
40 LET x = 5
```

tells the computer that the variable x takes the value 5 *from now on, and until it is told otherwise by some other part of the program*. Try this program.

```
10 LET a = 4
```

```
20 LET b = 3
```

```
30 LET c = 7
```

```
40 LET x = 5
```

```
50 PRINT a * x  $\uparrow$  2 + b * x + c
```

If all is well in that diminutive skull, you will see the answer 122 popping up at the top of the screen.

There are easier ways to get this particular answer: try

```
10 PRINT 4 * 5  $\uparrow$  2 + 3 * 5 + 7
```

(You can even omit the line number: see Chapter 3 of the Introductory booklet. But we want you to see how LET works.)

The right-hand side of a LET command can be an expression that the computer already knows how to work out. For instance, you could change line 50 of the above program to define a new variable y, taking the value $a * x^2 + b * x + c$:

```
50 LET y = a * x ↑ 2 + b * x + c
```

Follow this by

```
60 PRINT y
```

and you can check it works properly.

Example: Falling bodies

According to Galileo, a body falling from rest under gravity will drop a distance of (approximately) $16t^2$ feet in a time of t seconds. To find out how far it will drop in 17 seconds, you could use the program

```
10 LET t = 17
```

```
20 PRINT 16 * t ↑ 2
```

You should get 4624 (feet) as output.

If you want the answer for a different time, you can change the first line of the program. Try it. How far will the body fall in:

- (a) 97 seconds?
- (b) 3 seconds?
- (c) 24.5779 seconds?
- (d) ~~10000~~1 seconds?

AN IMPROVEMENT: THE INPUT STATEMENT

You will probably have got fed up with repeatedly changing that line 10. A more civilized program would make use of the INPUT command, which lets the machine ask you what value a variable takes. Type out

```
10 INPUT t
```

```
20 PRINT 16 * t ↑ 2
```

and RUN it. The L cursor comes up: this is the machine waiting to be told what t is. Type in 17, then ENTER; you'll get the answer we had at first. But now, to answer (a) to (d), all you have to do is type RUN again, but tell it the new value of t. Try it.

For more on INPUT, see Input/Output, Chapter 5.

PROGRAM PROJECTS

Here are three programs for you to write, using just the commands we've looked at so far. They are minor variations on the program for falling bodies. If you get stuck, there's a crib at the end of this chapter!

- (e) The volume of a cube whose side is x is given by x^3 (that is, x^3 in BASIC if x denotes the side). Write a program which INPUTs x and PRINTs the volume of the corresponding cube.
- (f) A bucket is attached to one end of a rope, which is coiled around a windlass shaped like a cylinder of radius r. For this particular bucket, according to the mechanics textbooks, its downward acceleration is given by

$$a = \frac{32}{1 + 3r^2}$$

Don't worry about the mechanics; but write a program which lets you INPUT r and PRINT the acceleration a .

- (g) At the Zedex Spectramarket, honey costs 61p a jar; Zxo cubes 25p a packet; Whizxas Supermeat 32½p a tin. Write a program which PRINTs the total cost of h jars of honey, z packets of Zxo cubes, and w tins of Whizxas Supermeat, when you input h , z and w . [With what I've told you so far, you will need three INPUT commands.]

A FORETASTE OF LOOPING

The commands FOR and NEXT allow us to try out some of the previous ideas without doing too much work. (In many circles the avoidance of work is called "laziness", but programmers have an advantage over ordinary mortals—they can call it "efficient programming" and point to savings in computer memory or time to prove it.) There's a lot that can be said about FOR/NEXT loops, and in a few pages we'll say it at suitable length. Run this program first, to see the kind of thing that can be done.

```
10 FOR x = 1 TO 20
20 PRINT x, x * x
30 NEXT x
```

You will find a list of the numbers from 1 to 20, and a corresponding list of their squares 1, 4, 9, . . . 361, 400. What the FOR/NEXT commands do is to send the machine round and round the series of commands in between (here just line 20) setting the variable x successively at 1, 2, 3, . . . until 20 is reached. Line 10 sets these limits up and starts the loop going; line 30 sends the computer through the loop again.

By changing line 20 in this little program, you can tabulate all sorts of things. You want cubes? Try

```
20 PRINT x, x ↑ 3
```

Change line 20 to each of the following, and notice the differences. The very minor changes from one to the next cause the machine to work out the algebra in different orders, with differing results. What are the programs calculating in ordinary algebraic symbolism?

- (h) 20 PRINT x, $1/x + 1$
- (i) 20 PRINT x, $1 / (x + 1)$
- (j) 20 PRINT x, $1/1 + x$
- (k) 20 PRINT x, $1 / (1 + x)$
- (l) 20 PRINT x, $1 + 1/x$
- (m) 20 PRINT x, $(1 + 1) / x$

Finally, notice that in these programs the values of the variable are not assigned by the command LET. The machine obtains the assignment instead from the FOR command.

ANSWERS

Falling bodies

- (a) 150544 (b) 144 (c) 9665.1707
- (d) 1600032000000. Don't be deceived by the apparent precision. The computer won't give 12 significant-digit accuracy. It has rounded off the true answer, which is 1600032000016.

Program projects

- (e) 10 INPUT x
20 PRINT x ↑ 3
- (f) 10 INPUT r
20 PRINT 32 / (1 + 3 * r ↑ 2)
- (g) 10 INPUT h
20 INPUT z
30 INPUT w
40 PRINT 61 * h + 25 * z + 32.5 * w

Your program doesn't have to look *exactly* like these to be "correct". In particular, the choice of symbols for variables is entirely up to you. You can do the algebra other ways, too: in (e) there's nothing wrong with 20 PRINT x * x * x, and in (f) with 20 PRINT 32 / (1 + 3 * r * r).

Looping

- (h) $\frac{1}{x} + 1$
- (i) $\frac{1}{x + 1}$
- (j) 1 + x [it works out 1/1 first,
which is 1, then adds on x]
- (k) the same as (i)
- (l) the same as (h)
- (m) $\frac{2}{x}$



Each key on the Spectrum can produce several different effects. Here's a quick guide to:

3 The Keyboard

As I said earlier, the Spectrum keyboard is quite sophisticated, and the result of pressing a key depends on the context in a fairly complicated way. You'll soon get used to it; but you'll need to spend some time learning to find your way around the keyboard.

The two things that affect the result of pressing a key are the *shift* that is being used, and the *mode* that the computer is in.

SHIFTS

There are two keys marked CAPS SHIFT and SYMBOL SHIFT. If one of these is held down, and a key is pressed while it remains held down, then the effect of that key is altered. Exactly *how* depends on the mode, so let's take a look at those:

MODES

The mode is the "internal state" of the computer, and is signalled by the cursor. There are five modes, with the following (flashing) cursors:

- K Keyword mode
- L Letter mode
- C Capitals mode
- E Extended mode
- G Graphics mode

Here's how to get into them:

1. Ordinarily the machine is in "L" mode.
2. After a line number, or at the start of a directly entered command, it automatically goes into "K" mode.
3. To get to "C" mode from "L" mode, press CAPS SHIFT and hold it down. To get back into "L" mode, release it. (You'll only see the "C" if you use CAPS LOCK—ignore this at first.)
4. To get to "E" mode from "L" mode, press *both* CAPS SHIFT and SYMBOL SHIFT. To get back to "L" mode, repeat this.
5. To get into "G" mode from "L" mode, press CAPS SHIFT and key 9 (GRAPHICS) together. To get back into "L" mode, repeat this.

HOW TO PRINT WHAT

Let's look at a typical key in the lower three rows, say key "R". It looks like this:



The INT is in green; the < and VERIFY in red.

1. To type "r" press the key when in "L" mode.
2. To type "R" press CAPS SHIFT and go into "C" mode; hold it down and press the key.
3. To type "RUN" press the key while in "K" mode.
4. To type "<" get into "L" mode; hold down SYMBOL SHIFT and hit the key.
5. To type "INT" get into extended mode "E" by hitting both shifts together. *Look for the "E" cursor*; if you hold the keys down too long it can flip back to "L". Then release the shifts and press the key.
6. To type "VERIFY" go into "E" mode as in (5), then *hold down the symbol shift* while hitting the key.
7. To obtain the user-defined graphics character corresponding to "R" (see the chapter on user-defined characters) go into "G" mode and then hit the key.

THE TOP ROW OF KEYS

These are a bit different. They do not have any keywords ("K" mode) written on them; instead, there is a graphics character shaped like a box. Over the top there isn't anything in green; but there are commands in white. The main symbol on the key and the two red symbols work just as in rows 2-4. The other items go like this:

1. To get the white symbol over the top, hold down CAPS SHIFT. These are control characters, and are not actually *printed*.
2. To get the graphics symbol, go into "G" mode and hit the key.
3. To get the graphics symbol in inverse video (interchange black and white, or more generally INK and PAPER) go into "G" mode, hold down CAPS SHIFT, and hit the key.
4. The colours are mnemonics only: for instance, the code for GREEN is 4.
5. You can actually do a bit more than this: see the *Manual*, Chapter 16. And I haven't mentioned what CAPS LOCK, TRUE VIDEO, and INVERSE VIDEO do. Rather than add to the confusion, since you won't actually *need* these keys, I'll leave you to experiment for yourself.

PREPACKED PROGRAMS

At the end of this book you will find a number of program listings, starting on page 103. *You should feel free at any time to copy out one of these listings and run it*, whether or not you understand the commands contained in the program. All you do is type out the listing on the keyboard, check carefully to make sure there are no mistakes, and press RUN followed by ENTER. We hope that by the end of the book you will be able to work out how these programs do their job; but to gain confidence early on it's a good idea to try running other people's programs. It also gives you some idea of what the Spectrum can be made to do.




The programs are generally accompanied by Program Notes, which explain various oddities, and occasionally suggest modifications to the programs, or projects for you to try in related areas. But they will run, subject to avoiding errors, whether these notes make any sense to you or not. Most of the programs illustrate particular techniques explained in this book, but some use ideas I don't have space to explain properly. In any case, I assume you are reading the *Sinclair Manual* as well as this one.

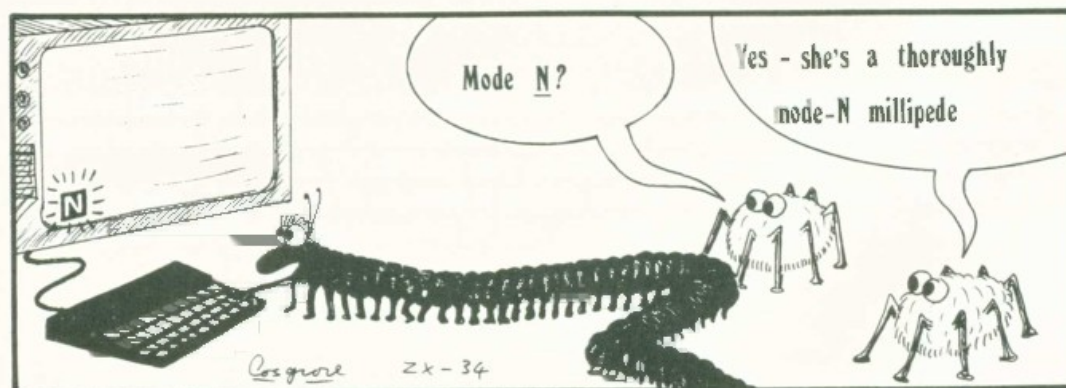
Not going as well as you hoped?
Maybe you can use some

4 Heeeeeeellppp!

If something doesn't quite go right, what should you do?

At the very worst, you unplug the power supply for a second. This crashes the program, wipes out everything you've written in—but unjams whatever the trouble was (unless in the hardware, in which case hard luck). But there are usually better ways.

- (a) If you print the wrong thing in a program line: Use the arrowed keys 5 and 8 (note you need CAPS SHIFT) to move the  cursor just past the offending beast, obliterate it using DELETE (key 0 with CAPS SHIFT).
- (b) If you want to change a line that's already gone into the program:
To delete it altogether, write its line number and press ENTER.
Otherwise, move the > cursor up and down the lines using 6 and 7: then press EDIT to bring the line down to where it can work on it. Now proceed as in (a).
Various tricks: It's an awful pain to move the cursor from line 10 to line 530 one line at a time. Find a line number just before 530 that you haven't used, say 529. Write 529 ENTER; then press EDIT. You'll see line 530 come down. (Why?)
- (c) If the program gets stuck, and nothing seems to be happening:
This is often the case during a debugging session! the last thing you want is to wipe everything out and type it all in again; and in any case it's likely to do exactly the same again.
- (c1) If the program is still executing, but you want to stop it: Press CAPS SHIFT and BREAK.
- (c2) If it's stuck on a numerical INPUT instruction: BREAK won't work; but STOP will. If by mistake you've got other junk sitting there, it will keep giving you a  cursor for a syntax error, which is vastly frustrating; DELETE the junk, then STOP.
- (c3) If it's stuck on a character input—  cursor: press DELETE and *then* STOP.
- (d) If the program runs but doesn't do the right things: Read the DEBUGGING chapters.



There are lots of helpful ways to get information into or out of the computer, and to produce well-organized displays:

5 Input/Output

There's much more to the INPUT command than I've said. For a start, you can INPUT several numbers in one go. A better way to answer program project (g) in Chapter 2 is:

```
10 INPUT h, z, w
20 PRINT 61 * h + 25 * z + 32.5 * w
```

When you RUN this, you will find that after *each* of the three inputs you will need to press ENTER. The numbers are temporarily printed on the bottom row (or rows) until all three are in.

The commas that separate them also control the places where the numbers are printed: they go in columns 0 and 16 alternately. If you change the commas to semicolons ";" you will find that the inputs are printed one after the other, with no spaces between them. To get spaces, you must define them in the INPUT command:

```
10 INPUT h; "□"; z; "□"; w
```

where a box □ denotes a space.

You can also print out *prompts* on an input: messages that remind you what input is required. To do this, you include the message inside quotes as part of the INPUT statement. For instance, change line 10 above to:

```
10 INPUT "honey", h
12 INPUT "zxo cubes", z
14 INPUT "whizxas", w
```

You can make more complicated combinations of such commands as part of a single INPUT statement, but this gives the main idea.

OUTPUT

So far, when we've PRINTed output data (like numbers) we've left it up to the computer to decide *where* to print. Now that's not always convenient, and it won't always produce pretty displays. To change the print position, you use PRINT AT.

For PRINT purposes, the TV screen display is thought of as being divided up into 22 rows numbered 0–21 from top to bottom; and 32 columns numbered 0–31 from left to

right. There's a diagram in the GRAPHICS chapter later, illustrating this; but for the moment here's a program which will let you experiment.

```
10 INPUT "row", r
20 INPUT "column", c
30 PRINT AT r, c; "£"
40 GO TO 10
```

This will literally print money! Try various values for the row and column numbers, and see where the pounds are printed.

To select a specific PRINT position, you of course specify *r* and *c* as numbers in the command. To get a message on the bottom row, starting five spaces in (which is column 4, because the numbers start at 0!), write

```
10 PRINT AT 21, 4; "Message"
```

To get roughly in the middle of the screen:

```
10 PRINT AT 10, 12; "Message"
```

And so on.

If you don't use the AT instruction, the computer will just move on automatically to the "next" position. Where this is depends on what's just been printed. If the last PRINT command does not end in ";" or ",", then the machine moves on to the next row. A ";" moves it on to the next space in the *same* row. A "," moves it to the next available slot in column 0 or 16, whichever is free first.

The colon and the comma have similar effects in INPUT commands.

In conjunction with these automatic features, the TAB command is very useful if you are trying to write data in organized columns. For example, this program lets you set up a personalized Telephone Directory.

```
10 INPUT "name"; n$
20 INPUT "exchange"; e$
30 INPUT "number"; t
40 PRINT TAB 1; n$; TAB 15; e$; TAB 25; t
50 GO TO 10
```

(TAB is key P in extended mode. The dollar signs indicate *strings*, and are explained in the chapter about those, later on.) RUN this, and INPUT things like:

Fred	Timbuktu	44399
Monty	Preston	12345
Police	Norwich	999

on the prompted INPUTs. See how the data get arranged in three neat columns. (To stop the thing, press STOP on a number input, or DELETE and then STOP on a character string input.)

Project

Write a program to INPUT 22 numbers, and PRINT them out sloping diagonally from top left to bottom right (using something like PRINT AT *i*, *i*; *n*). Now think about top right to bottom left (PRINT AT *i*, 21 - *i*).

*There were 10 in the bed and the little one said
"Roll over, roll over!"
And they all rolled over and one fell out.
There were 9 in the bed and . . .*

6 Looping

By using the FOR and NEXT instructions, you can make the machine carry out an instruction over and over again a specified number of times. This may not sound very interesting, but in fact it is one of the most useful weapons in the programmer's armoury because you can use variables to alter what each step does.

We've already seen how FOR/NEXT loops allow us to print out tables of values of a function like x^3 . Here's a slightly less simple use.

The factorial function $n!$ is defined, as every schoolboy knows, to be:

$$n! = n(n-1)(n-2)(n-3) \dots 3 \times 2 \times 1.$$

It tells you the number of possible ways to arrange n objects in order. For instance, $3! = 3 \times 2 \times 1 = 6$; and the letters abc can be arranged in order 6 ways: abc, acb, bac, bca, cab, cba.

To compute $n!$ we can use loops. The idea is to compute it in stages: 1 ; 1×2 ; $1 \times 2 \times 3$; $1 \times 2 \times 3 \times 4$; . . . and to keep going until the largest number is n . In each case you take the result of the *previous* stage and multiply by the next number up. This is just the set-up needed for a loop. In fact, the sequence of statements

```
10 LET i = 1
20 FOR k = 2 TO n
30 LET i = i * k
40 NEXT k
```

has just this effect.

What does it do? And how? Line 10 just sets up a starting value for the variable i . Line 20 tells the machine to carry out succeeding lines over and over again, letting k successively take the values 2, 3, 4, 5, . . . , n . Line 40 tells it when it has reached the end of these instructions and should go back to the start of the loop again. So the first time through, it takes $k = 2$, and line 30 works out

$$i = i * k = 1 \times 2$$

Next time through, k has become 3 and i has become 1×2 ; so it works out $i * k$ which is now $1 \times 2 \times 3$. Next time, k is 4, and it works out $1 \times 2 \times 3 \times 4$. And so on, until the final stage, when k is n and it works out $i = 1 \times 2 \times 3 \times 4 \times \dots \times n$. Then, thanks to the limits set in line 20, it knows that the loop is finished.

This won't quite work out $n!$ for you, because the program doesn't include instructions for what n is, or to print the answer. So you need to embed the loop in a larger program:

```
5 INPUT n
10 LET i = 1
```



```

20 FOR k = 2 TO n
30 LET i = i * k
40 NEXT k
50 PRINT "Factorial □"; n; "□ is □"; i

```

(Line 50 is just fancy printing: you get outputs like

Factorial 6 is 720.

Notice the spaces, shown as boxes □, to make it look pretty.)

RABBITTING ON

In about 1220 a gentleman called Leonardo of Pisa, nicknamed Fibonacci (Son of Good Humour) came up with an interesting problem about rabbits.

If a breeding pair of rabbits produces one new pair of offspring once a month, and it takes one month for the new pairs to become productive, and you start with one pair—how does the population grow? (For simplicity we assume exactly regular production each month, and that every pair consists of one male and one female.)

It helps to make a table:

Month No.	No. of breeding pairs	No. of new pairs
In month 0 we have 1 breeding pair, 0 new pairs, so we get entry		
0	1	0
In month 1 we get 1 new pair only:		
1	1	1
In month 2 we get 1 new pair, and the previous new pair becomes productive:		
2	2	1
In month 3 all of these are productive, and we get 2 new pairs		
3	3	2
and so on		
4	5	3
5	8	5
6	13	8
7	21	13

We're starting to get an awful lot of rabbits, which isn't really surprising.

If we let the total number of pairs of rabbits in month m be $f(m)$, then we have $f(0) = 1$, $f(1) = 2$, $f(2) = 3$, $f(3) = 5$, $f(4) = 8$, and so on.

Now let's think more generally. Suppose in month m we have b breeding pairs and n new pairs. Then next month all of these become productive, giving $b + n$ breeding pairs; and the b breeders give us b new pairs. So the table has two consecutive lines looking like this:

Month No.	No. of breeding pairs	No. of new pairs
m	b	n
$m + 1$	$b + n$	b

This table suggests how we can calculate $f(m)$ by using loops. The idea is that to get from

month m to month $m + 1$ we have to turn the old b into $b + n$, and the old n into b . This will work:

```

10 LET b = 1
20 LET n = 0
30 INPUT m
40 FOR t = 1 TO m
50 LET c = b
60 LET b = b + n
70 LET n = c
80 NEXT t
90 PRINT "f("; m; ") is "; b + n

```

If you study this you'll see that it follows exactly the steps used to build up the table. Lines 10 and 20 set the initial values of b and n . Line 30 asks what value of m we want $f(m)$ for. Lines 40 to 80 perform a loop which generates successive lines of the table. Note line 50, which remembers the *old* b value, calling it c , for use in line 70. (If you didn't do this, line 60 would change b too soon, and line 70 would produce the wrong n value.)

The numbers $f(m)$ are called *Fibonacci numbers*. If you notice that the b and n columns in the table contain the same numbers, but shifted one line apart (why?) you'll see that $f(m + 2) = f(m + 1) + f(m)$; that is, any Fibonacci number is the sum of the previous two.

What is the value of $f(14)$? $f(77)$?

NESTED LOOPS

Even better—you can put loops-within-loops, or even loops-within-loops-within-loops (as far as the memory can hold). You'd be surprised how often this is necessary.

For instance, suppose you want to print out a table of values of the factorial function. You can use a loop like the one for $x \uparrow 3$ on page 11; but then you need *another* loop for the $n!$ calculation. So you get something like this:

```

5  FOR n = 1 TO 20
10  LET i = 1
20  FOR k = 2 TO n
30  LET i = i * k
40  NEXT k
50  PRINT n, i
60  NEXT n

```

} inner loop } outer loop

Notice that the "FOR n /NEXT n " loop is completely outside the "FOR k /NEXT k " one. You have to do this to make sense — it's just like brackets. An expression $[a + (b - 2c)] + d$ makes good sense; but $[a + (b - 2c)] + d$ does not. Try interchanging lines 40 and 60 and see what happens. Not much use, is it? The trouble is, the Spectrum will accept, and run, programs with wrongly nested loops; but of course the results are not what you intend. It does *not* print out an error message. Be careful!

STEP SIZE

If you just write `FOR i = 1 TO 20` the machine *assumes* you want the variable `i` to take values 1, 2, 3, 4, 5, . . . , 19, 20. That is, it assumes you move up in *steps* of size 1.

But you don't have to do this. You can set a different step size using the key `STEP`. For example, the instruction

```
10 FOR j = -3 TO 3 STEP .5
```

makes `j` run through the values -3, -2.5, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3. Similarly,

```
10 FOR j = 3 TO 4 STEP .01
```

runs through the values 3, 3.01, 3.02, 3.03, . . . going up in hundredths until it ends with 3.98, 3.99, 4.

You can also `STEP` downwards:

```
FOR j = 10 TO 0 STEP -1
```

makes `j` run through the sequence 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0. Reminds you of something? Run this program.

```
10 FOR j = 10 TO 0 STEP -1
```

```
20 PRINT j
```

```
30 NEXT j
```

```
40 PRINT "BLAST-OFF!"
```

Hardly sophisticated, but . . .

It is said that the colourful term “getting the bugs out” arose in the early days of computing, when insects used to crawl inside the machine and cause short circuits. Nowadays, if the computer goes wrong, it’s usually the programmer’s fault. But to put it right, you still need to know about . . .

7 Debugging I

It is a hard fact of life, which all programmers learn very rapidly, that programs hardly ever work—at least, the first time they run. The process of eliminating the errors from a program is known as *debugging*. It is important to adopt a systematic approach to this process because, even in a fairly small program, the source of error is not necessarily easy to find. And there are few things more frustrating than programs with elusive bugs in them.


Let’s look first at the kinds of error that can occur. Broadly, they split into two groups: *syntax error* and *runtime error*.

SYNTAX ERRORS

These are errors which the machine can identify as soon as you have typed them in. For example, suppose I were to type the line

```
FOR p = 1 - 7
```

under the misapprehension that the “-” symbol can be used as a synonym for “TO”. (This is, if you like, an error in the use of the grammar of the language, hence the term “syntax”.)

The Spectrum is particularly helpful to the beginner in this case. It will allow you to type the “-” symbol; but it will realize that there are no circumstances in which a complete program line can appear as `50 FOR p = 1 - 7`, and it will display a  prompt symbol (to tell you there is a syntax error), and refuse to accept the line until the offending “-” is replaced. (Many popular microcomputers will be perfectly happy to allow you to sprinkle mis-statements like this one throughout a program, and will only object when you try to run it. This doesn’t matter to an experienced programmer, but the beginner tends to make many such errors, and it saves a lot of time to be told of them immediately.)

At this point a question may be worrying some readers, which we could phrase thus: “If the Spectrum knows that the only thing which can appear after the “1” in statement `50` is the keyword “TO”, why doesn’t the machine insert the “TO” of its own accord?”


The answer is that it doesn’t quite know enough to do this. For instance, another digit could follow the “1”, as in

```
50 FOR p = 12 TO 40
```

(There are other, more complicated possibilities. The command

```
50 FOR p = 1 - 7 TO 5
```

is legal, and means the same as `50 FOR p = -6 TO 5`.)

The  prompt can appear before you reach the end of a line. For example,

```
20 LET e * s = q
```


The machine expects to find an "=" symbol—or another letter or number—after e, so it can issue the [?] prompt in front of the "*"; although it does not do this until after "ENTER" is hit.

So the rule with syntax errors is: *watch for the [?] prompt*. When it appears the reason will usually be fairly obvious. If it isn't, check the statement you are writing against the appropriate sections of the *Manual*. (One source of error that is occasionally perplexing occurs if you accidentally type out a keyword in full, rather than using its special key. For instance, if you type the two letters "T" and "O" instead of the keyword "TO". The machine will not accept this, although there is no difference on the screen.)

RUNTIME ERRORS

There are many different types of error that can occur when a program is run. I once came across a computer routine that would only work for programs with an even number of characters, because of a rather subtle error; before it was fixed, you could get round it by adding an extra character, somewhere harmless, to anything that failed to run!

It is more useful to give specific examples of possible errors than to define them in general terms. To begin with, let's look at the following piece of code:

```
10 FOR p = -5 TO 5
20 LET a = 10/p
30 PRINT a
40 NEXT p
```

Each of the lines is perfectly valid BASIC and so no syntax errors would be indicated. In fact the program will begin to execute perfectly after "RUN" is entered, and will output

-2	[i.e. 10 / (-5)]
-2.5	[i.e. 10 / (-4)]
-3.333333	[i.e. 10 / (-3)]
-5	[i.e. 10 / (-2)]
-10	[i.e. 10 / (-1)]

But it will then halt with the error message

```
6 Number too big, 20: 1
```

Here the problem is pretty clear—even without looking up the error number in the *Manual*. First, the message shows that the machine is objecting to line 20, which reads

```
20 LET a = 10/p
```

Second, this statement has already been executed several times without any fuss; so the problem must be with one of the quantities which is *changing*, that is, the value of a or p. The value of p which has just been successfully dealt with is -1, so the current value of p is 0. In other words, the machine is trying to work out 10/0, which is infinite (or not defined according to taste) and so cannot be handled, however large a space the machine allocates for the answer.

You probably reached this conclusion long before completing the (rather laborious) analysis above; but I'm using this simple example to illustrate the *kinds* of pointers you should look for when an error has you foxed.

1. Identify the problem line (the number after the comma in the terminating message).
2. Determine whether this statement has been executed at least once, before generating the error message. (If it has, the problem is with the particular value of one of the variables at the instant when the error occurred.)

3. Use the message to get a further clue. In the example this is "Number too big". Note that the error message does not say *exactly* what has happened (i.e. it does not say "Attempt to divide by zero"), so it is not always enough simply to look at an error message in the hope that this will explain precisely what has gone wrong.

There are two other things to say about the form of the Spectrum's error reports.

First, they start with a number or a letter (here 6) which is just an identifier, referring to an entry in Appendix B of the *Manual* (pp. 189–192). This entry may, or may not, give you some more help in deciding what's gone wrong. In this case the *Manual* says "Calculations have led to a number greater than about 10^{38} ", which seems to me much like saying "Number too big" . . .

Second, there is a number, which is normally 1, at the end of the message, after the colon. This refers to "multi-statement lines" where more than one command goes on a single line. We haven't used this technique so far: see Chapter 9 on "Branching" for details. What happens, roughly, is that you can put several statements on a single line, separated by colons; and this final number in the report tells you which of these is the culprit. So 20: 1 means "the first statement in line 20", and 20: 3 would refer to the third.

This seems (and is) very sensible, but there's a possibility of confusion here unless you're careful. The reason is that the Spectrum regards any "natural break" in the sequence of statements (such as THEN) as the beginning of a new "statement" for this purpose. So the line:

```
10 IF p/0 = 2 THEN LET p = p + 1
```

would get the report

```
6 Number too big, 10: 1
```

But if it read

```
10 IF p = 2 THEN LET p = p/0
```

you'd see

```
6 Number too big, 10: 2
```

because the error is in the second part of the statement.

Incidentally, this explains the odd messages you get when *nothing* goes wrong. For instance, type LIST; you'll get the message

```
0 O.K., 0: 1
```

which means

- 0 Report code 0: the machine has done whatever it was asked and encountered no problems.
- O.K. More succinct form of the above.
- 0 It's just executed "line 0", which is just a way of saying that the command had no line number.
- : 1 It was the first statement of that line.

The problem of dividing by zero crops up fairly often, and not necessarily in such an obvious way as that just described. For instance:

```
30 INPUT p, q, r
```

```
40 LET a = (p + q - r * 2) / (5 + (p - r) * (p - r) - 2 * q)
```

will cause the same problem if the values 7, 15, and 2 are entered for p, q, and r respectively. (Try it!) In general, it makes good sense to test divisors to see if they are zero, before attempting the sum. We might rewrite the above example as follows:

```
30 INPUT p, q, r
```

```
32 LET d = 5 + (p - r) * (p - r) - 2 * q
```


34 IF $d = 0$ THEN PRINT "No can do": GO TO 30

40 LET $a = (p + q - r * 2) / d$

The meaning of the GO TO instruction is (I hope) obvious! Line 34 is an example of a multi-statement line.

DEBUGGING PROBLEM

Finally, here's a chance to test your grasp of what has been said so far. The following program is intended to accept a series of positive numbers, and to print out their average. If we wanted to take the average of 2.4, 8.1, 7, and 14 we would enter:

2.4

8.1

7

14

-1

The "-1" at the end is used solely to indicate that no more data are to be entered, and is *not* part of the data. (Such a value is often called a *delimiter*; and you'll see how it works if you look at line 40 of the program below.)

10 LET $s = 0$

20 LET $c = 0$

30 INPUT n

40 IF $n < 0$ THEN GO TO 100

50 LET $c = c + 1$

60 LET $z = s + n$

70 GO TO 30

100 PRINT "AVERAGE IS \square "; z/c

There are some typographical errors in the listing. See if you can correct them by *entering the program as it stands and tinkering with it*. When you've had a go at it, compare your solution to mine by looking at page 39.

Happy debugging!

*There's a little bit of unpredictability
built into the Spectrum*

8 Random Numbers

The instruction RND produces a "random" number between 0 and 1, which can equal 0 but not 1. Actually it's not really random, it's a *pseudorandom* number, and the numbers repeat every 65537 times, but you won't notice this in practice. Since nobody knows what a "random" number really looks like, the use of "pseudo" is a bit pseudo itself.

You can use this in game programs. For instance, to simulate the throw of a die, you note that $6 * \text{RND}$ is a random number between 0 and 6 (not including 6), so $\text{INT}(6 * \text{RND})$ is either 0, 1, 2, 3, 4 or 5, at random; so $1 + \text{INT}(6 * \text{RND})$ is one of 1, 2, 3, 4, 5, 6 at random. Which is what a die does. (Die is the singular of "dice".) To pick a random card from a pack of 52 you'd play a similar game using $52 * \text{RND}$, but you'd need a fancy bit of programming to convert numbers between 0 and 51 into names of cards like "JACK OF CLUBS". It can be done if you're cunning.

You can also use random numbers to do statistical simulations. There's a nice example of this on page 112. MONOPOLY DICE.



Sometimes what you want the computer to do depends on what's happened so far. In this case you use a technique known as

9 Branching

This section is about logic, and *conditional* statements, where what the computer has to do depends on various other things. For example, when going to the pictures, IF you are under 16 THEN you will not be let into an X-rated film. You can put similar conditions on what the computer does.

The instruction you use to do it is IF . . . THEN . . .—but what goes into the dotdottedots is important.

Let's write a program to tell whether a given number is odd or even. ("Big deal!" I hear you cry, and fair enough; but as elsewhere in this it is the principles that count, not the actual result.)

Here goes.

```
10 INPUT n
20 IF n = 2 * INT (n/2) THEN PRINT "even"
30 IF n < > 2 * INT (n/2) THEN PRINT "odd"
```

How does this work? Even numbers are those exactly divisible by 2. So n is even exactly when $n/2$ is an integer. That means taking INT leaves it the same, so $n/2 = \text{INT}(n/2)$. And *that* is the same as $n = 2 * \text{INT}(n/2)$. So line 20 is a fancy way of saying

```
20 IF n is even THEN PRINT "even"
```

which makes reasonable sense to *us*—but not to poor old Spectrum, who has no idea what "is even" means until we tell him in the language he understands.

Just to drive this point home, we'll work out a few cases.

n	22	23	24	25	26
$n/2$	11	11.5	12	12.5	13
$\text{INT}(n/2)$	11	11	12	12	13
$2 * \text{INT}(n/2)$	22	22	24	24	26

That should be enough to convince even the direst sceptic.

(In the same way, k divides n exactly – for whole numbers k , n – if and only if $n = k * \text{INT}(n/k)$. This is a very useful thing to know.)

Perhaps the one other thing to point out is that in line 30 the symbol $< >$ means "is not equal to". A lot of mathematicians would use \neq but the Spectrum prefers $< >$. Don't ask me why.

In general you do the same kind of thing. The crucial instruction takes the form

```
10 IF this THEN that
```

where "this" and "that" are statements. (In the line 20 above, "this" is " $n = 2 * \text{INT}(n/2)$ " and "that" is "PRINT "even"").

The "this" statement has to be something that the computer can recognize as either

true or false. (IF STOP THEN . . . isn't terribly informative, but IF $n = 1981$ THEN . . . is.) If "this" is true the computer then goes on to do "that"; if "this" is false it *goes on to the next program line without doing "that"*.

Two particularly common types of conditional command follow.

CONDITIONAL JUMPS

These take the form

```
10 IF this THEN GO TO n
```

where n is a line number. Instructions like this can be used to change horses in midstream—that is, alter the whole course of the calculation by moving away to another part of the program.

For instance, when calculating the square root of a number it's important to remember that negative numbers don't have any. So you avoid error messages by doing something like this:

```
10 INPUT n
20 IF  $n < 0$  THEN GO TO 50
30 PRINT n, SQR n
40 STOP
50 PRINT "square root not defined"
```

Note the STOP in line 40. Why? Chop it out and see what happens!

In the same way, when using PRINT AT a, b , you can protect against a and b in unPRINTable ranges by using:

```
100 IF  $a < 0$  THEN GO TO 1000
110 IF  $a > 21$  THEN GO TO 1000
120 IF  $b < 0$  THEN GO TO 1000
130 IF  $b > 31$  THEN GO TO 1000
150 PRINT AT a, b; "*"
160 STOP
1000 whatever you think is a useful response . . .
    etc.
```

We assume here that a, b are assigned in a previous bit of program.

Tack on the front

```
10 INPUT a
20 INPUT b
```

and then ask the machine to PRINT AT 999, -37 by setting $a = 999, b = -37$.

Make line 1000 read

```
1000 PRINT "I'm not so stupid any more!"
```

Is it?

If you think lines 100–130 look clumsy . . . they are. See the section opposite on LOGIC.

CONDITIONAL ASSIGNMENTS

These take the form IF this THEN LET something. For instance, an alternative to the first program above is:

```
10 INPUT n
20 LET a$ = "odd"
30 IF n = 2 * INT (n/2) THEN LET a$ = "even"
40 PRINT a$
```

Here a\$ has a dollar sign because it's not a number, but a *string* of characters (see Chapter 17).

What is this program?

```
10 LET s = INT (2 * RND)
20 IF s = 0 THEN LET a$ = "HEADS"
30 IF s = 1 THEN LET a$ = "TAILS"
40 PRINT a$
```

Whenever you are writing a program, and what you want to do depends on certain things happening or not happening, start thinking about IF . . . THEN . . .

LOGIC

A big subject, examined endlessly in learned treatises . . . but we don't need to know most of that. The Spectrum can do logic—probably better than you or I can. Specifically, it can combine statements occurring in the "this" position of an IF this THEN that, by using AND, OR, and NOT.

Basic rules: p AND q is true only when p is true and q is true.
 p OR q is true when p is true, or q is true, or *both*.
 NOT p is true only when p is false.

The Spectrum works out NOTs before ANDs and ANDs before ORs. It works out almost everything else you can think of before it does any of these. As a result, you don't often need brackets to make the order clear.

For instance, we can improve lines 100–130 above by changing them to the single line

```
100 IF a < 0 OR a > 21 OR b < 0 OR b > 31 THEN GO TO 1000
```

There are all sorts of marvellous ways to use Spectrum logic, but they're complicated to explain and I haven't much space, so I'll reluctantly stop here. Chapter 10 of the Sinclair *Manual* will get you started, but it's by no means the whole story.

MULTI-STATEMENT LINES

The Spectrum will allow you to put more than one command on a single line. It is merely necessary to separate the commands by colons ":". So the program in Chapter 1 on Basic BASIC *could* have been written as (say)

```
10 PRINT "Doubling": INPUT x: LET y = x + x
20 PRINT x, y: STOP
```

And in fact it could all have gone on one line, if you'd wanted.

This can be used to save a bit of space (it cuts out line numbers) or to make a program easier to understand. The main snag is that you can only GO TO the start of a multi-statement line. On the whole I've avoided multiple statements—it's usually easier to see what's going on without them—but there is one occasion when they can be very useful indeed. I've already used this trick in Debugging I:

```
34 IF d = 0 THEN PRINT "No can do": GO TO 30
```

This lets the machine take *two* actions if a given condition holds, and avoids using lots of GO TOs.

The essential point to remember is that after a THEN, all of the commands in that line are conditional on the IF statement that precedes the THEN. In other words, a command

IF this THEN that: the other: something else

will lead to *all three* of that, the other, and something else being done (if this is true) or *no action whatever* (if this is false).

Useful, but also a trap. It's very easy to write code like:

```
10 IF x = 0 THEN GO TO 20: IF x = 1 THEN GO TO 1000
20 PRINT "x is zero"
etc.
```

under the impression that the machine will branch to 20 if $x = 0$ and to 1000 if $x = 1$. Not so, not so. If $x = 1$ then it takes line 10 as meaning:

```
IF x = 0 THEN ...
... GO TO 20 and IF x = 1 THEN GO TO 1000
```

but

```
IF x <> 0 (which is the case when x = 1) THEN ...
... ignore the rest of the line and go on to the next.
```

Which here is line 20, exactly where you didn't want it to go!

However, if you replace 10 by

```
10 IF x = 0 THEN GO TO 20
15 IF x = 1 THEN GO TO 1000
```

it all works as you'd expect.

Moral: *multi-statement lines are at their most useful in IF/THEN commands, and also at their most dangerous.*

One of the main reasons for buying a Spectrum is its splendid graphics. Kick off with:

10 Plotting

As small computers go, the Spectrum is equipped with some pretty sophisticated tools for drawing pictures. It has what are known as high resolution and low resolution graphics. These are good phrases for stopping conversation at parties, but what they really mean is that you can draw pictures with a pencil (high resolution) or a creosote brush à la Rolf Harris (low resolution).

In this chapter, I'm concerned only with the high resolution breed of graphics. When you're in this mode, the screen is divided up into a large number of small squares called pixels, 256 of them in the horizontal direction and 176 in the vertical direction. So there are $256 \times 176 = 45056$ pixels altogether.



Figure 10.1

Figure 10.1 shows how they are referred to. The first column is labelled 0 and the last one 255. The bottom row is labelled 0 and the top one 175. A point is blacked in on the screen by writing:

```
50 PLOT 79, 67
```

for instance and, as you can see from the diagram, the first value after PLOT indicates the column number and the second value indicates the row number. The keyword PLOT tells the system that you want to think in high resolution terms; you simply cannot use PLOT for low resolution work. There are two other keywords which are used in high resolution: DRAW and CIRCLE.

Let's deal with DRAW first. At its simplest, DRAW is used to draw a straight line. The starting point for the line is implied; it's wherever the "pencil" is at the moment. The two values after DRAW give the number of columns and number of rows to move to get to the end of the line. So:

```
10 PLOT 30, 30
20 DRAW 40, 80
```

would generate a line as shown in Figure 10.2.

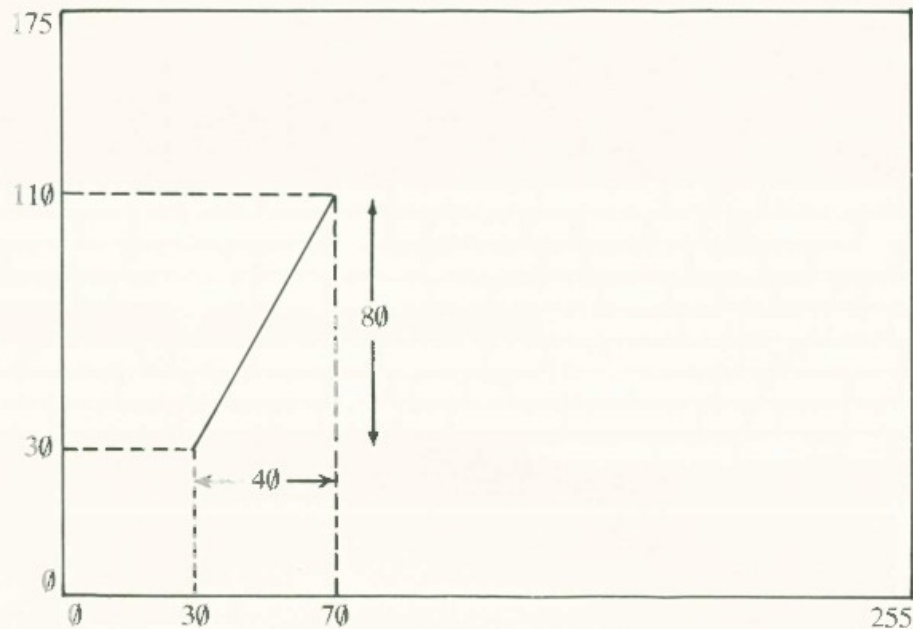


Figure 10.2

If we add:

```
30 DRAW 50, -10
```

we end up with Figure 10.3.

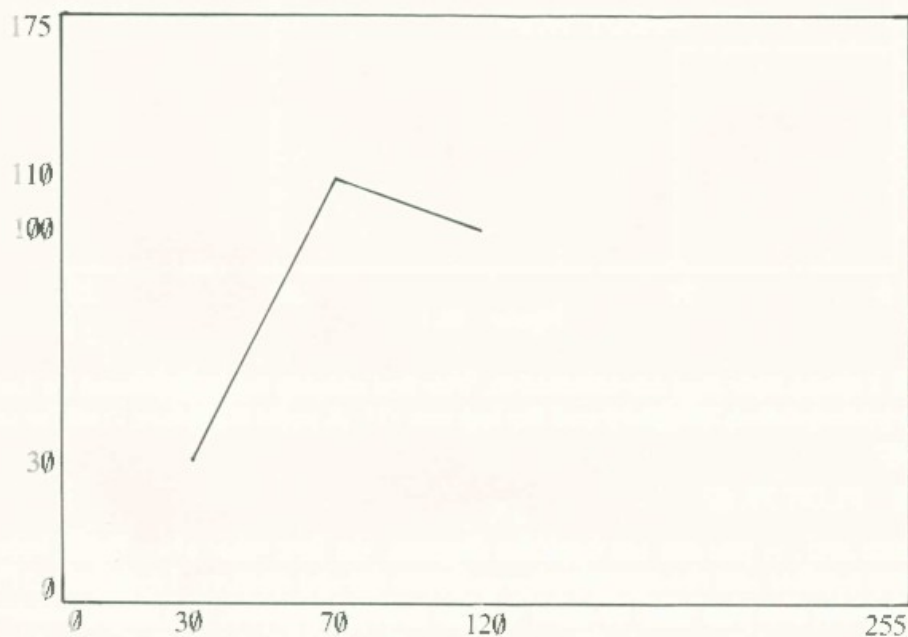


Figure 10.3

So it's pretty easy to draw things like rectangles to order. Suppose that we input the rectangle's details first, giving the values at its bottom left-hand corner, its width, and its height:

```
10 INPUT "Leftmost column"; lc
20 INPUT "Bottom row"; br
30 INPUT "Height"; h
40 INPUT "Width"; w
```

First, we can draw in the bottom left corner:

```
50 PLOT lc, br
```

Now draw in the bottom of the rectangle:

```
60 DRAW w, 0
```

the right-hand side:

```
70 DRAW 0, h
```

the top:

```
80 DRAW -w, 0
```

and the left-hand side:

```
90 DRAW 0, h
```

If the rectangle is to be blocked in, things are a shade trickier. We now need to draw all the vertical lines between the left and right edges. Suppose we wanted to draw just one of them, the one in some column labelled c . (All we know about c for the moment is that it's bigger than lc , the column with the left-hand edge in it, and smaller than $lc + w$, which is where the right-hand edge is.) The code would be:

```
110 PLOT c, br + 1      [to put the "pencil" in the right place]
120 DRAW 0, h - 1
```

Now we need to do this for all values of c from $lc + 1$ to $lc + w - 1$. The -1 's and -1 's are there because there's no point in drawing the edges again. Obviously we can use a FOR loop:

```
100 FOR c = lc + 1 TO lc + w - 1
130 NEXT c
```

Now, to repeat the process so that we can draw rectangles all over the screen all we have to add is:

```
140 GO TO 10
```

Of course, the rectangles will always be filled in because lines 100 to 130 are always executed. We could ask the user whether he wants to block in his rectangle like this:

```
48 INPUT "Block it in? (yes/no)"; b$
```

and then ignore the "blocking in" code if the answer was "no":

```
95 IF b$ = "no" THEN GO TO 10
```

All this assumes that the user is behaving sensibly, and not trying to draw lines outside the screen limits. This is a dangerous assumption to make. Users should be regarded, not so much as idiots who may do something wrong, but as malicious individuals who will certainly do whatever they can to crash your program if you give them half a chance.

So we should insert some tests to make sure that the specified rectangle can be drawn before trying to do so.

First, is the leftmost column on the screen?

```
15 IF lc < 0 OR lc > 255 THEN PRINT "cannot draw this": GO TO 10
```

Next, is the bottom row possible?

```
25 IF br < 0 OR br > 175 THEN PRINT "cannot draw this": GO TO 20
```

Is the height negative?

```
33 IF h < 0 THEN PRINT "that's silly!": GO TO 30
```

Will the top line fit in?

```
36 IF br + h > 175 THEN PRINT "top line won't fit": GO TO 30
```

Is the width negative?

```
42 IF w < 0 THEN PRINT "that's silly!": GO TO 40
```

Will the right-hand edge fit?

```
44 IF lc + w > 255 THEN PRINT "r. h. edge won't fit": GO TO 40
```

We ought also to check that only "yes" or "no" has been entered in answer to the question in line 48:

```
49 IF b$ <> "yes" AND b$ <> "no" THEN PRINT  
    "Please enter yes or no": GO TO 48
```

There's an interesting point here which you should be aware of. As far as the machine is concerned, "yes" is not at all the same thing as "YES". So if we leave the program as it is, the user has caps lock on (i.e. everything is coming up capital letters), he will type either "YES" or "NO" in answer to line 48, and the infuriating machine will keep responding:

Please enter yes or no

Block it in? (yes/no)

See if you can modify line 49 so that the machine will allow the user to type in either upper or lower case.

RUNNING RINGS ROUND THE SPECTRUM

Drawing circles is very easy. There's a special keyword, **CIRCLE**, after which you specify the column and row you want the centre to be in, and then the radius. So, for instance:

```
20 CIRCLE 50, 70, 30
```

will draw a circle whose centre is at the intersection of column 50 and row 70, whose radius is 30.

However, **DRAW** can also be used to produce circles, or rather bits of circles. Try this:

```
10 PLOT 20, 30
```

```
20 DRAW 60, 100
```

```
30 DRAW -60, -100, 1
```

You'll see that line 20 draws a straight line, as you'd expect, and that line 30 gets back to the originally PLOTted point, also as you'd expect, but it draws along a circular path, not

a straight line. It's the third variable after DRAW which tells BASIC to draw an arc of a circle, and the way it works is slightly confusing, but I'm going to explain it anyway.

Imagine the whole circle (the bit which isn't drawn is shown dotted in Figure 10.4).

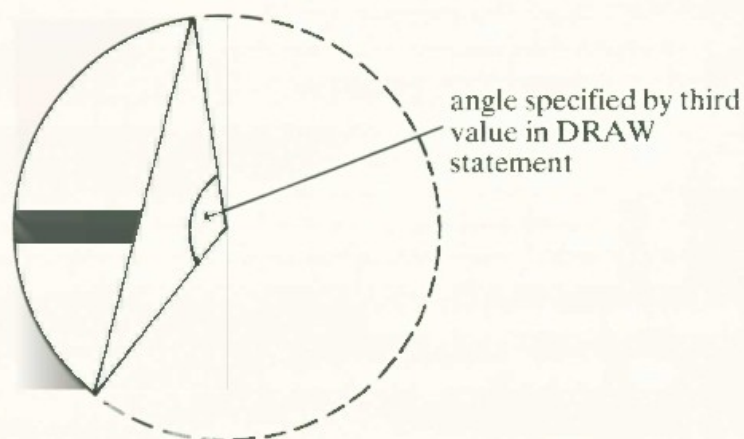


Figure 10.4

If you draw lines from the centre to the ends of the arc, the angle between them is the third value after DRAW. You may find this surprising, because this angle is obviously much bigger than 1. This is because, just so life isn't too easy, the angle is measured in radians, and 1 radian is just under 60° . Now if all this isn't crystal clear and you thought radians were tyres which look flat even when they're pumped up properly, don't worry. All it means in practice is that if you choose a small angle (say 0.1) you'll get a line which doesn't deviate from straight by very much. As you choose larger angles, the circular nature of the line becomes more pronounced. Try:

```
10 PLOT 60, 20
15 FOR a = 0.4 TO PI STEP 0.4
20 DRAW 100, 0
30 DRAW -100, 0, a
40 NEXT a
```

See how the shape approaches a semicircle as the angle gets close to PI (3.14159)?

Now change line 15 to read:

```
15 FOR a = 0.4 TO 2 * PI STEP 0.4
```

You get more and more of a full circle, and eventually it won't fit on the screen. It doesn't matter how small you make the original straight line, it still won't fit sooner or later. The reason is that $2 * \text{PI}$ radians is 360° —a full circle! Since this full circle includes the original straight line—a flat bit—it must have an infinite radius, so it won't fit on anything! The moral is: don't let the angle get too big (about 5 gives you most of a circle) and even then, be careful; it's easy to go careering off the screen, and not easy to test to see where you're going to.

FILLING IN THE HOLES

We've seen how to fill in a rectangle, but filling in a circle or a segment of a circle looks altogether a more difficult proposition. The reason is that it's more difficult to see where to start and stop the DRAW that does the shading in, since these values aren't fixed as they are for a rectangle. And that's the clue. We need a way of finding where the edges of the figure are in each row we want to shade in. (We'll shade the rows, rather than the columns, just for a change).

Fortunately, the Spectrum gives us a way of finding out whether a particular pixel is inked in. It's called the POINT function. If I write:

```
200 LET g = POINT (20, 30)
```

g will be set to 1 if the pixel at 20, 30 is inked in, and zero if it's not.

So the problem breaks down like this:

1. Set up a rectangular space around the figure to be filled in, in which the search for edges is going to take place.
2. For each row do the following:
 - (a) Search from the left until the figure is hit. Note where this is.
 - (b) Search from the right until the figure is hit. Note where this is.
 - (c) Draw a line between the two points found in (a) and (b).

Here's the resulting program:

```
600 INPUT "frame cols"; lc, rc           [the left and right columns of the
                                         framing rectangle]
610 INPUT "frame rows"; br, tr           [the bottom and top rows of the
                                         framing rectangle]
620 FOR r = br TO tr
630 FOR c = lc TO rc                     [search from left to right,
                                         looking for the edge]
640 IF POINT (c, r) = 1 THEN GO TO 660
650 NEXT c
655 GO TO 730                           [if program gets here, there is no
                                         edge on this row, so go to the next]
660 LET c1 = c                           [c1 is leftmost column of figure]
670 FOR c = rc TO lc STEP -1             [search from right to left,
                                         looking for the edge]
680 IF POINT (c, r) = 1 THEN GO TO 700
690 NEXT c
700 LET c2 = c                           [c2 is rightmost column of figure]
710 PLOT c1, r                           [shade
720 DRAW c2 - c1, 0                      in]
730 NEXT r
```

Of course, you'll have to precede this with a routine to draw a closed figure, such as the one to draw a circle segment at the beginning of the previous section, to see anything happen.

There are some simple modifications to this which make it quite powerful.

First edit line 620 to read:

```
620 FOR r = br TO tr STEP s
```

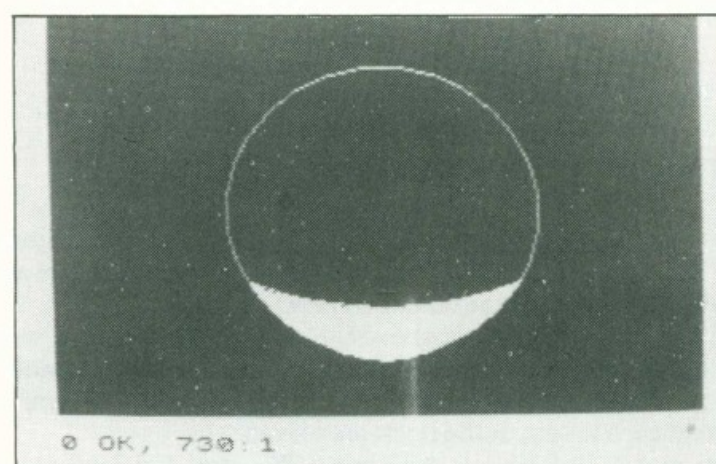
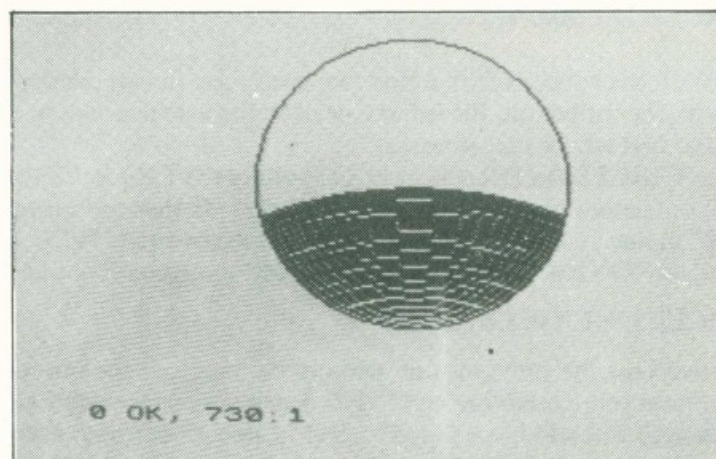
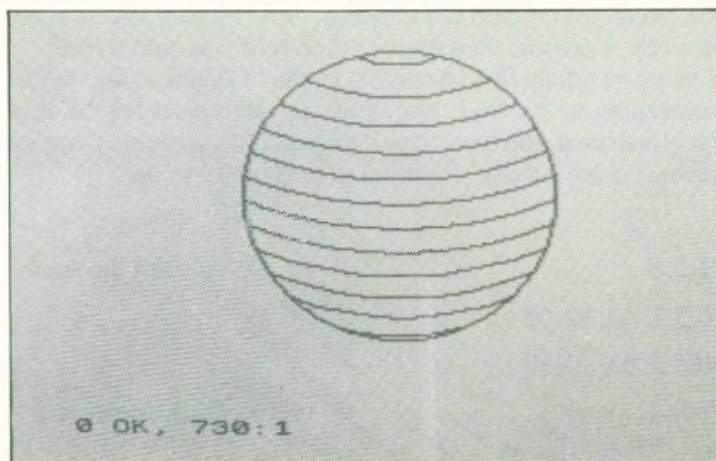
and then write a line which allows the user to enter any value of s he wishes (somewhere convenient before 620). If s is set to 1, the effect is the same as before, but with s = 2, only every other row is drawn in, so we get a hatching rather than a shading effect. With larger values of s, the hatching gets wider, of course.

Now edit line 720:

```
720 DRAW c2 - c1, 0, a
```

and INPUT "a" somewhere convenient. Now try hatching in a circle with a fairly small value of "a" (0.5, say). See the 3-D effect?

Now choose the framing rectangle so that the top half of the circle is above it. Use the same value of a, but set s back to 1. Now you've got a moon in shadow. (Use white ink on black paper to get the best effect.) And so on, and so on. When I first wrote this program I had a lot of fun drawing quill pens in ink pots. Simple things . . .



BASIC'S ELECTRONIC RUBBER

Watching shapes being erased is almost as much fun as drawing them. You still use the `PLOT`, `DRAW` and `CIRCLE` commands for this, but you have to tell BASIC that you've got the rubber in your hand, not the pencil. This is done with the command: `OVER 1`

Try this:

```
50 CIRCLE 120, 85, 60
```

```
60 OVER 1
```

```
70 CIRCLE 120, 85, 60
```

and you'll see the circle drawn, and then erased. Now replace line 70 with GO TO 50 and run it again. The circle is drawn, then erased, then redrawn, and so on!

All right, so I've been telling fibs. Actually, in the "OVER 1" mode the computer rubs out if there is something to rub out, but draws in otherwise. So the first time line 50 is executed, the circle is drawn, the next time it's erased (because it's there to be erased), the next time it's drawn (because there's nothing to erase) and so on.

Now try this:

```
10 OVER 0 [to turn the eraser action off]
20 CIRCLE 40, 40, 30
30 CIRCLE 60, 40, 30
40 OVER 1 [turn eraser on]
50 CIRCLE 100, 100, 30
60 CIRCLE 120, 100, 30
```

See the effect? With the rubber off, the first two circles are drawn overlapping in the way you'd expect. With the rubber on, the other pair of circles also overlap, but the second one rubs out bits of the first where they intersect.

So the action of CIRCLE (or DRAW or PLOT) under OVER 1 is: "If there's something to rub out, rub it out; otherwise draw the desired shape". In the above program, the effect of the "OVER 0" in line 10 lasts until it is countermanded by the "OVER 1" in line 40. It's possible to make an "OVER" command last for only one statement like this:

```
70 CIRCLE OVER 0; 130, 100, 20
```

The new circle overlaps the previous one without rubbing out the intersections but any subsequent statement not containing an OVER specification (try 90 PLOT 0, 0: DRAW 150, 148 for instance) will still have the OVER 1 of line 40 in force, and so will erase at crossing points, although where the crossing point is a single pixel, the effect has to be looked for closely. What would be the effect of editing out the "OVER 0;" from line 70? Try it!

PRESERVING YOUR ART FOR POSTERITY

You can use the techniques we've discussed to draw some pretty fancy patterns, and there's a program in "Prepacked Programs" to make this fairly straightforward (it uses the procedures for shading and hatching described here).

So it would be handy if we could save the results on tape. For instance, we might design a beautiful lunar surface display which we would like to use in a "moon lander" program; or we'd like a series of pictures each of which represents a hole on a golf course. A golf program could load each in turn as the holes are played.

The Spectrum makes this process very easy. We save and load the display screen contents in almost exactly the same way as we save and load programs. The only difference is that we write the word "SCREEN\$" after the normal SAVE or LOAD instruction to indicate that it's the SCREEN we want dumped to tape (or read back) and not a program. For instance:

```
SAVE "GHOLE3" SCREEN$
```

says "save the screen contents as a tape file called GHOLE3" and to get it back:

```
LOAD "GHOLE3" SCREEN$
```


II Debugging II

Earlier, on page 25, I left you with the averaging program which is reproduced below (so you don't have to page-flip too much) and suggested you might like to try debugging it.

```
10 LET s = 0
20 LET c = 0
30 INPUT n
40 IF n < 0 THEN GO TO 100
50 LET c = c + i
60 LET z = s + n
70 GO TO 30
100 PRINT "AVERAGE IS "; s/c
```

Let's now consider a step-by-step approach to the problem. Obviously, the first thing to do is to see if it works as it stands, so we key it into the Spectrum and run it with a few simple sets of data. Suppose we try:

```
3
7
5
-1      [remember—this value just acts as a terminator or delimiter]
```

the answer to which *should* be 5. In fact, when the program is run, you get the error report 2 Variable not found, 50: 1. Hmmm. It means that a previously undefined variable has been used in line 50. From the listing we can see that this is *i*. So let's give *i* a value, say

```
5 LET i = 0
```

That will fix things up so that the program continues beyond line 50. If we add line 5 and RUN again, we find that the message

AVERAGE IS

is printed correctly, but then an error report:

```
6 Number too big, 100: 1
```

is displayed.

We've met this message before: you probably remember that the "6" indicates that the machine has attempted to do a piece of arithmetic leading to a result larger than it can

hold, and that the "100" indicates the line number when the problem arose. So a fair bet would be that the program has tried to divide by zero, as was the case the last time this error report cropped up. Before we start the real detective work, let's collect a little more evidence. Try a different set of data:

2
1
4
6
-1

Exactly the same set of messages results. So a working hypothesis seems to be: "Whatever data are entered, the program terminates with an arithmetic overflow report." Test the program with three or four more sets of data and our working hypothesis looks more and more sound.

So, where to start looking? I used the words "detective" and "evidence" a minute ago; I didn't use them lightly. Debugging programs is like detective work, and any detective will tell you that you have to try to think like the villains to be successful, or, if you like, "it takes a thief to catch a thief". The villain of this piece is the Spectrum, so our problem is to try to think like it. The first thing to do is to slow down your thought processes. That surprised you, didn't it? You were under the impression that computers were a bit quick. Well, so they are, but the way they "think" about a problem is usually pretty laborious. Next, set up a model of the computer's memory, or at least that part of it which is relevant to the problem in hand. In our example, five memory elements have been set up, and have been given the names s, c, n, i and z. A convenient model, then, will just be a table in which we can show how the contents of these elements change as the program is executed. It can also be helpful to have an indication of the line number at which a branch takes place, although this is not essential. So the table might appear as:

Line No.	s	c	n	i	z	Branch

I have added an extra column labelled "Branch" whose function I'll deal with shortly. Now we build up the table by considering the action of each statement in turn. We have to define a data set to deal with; let's choose:

2
1
4
6
-1

So, the first statements to be obeyed are on lines 5 and 10 and simply set the memory elements called i and s to zero. The table appears like this:

Line No.	s	c	n	i	z	Branch
5				0		
10	0					

After line 20 has been obeyed we have:

Line No.	s	c	n	i	z	Branch
5				0		
10	0					
20		0				

Line 30 picks up the first value from the data set and puts it in n, so:

Line No.	s	c	n	i	z	Branch
5				0		
10	0					
20		0				
30			2			

Line 40 is an "If" statement, and so merely performs a test to see if n is negative. It isn't (it's currently 2) so the test is false, which we'll indicate by an "x" in the "branch" column and the branch to 100 does not take place. When a test is true, and the branch does occur we will show this with a "✓" in the "branch" column. Now we have:

Line No.	s	c	n	i	z	Branch
5				0		
10	0					
20		0				
30			2			
40						x

Line 50 asks the machine to take the contents of i, add the contents of c and put the result back in c. This should make us a bit suspicious because we know that i was not set up originally. So maybe adding line 5 wasn't such a good idea. Anyway, let's carry on with our simulated program run.

Line No.	s	c	n	i	z	Branch
5				0		
10	0					
20		0				
30			2			
40						x
50		0				

Of course, adding zero to zero has had no useful effect, and that should make us even more suspicious, so remembering that we are looking for typographical errors, we might suspect that i shouldn't be i at all, but 1, a pretty common slip, after all.

Now we start all over again with a new table, working on the new assumption that line 5 is unnecessary and line 50 reads

50 LET c = c + 1

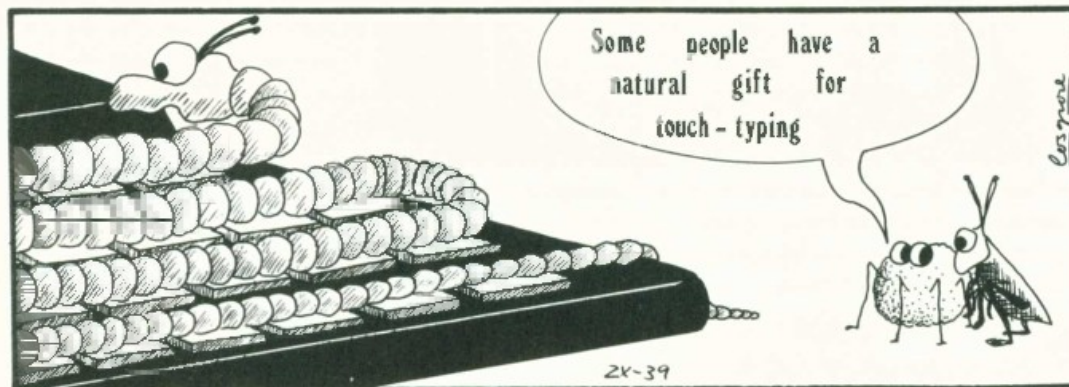
Line No.	s	c	n	z	Branch
10	0				
20		0			
30			2		
40					x
50		1			
60				2	
70					✓
80			1		
90					x
100		2			
110				1	
120					✓
130			4		
140					x
150		3			
160				4	
170					✓
180			6		
190					x
200		4			
210				6	
220					✓
230			-1		
240					✓
250					

When the program reaches line 100 it will print the message "AVERAGE IS" followed by the value of s/c, which is zero!

So the program still doesn't work, but what is interesting is that now there is no error report from the Spectrum; we have introduced a new kind of error—a logical error. The Spectrum can perform the operations we've asked for all right; it's just that having done so, it gets the wrong answer.

Let's have a look at the table we have generated. It is now fairly clear what the function of c is. Every time a new value is entered in n the value in c is increased by one, so that when the branch to line 100 occurs, c contains the number of values entered altogether, in this case 4. But nothing at all has happened to s after it has been set to zero, and z only contains the same values as n, but slightly later. Perhaps s and z should really be the same place, and since s is mentioned at line 10 first, let us assume that z is a misprint for s. That means line 60 reads:

60 LET s = s + n



and the table becomes:

Line No.	s	c	n	Branch
10	0	0	2	x
20				
30				
40				
50	2	1	1	✓
60				
70				
30				
40	3	2	4	x
50				
60				
70				
30	7	3	6	✓
40				
50				
60				
70	13	4	-1	x
30				
40				
50				
60				✓
70				
30				
40				
100				

We now get printed:

AVERAGE IS 3.25

which is correct!

The process I have been describing is called "dry running" the program, and it is a common way of searching for errors. Of course, it isn't always necessary to include all the details I have shown in a dry run program table (for example, the line numbers have not been very useful to us in this case) and it often isn't necessary to complete a table before the light dawns, but I think you can see that it is a nice way of forcing you to think in the strait-jacketed way the machine does, and at the same time giving a clear view of the way the program runs.

Now you may say "all this is very well, but who is going to make typographical errors of this type keying in programs?"

The answer is that it happens all the time, for a number of reasons. First, if you copy a program from a magazine, there is the possibility that the error is present on the printed page. Second, you may easily make the odd keying error in a long program—i for l (if the listing you're copying from has used all capital letters), letter o for zero, 2 for z and so on. Third, even when you are writing a program yourself you can make an error which is equivalent to a typographical error.

For example, suppose you call a variable b3 at the beginning of a program. You write the program over several days, tinkering a bit here, modifying a line there. At the end of this time you need to write a few more lines using this variable, which you remember quite well was called b2, so you don't bother to check.

You don't think it can happen? Wait till you've been programming for a few months.

The Spectrum can draw things. Once it knows what to draw, it can move it around. Once it can move it around, the operator can control the movement through the keyboard. Particularly if you want to write computer game programs, you'll need to know something about

12 Graphics

An extremely attractive feature of computers is that they can draw pictures—with enough hardware, beautiful and intricate multicoloured pictures. And while the Spectrum is rather more limited, it has enough capabilities to provide a stimulating introduction to computer graphics.

Chapter 10 describes one way of using the computer to draw; but there we were thinking of rather academic things like rectangles and circles. By using the PRINT facility, it is possible to draw objects with more visual appeal.

THE “PRINT” INSTRUCTION

The fundamental instruction here is PRINT x\$ where x\$ is a character or a string of characters. It's almost self-explanatory. If you experiment with PRINTing, however, you will soon find that this instruction alone gives you little control over *where* a given character is printed: the machine starts every such instruction at the left of screen on the next available line, which isn't always what you want.

PRINT AT x, y takes care of this. Much as for PLOT, the screen should be thought of as divided into squares, labelled with two coordinates x and y. But there are several differences. First, the squares are twice as large. This means there are half as many each way; a quarter of the total altogether. Next, the coordinate numbering system is quite different—and more straightforward. The first number, x, is a *line* number on the screen; it runs from 0 at the top to 21 at the bottom of the area available for PRINTing. The second, y, is a *column number* giving the distance along a line (that is, horizontally), and running from 0 to 31. Figure 12.1 shows this system in more detail.

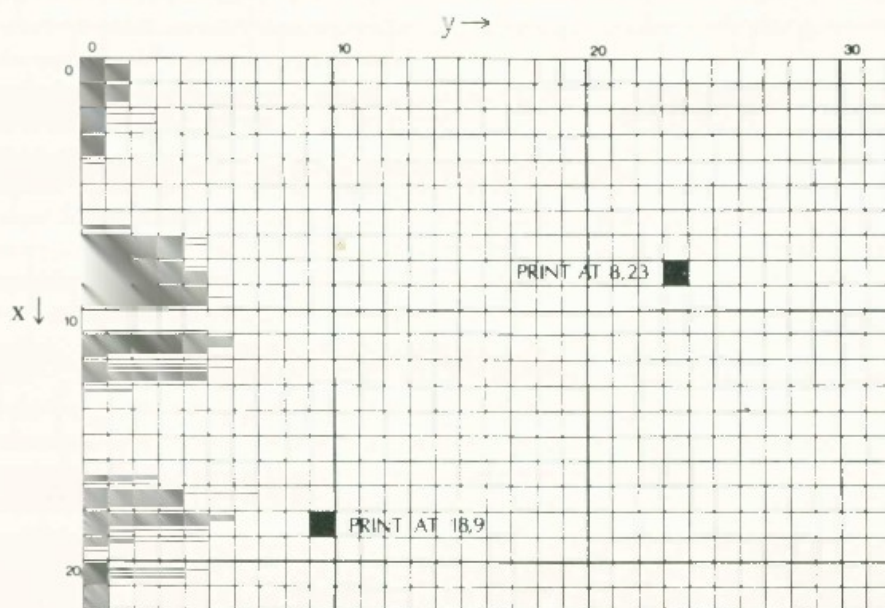


Figure 12.1

Suppose you want to print the graphics character ■ in the middle of the screen. The exact middle can't be achieved; but the square in line 11 and column 15 is pretty close. So you would write

```
10 PRINT AT 11, 15; "■"
```

Note the semicolon (;) which is needed to tell the computer to run what it thinks of as *two* instructions together: move to 11, 15; PRINT something.

By fitting several graphics characters together you can achieve more interesting effects. The most direct way to do this, by using large numbers of PRINT ATs, uses up precious memory the way a Cadillac uses petrol; but for the moment let's not worry about efficiency: only the principle is important. Try this program:

```
10 PRINT AT 10, 10; "■■■■■"
20 PRINT AT 11, 8; "■■■■■"
30 PRINT AT 12, 8; " 00000  "
```

It should remind you of something of a military nature. (If not, blame my poor design.)

When printing graphic designs like this, a useful approach is to draw the design on squared paper, and number the lines and columns; this makes it easier to read off the required instructions. The above program is obtained by this process from Figure 12.2.

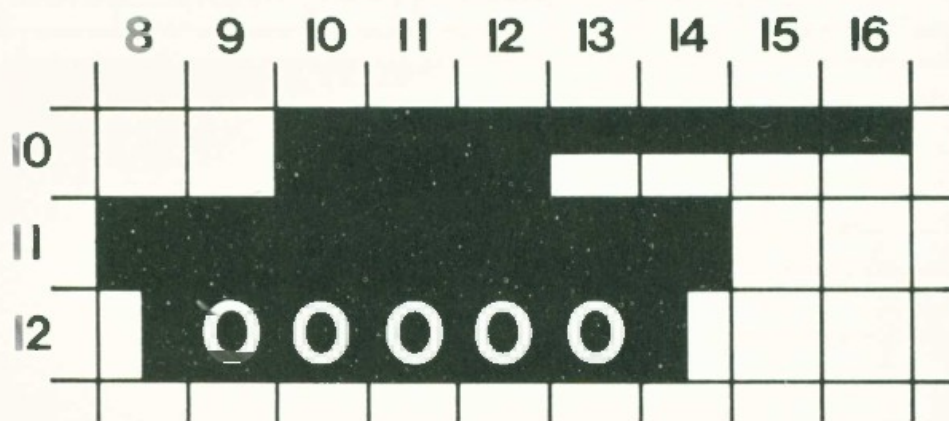


Figure 12.2

The Spectrum has 16 special graphics characters (numbers 128–143 in the character set—see *Manual*, page 186). But you can make good use of other characters too (as in the above example). Inverse video (white on black) characters are especially useful in this connection.

CHANGING POSITION

Having worked out how to PRINT in a chosen position on the screen, it's not hard to modify the program so that you can PRINT the same graphics at any position you choose. The tank (yes, that's what it was supposed to be) above sits on line 12 with its left-hand edge in column 8. If we want to draw it sitting on line *a* with its left-hand edge in column *b*, we just renumber the lines and columns in the picture to achieve this effect (see Figure 12.3).

From this we can read off the new program:

```
10 PRINT AT a - 2, b + 2; "■■■■■"
20 PRINT AT a - 1, b; "■■■■■"
30 PRINT AT a, b; " 00000  "
```

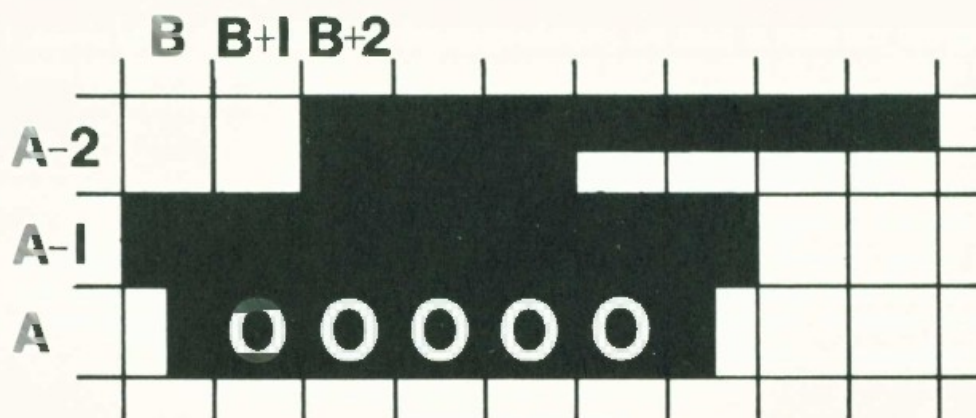


Figure 12.3

Of course this won't run unless you tell it the value of *a* and *b* in a previous piece of program. It also won't run if it encounters values of *a* and *b* that make the picture run off the edge of the screen. Looking at the left edge, that means *b* must be 0 or more; at the right, we need *b* + 9 to be 31 or less, that is *b* = 22 or less. Similarly *a* must be 2 or more, and 21 or less.

The great advantage of thinking generally, like this, is that we can draw our tanks all over the place on the screen, just by specifying *a* and *b*. Here are some samples: tack them on to the lines 10–30 above (the line numbers will automatically be ordered by the Spectrum) and run them, to see what you get.

- (a) 1 LET *b* = 7
 2 FOR *a* = 2 TO 21
 40 NEXT *a*
- (b) 1 LET *a* = 15
 2 FOR *b* = 0 to 22
 40 NEXT *b*
- (c) 1 LET *a* = 2 + 15 * RND
 2 LET *b* = 20 * RND
 40 GO TO 1

(BREAK this to put it out of its misery!)

Once you've learned about subroutines you'll be able to find all sorts of uses for this ability to draw a given shape in an arbitrary position. The next section describes one common circumstance in which this might be used.

MOVING GRAPHICS

Suppose you want the tank to move across the screen from left to right. Program (b) above almost does this, except that it leaves a trail of rear ends of tanks behind it as it goes. You *can* get rid of these by overprinting blank spaces; but the nicest way is to use an "invisible border" which does this automatically. Change lines 10–30 like this:

- 1 LET *a* = 15
- 2 FOR *b* = 0 TO 21


```

10 PRINT AT a - 2, b + 2; "□■■■■■□□□□"
20 PRINT AT a - 1, b; "□■■■■■■■■■"
30 PRINT AT a, b; "□□0000000□"
40 NEXT b

```

Note the blank spaces added at the front of each graphic line. (To avoid running over the edge, we make *b* run only up to 21 in line 2.)

The border of blanks is not visible on the screen; but the way the computer prints out characters means that these blanks are printed on top of that awkward "trail", blotting it out. (An old saying has it that foxes erase their tracks with their tail: so does our tank, but its tail is invisible.)

If we want to be able to reverse the position of the tank, then we need an invisible border at the right end too. To move it upwards, we need a border below it; to move downwards, a border above. To move in all directions (by varying *a* and *b* accordingly) we put the border all the way round the edge of the tank, which means two more lines of blanks in rows *a* - 3 and *a* + 1.

Project

Write a program to make the tank move round and round the edge of a square, size 10 by 10, on the screen. Put invisible borders all round; work out how *a* and *b* have to vary; program that.

KEYBOARD CONTROL OF MOVEMENT

Now we can draw things that move around as we wish, it's possible to use the keyboard to direct the movements from outside the program.

The clumsy way to do this is to write the program as a loop, and to input something from the keyboard. This holds everything up, moving display and all, until the keys are pressed.

Better is the INKEY\$ instruction. A program line like

```
10 LET c$ = INKEY$
```

tells the machine to see which key is being pressed, and assign to the string variable *c\$* the corresponding character. By playing tricks with that character, you can do all kinds of things.

For example, let's direct that tank either to left or right, using keys 5 for left and 8 for right. (This is useful for reminding *you*, because of the arrows on those keys; but it's *not* necessary to press CAPS SHIFT and actually input the arrows.) What do we do? We tell the machine to read the INKEY\$; and adjust the print position according as this is 5 or 8. Like this:

```

1 LET b = 15
2 LET a = 12
3 LET c$ = INKEY$
6 IF c$ = "5" THEN LET b = b - 1
7 IF c$ = "8" THEN LET b = b + 1
10 PRINT AT a - 2, b + 2; "□■■■■■□□□□"
20 PRINT AT a - 1, b; "□■■■■■■■■■"
30 PRINT AT a, b; "□□0000000□"
40 GO TO 3

```

Note the invisible border at both edges. The trouble with this is that if you press the keys for too long, you can run off the edge of the screen. See if you can modify the program to prevent this, by adding lines something like IF a < 0 THEN . . . something or other.

If you keep a key pressed, this program keeps responding, and the tank keeps moving. Sometimes this is a terrible nuisance; often what you really want is for the program to respond only to *changes* in the INKEY\$, or at least to renewed pressings of it.

```
3 IF INKEY$ < > " " THEN GO TO 3
4 IF INKEY$ = " " THEN GO TO 4
5 LET c$ = INKEY$
```

.....

This has the effect of holding everything at line 3 if you're still pressing down the old key. When you let go, it moves to 4 and holds there. When you press a new key or repress the old one—away she goes!

WARNING: do be careful what you tell the machine to do with that INKEY\$. You may only *want* to press keys giving numbers, and then to work out VAL INKEY\$ which converts that number from a character to something you can actually use for arithmetical calculations. But to start the program you have to press ENTER—and sometimes the Spectrum reads *this* as the INKEY\$, tries to turn it into a number—and the program crashes. Or worse, if no keys are pressed, it does VAL (the empty string). This can be a trifle puzzling until you've realized. (You can protect against it by using suitable IF . . . THEN instructions.)

PRINT AND PLOT IN COMBINATION

In some programs you may need to use both PRINT and PLOT together for graphic displays. It is important to remember that PRINT AT x, y and PLOT x, y produce quite different results, because the x, y refer to two very different coordinate systems on the screen. The Sinclair *Manual* has a diagram on page 102 showing both. But usually the easiest way to make the two kinds of command fit together properly is to make a rough sketch, on squared paper, of the region you want to draw graphics in; mark in both coordinate systems; and refer to this sketch while writing your program. It is often worth spending some time thinking about the structure of the program *before* sitting yourself at the keyboard. (Equally, if your first attempt goes wrong, it is often easier to debug by experimenting on the machine, instead of bashing your brains out. The knack is to allocate your time and effort to best effect.)

PAUSE

The Spectrum has a PAUSE command, which makes it wait for a specified period of time. This is very useful for moving graphics; say, to slow down something which would otherwise move too fast. To pause for n seconds type

```
PAUSE 50 * n
```


If the character set doesn't contain one that you want, you can invent a new one for yourself:

13 User-defined Characters

The Spectrum has 21 characters that you can alter as you wish. They are numbers 144–164 in the character set, and to begin with they are set as letters A–U. To access them from the keyboard, go into “G” mode and hit the corresponding letter key.

The technique for building your own graphics characters is basically simple, but deserves proper explanation because it is such an elegant feature, and can turn an otherwise dull program into something quite special.

The first step is to sketch out the character you want on an 8 by 8 grid, blacking in those squares which will be INK when the character is printed. For example, Figure 13.1 shows a “cat” character from *Computer Puzzles: For Spectrum and ZX81* (Shiva, naturally).

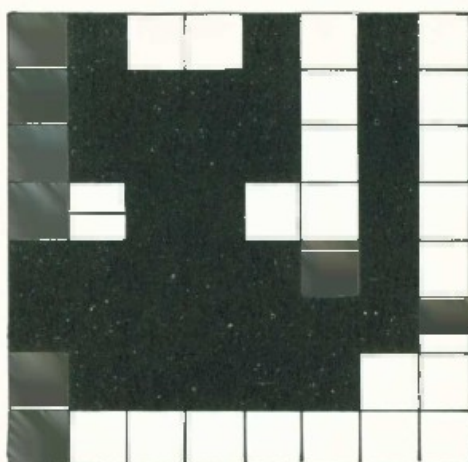


Figure 13.1

Next, replace the black squares by 1's and the blanks by 0's, to get a list like this:

0	1	0	0	1	0	1	0
0	1	1	1	1	0	1	0
0	1	1	1	1	0	1	0
0	0	1	1	0	0	1	0
1	1	1	1	1	0	1	0
1	1	1	1	1	1	1	0
0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

Next, decide which letter you want to use: the natural one is "C".

Doing things the hard (but easy) way, input from the keyboard the following commands:

```
POKE USR "C", BIN 01001010
POKE USR "C" + 1, BIN 01111010
POKE USR "C" + 2, BIN 01111010
POKE USR "C" + 3, BIN 00110010
POKE USR "C" + 4, BIN 11111010
POKE USR "C" + 5, BIN 11111110
POKE USR "C" + 6, BIN 01111100
POKE USR "C" + 7, BIN 00000000
```

Consider this as a bit of magic, if you wish! POKE puts certain information into the computer's memory (see Chapter 23, on PEEK and POKE), USR just tells it that user-defined characters are in operation, and BIN stands for "binary". The important things are the "C", the letter you chose; the numbers 1, 2, . . . , 7 added in; and the rows of 0's and 1's which are copied, in turn, from the table that I got from my cat picture.

Any 8×8 pattern of 0's and 1's can be treated this way; and the machine will store up to 21 different user-defined characters at a time. They are *not* destroyed by NEW; but they aren't SAVED.

There are other ways to set up the characters. I've written out a prepacked program (CHARACTER BUILDER) that will let you *design* your character and then load it up. One way is to convert the binary numbers to decimal—say by direct commands like

```
PRINT BIN 01001010
```

which yields the answer 74. The rows of the cat, in decimal, are

74, 122, 122, 50, 250, 254, 124, 0

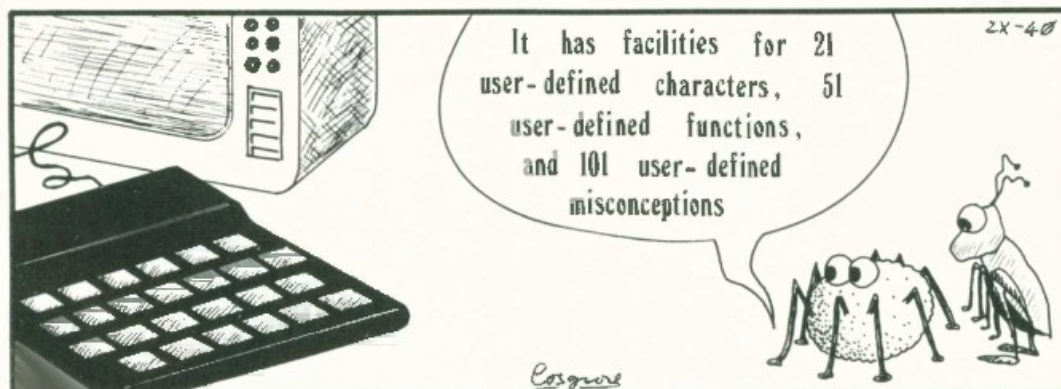
and instead of POKEing around in the BIN you can use these numbers directly:

```
POKE USR "C", 74
POKE USR "C" + 1, 122
. . . . .
POKE USR "C" + 7, 0
```

Now, by writing these instructions into the program, you can set up the characters during a RUN; and by saving the program, you effectively save the characters.

You can also save the characters by using the version of SAVE that saves a block of memory, and LOADING back the same way; but that's a little advanced for this book.

Of course you can store the list of numbers 74, 122, etc. as an array; or use the DATA command (see Chapter 18, on Data).



Projects

1. Set up user-defined characters for the four suits of cards, stored as "H", "C", "D", "S", by using the grids in Figure 13.2.

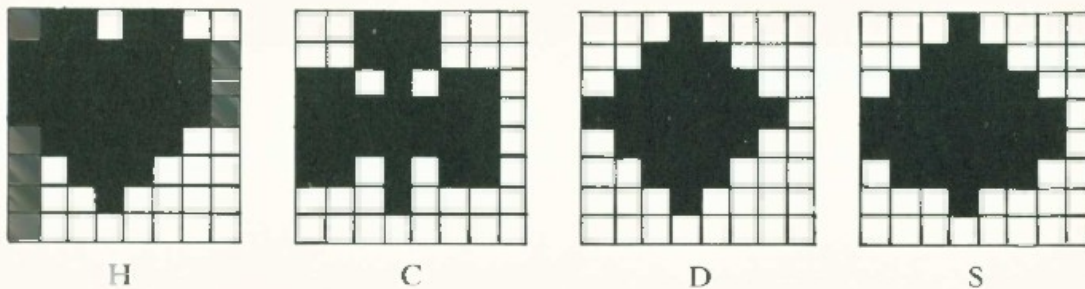


Figure 13.2

2. Use Figure 13.3 to set up a character for "1/2".

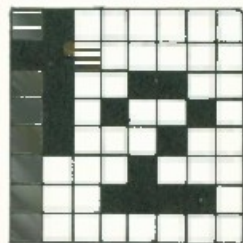


Figure 13.3

3. Design a set of characters for the twelve signs of the Zodiac, and set them up in positions A–L.
4. Design two different characters representing a dog, but differing only in the position of the tail. PRINT them alternately at a fixed position, and make the dog wag his tail.
5. Possibly using several characters stuck together, make a man that walks.

If you can break a job down into a number of identifiable subtasks, then a powerful weapon in the programmer's armoury becomes accessible:

14 Subroutines

There's a story about mathematicians that could equally well apply to computer programmers (yes, they *are* different animals!). About boiling a kettle. First the mathematician is asked to describe the stages to make a cup of tea, when the kettle is hanging on the hook in the kitchen: so he says something like "Take the kettle off the hook, fill it with water, put it on the stove, light a match . . ." and so forth. Then he is asked to describe the steps needed to make a cup of tea when the kettle is standing on the kitchen table—and he answers "Hang the kettle on the hook and proceed as before".

What he's doing is using the first sequence of operations as a *subroutine*. Second time, he modifies the existing situation to make the subroutine applicable, and then uses it as if it were a single indivisible operation.

In computer jargon, a subroutine is a chunk of program that can be written in its own right, and used repeatedly as part of some larger program. Subroutines are a very civilized way of writing programs because they usually make it easier to see what's going on. It is also easier to debug a program written with subroutines, because it is possible to debug each on its own, and then just to check they link together properly.

The relevant command is GO SUB. It's like GO TO but much more versatile. It typically occurs in this sort of way:

```
100 GO SUB 500
110 various other junk
    .....
500 do something
    .....
570 RETURN
```

where, as usual, the small letters denote other bits of program. What this does is:

- (a) On hitting line 100 the program jumps to 500, *remembering where it came from*.
- (b) It then executes 500, and whatever follows it, until it hits line 570 and is told to RETURN.
- (c) It then goes back to the original line (here 100), and carries on with the *next* instruction after that.

Essentially this is like GOTO, except that the return to the starting point is automatic. The important feature, though, is that you can enter the same subroutine from *different* lines in the same program, and the computer still keeps track of which it came in by, and RETURNS to that one.

As an example, I'll give a blow-by-blow account of the writing of a program which:

- (d) Uses keys 5, 6, 7, 8 to move around the screen a cursor P.
- (e) Prints in place of the cursor any character input from the keyboard.

For this character input we'll need to use INKEY\$. We'll read the current key using this, and assign that to a string variable a\$. We want the program to respond only to newly pressed keys, as described on page 48; so we need a chunk of program in the form

```
1000 IF INKEY$ <> "" THEN GO TO 1000
1010 IF INKEY$ = "" THEN GO TO 1010
1020 LET a$ = INKEY$
```

} will become
the subroutine

The 1000's are used because this will be a subroutine and we want to tuck it well out of the way (although actually you can often save a few program lines by putting all subroutines at the *front* end and starting the run using GO TO and not RUN—but that's a refinement not worth going into in detail here). To come back out of it we'll need the extra line

```
1030 RETURN
```

Next, to move that **P** cursor around, we have to . . . what? Well, we're certainly going to need to know *where* to print it; so we'll invent two variables, a and b, which give the values of the row and column on which to PRINT it. To stop the whole thing crashing before we get properly started we'll need to assign values to these. The middle of the screen is a good place to start:

```
10 LET a = 10
20 LET b = 15
```

Now, we want to use keys 5, 6, 7, 8 to modify a and b, thus moving the cursor around. Good old Subroutine above will read the keyboard; so the obvious thing to do next is

```
30 GO SUB 1000
```

Now a\$ will tell us which of 5, 6, 7, 8 has been pressed. We want to move the **P** around in the direction of the four arrows on these keys. (That's why we use them: the arrows are good mnemonics!)

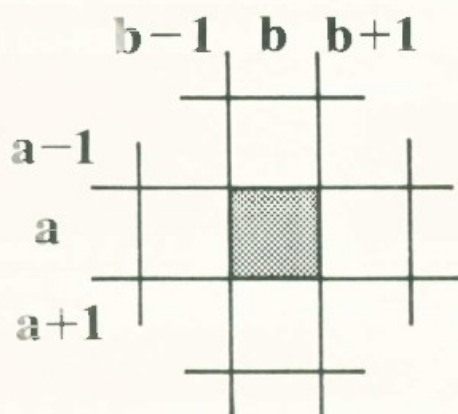


Figure 14.1

Take a look at Figure 14.1 which shows position a, b and the four neighbours. Using the clues in the picture, we see that we want:

- key 5 to change b to b - 1 and leave a alone,
- key 6 to change a to a + 1 and leave b alone,
- key 7 to change a to a - 1 and leave b alone,
- key 8 to change b to b + 1 and leave a alone.

Here's one way:

```
40 IF a$ = "5" THEN LET b = b - 1
50 IF a$ = "6" THEN LET a = a + 1
60 IF a$ = "7" THEN LET a = a - 1
70 IF a$ = "8" THEN LET b = b + 1
```

Having moved the cursor, we'd like to see where it's gone; and for fun we'll make it flash:

```
80 PRINT AT a, b; FLASH 1; "P"
```

Now what? We want to input a character to print in place of this **P**. Subroutine again!

```
90 GO SUB 1000
```

This time the machine reads the keyboard, assigns a\$ the value it finds there (which is whatever we press, a, b, c, d, . . . , 7, 3, >, . . .), and returns to *the next line after line 90*. And we want this to tell it to print the character it's found:

```
100 PRINT AT a, b; a$
```

Nearly done now. So far this all works only once. We want to go back to the front and start again—but keeping the new a, b, *not* resetting to 10 and 15. That's easy:

```
110 GO TO 30
```

(30 then immediately sends it to 1000 again. Why won't 110 GO SUB 1000 work?)

Write all the above lines in (we've made sure the numbers work out in the right order, which isn't always the case first time round when you design a program, so don't be misled!) and press RUN. Nothing will happen; but if you press any of 5, 6, 7, 8 you'll see the **P** cursor in its new position. (If you press anything else, it appears at the old a, b—a programming bonus we hadn't actually planned.) Then press a key, say t. The **P** disappears, to be replaced by a t. Move the cursor (now invisible) using 5, 6, 7, 8; print your next character; and keep going. You can draw words all over the screen. (How about a program for computer-aided crossword design?)

To avoid crashing at the edges of the screen, a bit of user protection is a good idea:

```
75 IF a < 0 OR a > 21 OR b < 0 OR b > 31 THEN GO TO 30
```

and no doubt you can think of refinements.

Project

Instead of the INKEY\$ subroutine, generate both 5, 6, 7, 8 inputs and the PRINTable characters at random; set the whole thing going and await developments. To print random characters use things like PRINT CHR\$ INT (65 + 26 * RND) which selects from the alphabet at random. (Why?)

Here's another example—an introduction to Computer Art. It draws random squares, black or chequered, until it runs out of memory or until you stop it with BREAK.

```
10 LET a = 10 * RND
20 LET b = 10 * RND
30 LET q = 5 * RND
40 LET r = 5 * RND
50 LET k = INT (2 * RND)
60 IF k = 0 THEN LET m$ = "■"
70 IF k = 1 THEN LET m$ = "■"
```



```

80  GO SUB 1000
90  GO TO 10
1000 FOR i = a TO a + q
1010 FOR j = b TO b + r
1020 PRINT AT i, j; m$
1030 NEXT j
1040 NEXT i
1050 RETURN

```

Project: Graphic dice

Generate a random number between 1 and 6 using `INT (1 + 6 * RND)`. Depending on what this number, say *n*, is, print out *n* dots spaced on the screen just like they are on a die, near the centre of the screen. If ENTER is pushed, repeat this.

To do this, invent a subroutine for each of the six possibilities. If the subroutine for *n* is written starting at line (say) `500 + 100 * n` (that is at 600, 700, etc.) and ends well before the next one starts, you can cheat and use `GO SUB 500 + 100 * n` to avoid complicated conditional jumps. But you do need six separate RETURN commands. . . .

For the ENTER repeat, use a line asking for an input character `INPUT k$`; follow it by a `GO TO` command sending everything back to the beginning. *k\$* isn't used for anything; but the machine waits for it, gets ENTER, and carries on to the `GO TO`. Get it?

15 Son et Lumière

Try this:

```
10 FOR i = 1 TO 6
20 INK i
30 CIRCLE 100, 80, i * 10
40 BEEP 0.5, i
50 NEXT i
```

Not terribly exciting, perhaps, but it illustrates a couple of points. Firstly we can see how to change the colour of the symbol being plotted (or printed). We just specify the number associated with the colour we want in an "INK" statement. We don't have to remember which number corresponds to which colour, because the colours are written above their equivalent digits on the keyboard. So in the program, the first time line 20 is executed $i = 1$, and the statement "INK 1" is interpreted by BASIC as "use blue ink for printing and plotting until further notice". Of course, the only thing being plotted while the "ink" is blue is the smallest circle. By the time the next circle is plotted, line 20 has become "INK 2" so red is used, and so on.

After each circle has been completed, the computer emits a triumphant beep. That's the effect of line 40. The first value after BEEP gives the duration of the note in seconds (so each of them lasts 0.5 seconds in this program—sounds longer, doesn't it?) and the second identifies the note to be played. If this value is zero, the note is middle C. 1 is C#, 2 is D, 3 is D#, 4 is E, 5 is F (there's no such note as E#!) and so on. Negative values take you below middle C. -1 is B, -2 is A# etc.

The simplest way of using BEEP is to signal to the user that something has happened (such as an error in the program) or that the computer is waiting for, or has accepted input. Of course, it can be used for playing music but I want to leave that until later (see *Prepacked Programs*) and deal here with some more features of the colour system.

First, it's not only the ink colour which can be changed. The background (which BASIC, quite sensibly, calls PAPER) can be any of the eight colours. Of course, if INK and PAPER are the same colour, you won't see anything happening, which can be confusing if you type LIST after running a program which has altered the INK colour to that of PAPER. (I mention this because I keep doing it, and then wondering why the program has been deleted. It hasn't of course; I just can't see it.)

Now you might expect to be able to write "PAPER 2" and get a red background, but if you insert this statement in the program at line 5, say, no change takes place when you RUN. The reason is that the system can't change the colour of paper which has already been written on, and it can't be sure that nothing has been written unless it has executed a CLS (clear screen) instruction. So only when a CLS is encountered will the machine respond to the most recent PAPER command.

It's also possible to alter the border colour (for instance by typing `BORDER 4` to get a green border), and this is effective as soon as it is encountered. So if you insert:

```
25 BORDER i
```

the border will always match the circle being plotted.

Edit line 25 so that it reads:

```
25 BORDER 7 - i
```

That's a little more arresting (not to say tasteless) isn't it?

Now add the following lines:

```
60 INK 4
```

```
70 PLOT 100, 0
```

```
80 DRAW 0, 175
```

```
90 PLOT 0, 80
```

```
100 DRAW 255, 0
```

and RUN the result.

A pair of green cross-wires are drawn in, as you'd expect, but if you look closely, you'll see that all the circles have now been coloured green in the region where the cross-wires intersect them. This is not a fault of your television or a bug in the Spectrum. It is a feature of the way the Spectrum handles the display. (A bug which can't be fixed is always called a "feature" in computing jargon. It makes it sound as though it was meant to be there.) Anyway, the reason for this odd behaviour is that the attributes of a point on the screen (that's to say its colour, brightness, whether it's flashing) are not limited to just one pixel. They refer to a whole character, which, as we've already seen, takes up 64 pixels arranged as an 8×8 square. Consequently, when you change the colour of one pixel, all the others that are INKed in from the same 8×8 region change with it.

Mostly, you can live with this. Changing the brightness of a character square generally presents no problems either. This is done using the statement `BRIGHT 1` to turn the brightness up and `BRIGHT 0` to turn it down again. Once a `BRIGHT 1` command has been executed everything is plotted bright until the next `BRIGHT 0` is reached. So if we wanted the two inner circles to be plotted more brightly than the others, we could add the statements:

```
5 BRIGHT 1
```

```
23 IF i > 2 THEN BRIGHT 0
```

Flashing is a different kettle of fish. You can turn flashing on and off with `FLASH 1` and `FLASH 0`, much like using `BRIGHT`. However because of the attributes problem, if you try this with the circle drawing program, great blocks of the screen flash at you in an unnerving way. Try:

```
4 FLASH 1
```

```
6 CLS
```

to see what I mean. (Like `PAPER`, `FLASH` needs a `CLS` to be activated). You may have a use for this kind of display, but the only one I can think of is inducing migraines in experimental rats.

Generally, then, `FLASH` is most used in `PRINT` statements, where we want to flash whole character squares, rather than in `PLOTs`, `DRAWs` and `CIRCLEs` where the effects are likely to be unwelcome.

Better yet, all the attributes, `INK`, `PAPER`, `FLASH` and so on can be "embedded" into a `PRINT` statement, in which case their effect is limited to the symbols displayed by

that statement and you don't need a CLS to activate them. For instance, you can write:

```
110 PRINT AT 10,0;INK 5;FLASH 1;PAPER 4;"xxx"
```

and RUN (having deleted line 4 and executed FLASH 0: CLS as a direct command to get rid of the flashing) and just the x's will flash. If you now type LIST you'll see that the ink is still green (set by line 60) even though it's been set to cyan in 110; and the listing doesn't flash.

I haven't dealt with all the ways of handling the display that the Spectrum allows, and I don't intend to. This is an introductory book, and the whole point is not to weigh you down with reams of detail. When you're happy with these techniques you'll obviously want to bone up on the remainder from the *Manual*, and you shouldn't then have much difficulty doing so.

Three small points to finish with. Firstly, the need to clear the screen before some attributes take effect doesn't actually mean you have to enter CLS. For instance, hitting the ENTER key with nothing before it will have the same effect as CLS: LIST. So instead of typing FLASH 0: CLS to turn flashing off you can type FLASH 0 and then hit the ENTER key twice instead of the usual once.

Secondly, you'll have noticed that the region of the border just below the paper behaves oddly, not always changing as it should and retaining its colour at the beginning of a run. You may have realized that this region corresponds with the command and message line (i.e. it's where the BASIC system communicates with you). That's the problem; the BASIC system doesn't want to fall into the trap I mentioned earlier of writing in green ink on green paper. It is, after all, rather important that you know when it's trying to tell you something. In fact, it varies its ink between black and white depending on the border colour so that the result is as clear as possible. So this region has a fixed colour during each run. If you loop the whole program by inserting:

```
120 RUN
```

and then BREAK in and CONTINUE at various points you'll see the effect very clearly.

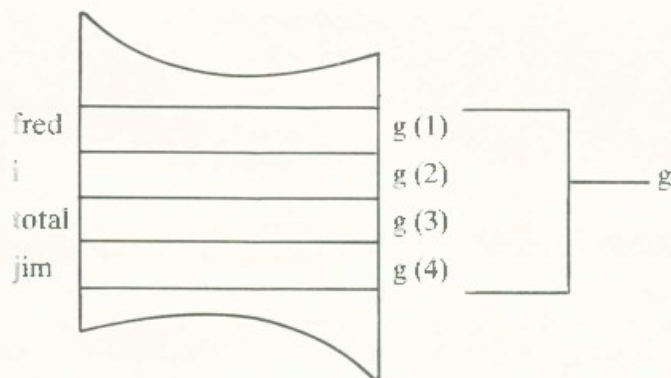
Thirdly, a point of style. When you write "FLASH 1" you are turning the FLASH attribute on. How much nicer it would be if you could write "FLASH on" and "FLASH off" or "BRIGHT on" and "BRIGHT off" instead of using the rather meaningless 0 and 1. It's easily done. Insert a line 1:

```
1 LET on = 1: LET off = 0
```

Now when you write "FLASH on", BASIC will replace the 1 by "on" (or 0 by "off") and the program reads better.

ARRAYS

It's possible for a group of memory cells to be related by having the same name. Such a group is called an array. We can think about it like this. Here's a chunk of memory:



Usually we would give each cell a BASIC name (fred, i, etc.) as shown on the left. But we can call all four cells by the same name (I've chosen "g") as shown at the right. If we adopt this approach (and why we should want to won't be clear for the moment) there are three things to bear in mind:

1. An array name is just a single letter, so you can't have an array called t2 or albert.
2. Arrays can be as long as you like (within the limits of the machine's memory) so you need to tell BASIC how big each one is before you use it. There's a DIM (short for dimension) statement for this, which would look like:

```
10 DIM g (4)
```

for the array in the diagram. In other words, the name of the array is stated, together with the number of cells in it, in brackets.

3. We need to be able to identify particular cells within the array, and the technique for doing so is shown in the diagram; the first cell is referred to as g (1), the second as g (2) and so on.

Let's look first at how a set of 20 numbers could be input to an array. The simplest way would be to write:

```
10 DIM n (20)
20 INPUT n (1)
30 INPUT n (2)
40 INPUT n (3)
.....
210 INPUT n (20)
```

but that is obviously tedious and there's clearly no advantage over using separate variable names, since every array element is named in the statements from 20 to 210.

However, the trick is that the content of the brackets doesn't have to be a number. It can be a variable name. So we can talk about n (p) for instance, and that will mean n (2) if p = 2 and n (17) if p = 17.

Now the problem's easy. We can see that the value in the brackets goes from 1 to 20 in steps of 1, and that's a cue for a FOR loop:

```
10 DIM n (20)
20 FOR p = 1 to 20
30 INPUT n (p)
40 NEXT p
```

First time round the loop p = 1 so line 30 is equivalent to INPUT n (1). Second time around p = 2 and line 30 becomes INPUT n (2) and so on. I've chosen the variable name "p" quite deliberately. It is pointing at the array element being referred to all the time, and this is a very useful way of thinking about array manipulation.

MUSICAL CHR\$

Now let's do something interesting with arrays. We've already met BEEP and remarked that it would be possible to use it to play music, although at the most primitive level it would be tedious, because we would need to work out the number corresponding to each note and then enter a whole series of BEEP commands with the appropriate numbers.

Why not let the computer earn its keep and do the translation from note to number for us?

We'll keep things simple to start with, and work in only one octave, that around middle C. The numeric codes for these notes are:

A : -3	D#: 3
A#: -2	E : 4
B : -1	F : 5
C : 0	F#: 6
C#: 1	G : 7
D : 2	G#: 8

Since the sharp notes always have a value one higher than the corresponding natural, we don't need them tabulated, so the fundamental information we require is:

A : -3
B : -1
C : 0
D : 2
E : 4
F : 5
G : 7

There's no neat connection between these numbers, so it makes sense to hold them in an array and look them up when we need them.

Storing them is easy; we could write:

```
10 DIM s (7)
20 LET s (1) = -3: LET s (2) = -1: LET s (3) = 0: LET s (4) = 2
30 LET s (5) = 4: LET s (6) = 5: LET s (7) = 7
```

(There are other ways which would be more satisfactory if the array were larger; but this works fine here.)

Now we input a note (as a letter) from the keyboard, but then we need some way of using it to "look up" the appropriate number in the array s. In other words we want something like:

```
40 INPUT "Enter note"; n$
   [Some process which generates the right
   value of p from the value of n$ (e.g. if n$ = "A", p = 1
   if n$ = "B", p = 2 etc.) ]
100 BEEP 0.5, s (p)
```

Now, of course, we could write a series of statements like:

```
45 IF n$ = "A" THEN LET p = 1
50 IF n$ = "B" THEN LET p = 2
....
```

and it would work, but it's messy. A better way is to make use of the internal numeric codes of the characters. "A" is actually stored as the number 65, "B" as 66 and so on. (See Appendix A of the *Manual*.)

You can write:

```
LET p = CODE "A"
```

to access the code for A. So if we put:

```
50 LET p = CODE n$
```

we would get values of 65, 66, 67 and so on for p, which are obviously 64 bigger than we want. So line 50 should be:

```
50 LET p = CODE n$ - 64
```

Now add a line to loop back to the input statement:

```
110 GO TO 40
```

and we've got a (very) primitive music keyboard. (Don't forget to use capital letters to input the notes!)

The first thing wrong with this is that we've conveniently ignored the sharps. So if you type in "C#" you'll still get C. Since the CODE function takes the code of the first letter in a string it won't even have noticed that the "#" was there!

We need to examine the second symbol in the string, n\$(2)

```
60 IF n$(2) = "#" THEN LET i = 1
```

Then we can modify line 100 to read:

```
100 BEEP 0.5, s(p) + i
```

so that the note being played is increased by 1 if i = 1. Of course, i must be set to zero at the beginning of the loop otherwise, once one sharp has been encountered, i will remain at one and all the notes will be treated as sharp:

```
45 LET i = 0
```

Unfortunately things are slightly complicated by the fact that n\$(2) doesn't exist if you enter a natural note (just B, for instance). So as things stand the program works if you enter C#, A#, G# but you get an error message as soon as you try C.

The easy way out of this difficulty is to define n\$ as an array of length 2:

```
11 DIM n$(2)
```

Now n\$(2) always exists, but if it isn't used, BASIC will fill it with a space.

The second thing wrong with our music player is that it plays notes in a halting fashion, depending on your typing ability and the vagaries of the keyboard (after all, the Spectrum isn't meant to be an electronic organ).

Rather than play a note as soon as it's entered, why not store it (you guessed—in an array) and play the whole tune when all its notes are in the machine. That will also have the advantage that we can repeat the melody whenever we like.

First we need another array to hold the coded notes:

```
12 DIM t(1500)
```

Our tune can have up to 1500 notes in it.

The input loop changes somewhat, because the number of notes is no longer unlimited, and also because the value to be BEEPed is stored in t, not played. Also, we'll need a way out of the loop if there are fewer than 1500 notes together.

Lines 20 and 30 don't alter, and are outside the loop of course, so let's start it at line 35:

```
35 FOR q = 1 TO 1500
```

After the INPUT at line 40 we'll need a test to see if the sequence is terminated yet. Let's use "***" to finish with:

```
42 IF n$ = "***" THEN GO TO 200
```

to get us out of the loop.

Line 100 has to alter to store the coded note in t rather than BEEP it:

```
100 LET t(q) = s(p) + i
```

Since q gets the successive values 1, 2, 3 etc. the first note will be stored in $t(1)$, the next in $t(2)$ and so on.

The loop terminates at 110 with:

```
110 NEXT q
```

instead of the GO TO which was there before.

Now we can set about playing the tune from line 200 onwards:

```
200 FOR r = 1 TO q      [Note: this loop only goes as far as q, not 1500.  
210 BEEP 0.5, t(r)      This is because, when the first loop is left, q has in it  
220 NEXT r              the number of notes in the tune.]
```

For continuous repetition, all we need add is:

```
230 GO TO 200
```

In case the various alterations and revisions we've gone through have confused you, you'll find the completed final program listed in the Prepacked Programs section.

Finally, let's add a coloured display to go with the music. About the simplest thing we can do is to alter the border colour with each note played. Now, it would be nice to write:

```
215 BORDER t(r)
```

so that each note had an individual border colour, but we can't, because the range of values which can be in $t(r)$ is -3 to 8 , and the range of possible colours is only 0 to 7 . If we add 3 to $t(r)$ the range becomes 0 to 11 , and if the result is multiplied by $7/11$, we get 0 to 7 as we want. So:

```
215 BORDER INT ((t(r) + 3) * 7/11)
```

Now alter the paper colour every time a note is played like this:

```
216 PAPER 8 * (r/8 - int(r/8)): CLS
```

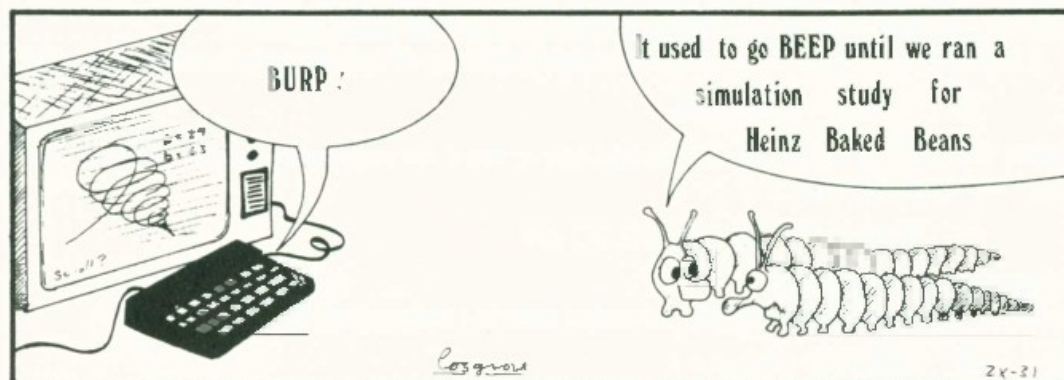
That slows things down somewhat, doesn't it?

Replace 216 with:

```
216 PRINT INK ((t(r) + 3) * 7/11); "□□□";
```

(It's not absolutely necessary to INT the expression; BASIC will do this anyway because INK can't have a non-integer value associated with it.)

Experiment; you can probably invent even more horrific looking displays!



... but why not use the Spectrum to debug itself?

16 Debugging III

In the last example we looked at, the debugging process was achieved by a considerable amount of hard graft on our part. It would be nice to get the machine, whose sole function so far has been to sit there smugly telling us the wrong answer, to do some of the work for us.

So the question is: what kind of things could the machine usefully tell us about the way it is executing a program? There are three main areas to consider:

1. What values does the program get for its variables at various stages in its execution?
2. Where does the program go? (i.e. what lines get executed and in which order?)
3. How often does it go there? (i.e. how many times do particular lines or groups of lines get executed?)

Some machines provide automatic features for printing out at least some of this kind of information, but you can't have everything for a hundred-odd pounds (yet!) so we have to build some extra statements into our programs to provide some of these details. Let's look at the headings above in turn.

PRINTING VALUES OF VARIABLES

It's very easy to print out values of variables anywhere we want. All we have to do is to insert a PRINT statement at an appropriate place in the program. For example, in our averaging program we might insert a line 55:

```
55 PRINT c
```

which would allow us to trace the changes in the value of *c* as the program executes. (In fact it wouldn't be difficult to get the machine to provide a copy of the dry run table which we produced by hand—you might like to try it.)

The real problem here is to choose where to output intermediate values sparingly and sensibly, otherwise you just get reams of numbers which take as long to analyse as a hand-generated dry run.

In the case of the second run of the averaging program, you will recall that an error report was generated at line 100 so we didn't get any values printed out. A sensible first step would be to include a line 95:

```
95 PRINT s, c
```

Don't forget that line 40 will need modifying:

```
40 IF n < 0 THEN GO TO 95
```

Otherwise the new statement will never get executed. That will have the effect of confirming our suspicion that *c* contains zero, but little more. Now insert line 65:

```
65 PRINT c; i; n; s; z
```

so that we get a complete list of all the variables (in alphabetical order for convenience) at the end of every loop. This will create a simplified version of the dry run table which will still show up the salient points.

Incidentally, here's a neat trick: in the process of program testing you may want to remove one of the new lines temporarily to avoid having too many values printed at once. This means typing the whole line in again later, or worse, as in the case of our line 95, editing another line as well, since line 40 will have changed back to

```
40 IF n < 0 THEN GO TO 100
```

if line 95 is removed. There is no need to do this. Simply edit in a REM at the beginning of any line you want to disable. Line 95 becomes:

```
95 REM PRINT s, c
```

Since it is now a remark the machine will ignore it, but branching to line 95 is quite legal! When we want the statement back again, all we do is edit out the REM.

TRACING THE ROUTE

The simplest way of tracing the route a program has taken is to follow every line with a PRINT statement which simply prints out the line number just executed. For example, the averaging program would become:

```
10 LET s = 0
11 PRINT "10"
20 LET c = 0
21 PRINT "20"
30 INPUT n
31 PRINT "30"
etc.
```

Again, we run the risk of generating too much information and not being able to see the wood for the trees, so let's be more selective about this "tracing" procedure. The kind of question which a trace will answer well is "does the program branch when it's expected to?" This being the case, it makes sense to restrict our trace to regions where there are branches.

For instance, suppose that a routine in a program is to input a day of a month. Such a value must be within the range 1-31 so it would be good programming practice to make sure that the user has entered such a value before continuing. We might write a piece of code like:

```
50 INPUT d
60 IF d > 0 OR d < 32 THEN GO TO 200
70 REM Here if invalid day
.....
200 REM Here for a valid day
.....
```

The program doesn't behave as it should, so we insert a trace statement after lines 50, 70 and 200:

```
50 INPUT d
51 PRINT "***50**"
```



```

60 IF d > 0 OR d < 32 THEN GO TO 200
70 REM Here if invalid day
71 PRINT "**70**"
    . . . . .
200 REM Here for a valid day
201 PRINT "**200**";
    . . . . .

```

We find that whatever input value we try, the resulting trace is always:

```
*50* *200*
```

(I'm using asterisks so that trace line numbers cannot be confused with numbers printed out by the program. Any special character that takes your fancy will do.)

So the program cannot be made to get to line 70. There is only one sensible conclusion: the condition $d > 0$ OR $d < 32$ is always met. Logically this is so—any number is either greater than zero or less than 32. What we should have said was:

```
60 IF d > 0 AND d < 32 THEN GO TO 200
```

Confusing AND with OR is a common error, because of the rather sloppy way we use these words in ordinary speech. In this case both conditions must be met before we have a valid day so the AND connective is required.

PROGRAM PROFILES

A program profile shows how many times each line of a program has been executed. As usual, this is overkill, and we should be selective about the parts of a program we want to profile. It's easy enough to do. Suppose that we want to find out how many times line 420 of a particular program is executed. We set up a count to zero at the beginning of the program and then increment it by 1 every time line 420 is passed through:

```

5 LET pc = 0
    . . . . .
420 LET a = a * (p - 1)
421 LET pc = pc + 1
    . . . . .
809 PRINT pc
810 STOP

```

Let's look at a concrete example. The following program is intended to accept a maximum of 20 values, terminated by a zero, and sort them into ascending order. Thus if the input sequence is:

```

3
3
1
4
2
0

```

the resulting output should be

1
2
3
4
8

The zero should not appear since it is only a delimiter.

```
10 DIM a (20)
20 FOR p = 1 TO 20
30 INPUT a (p)
40 IF a (p) = 0 THEN GO TO 60
50 NEXT p
60 LET n = p
65 LET f = 0
70 FOR p = 1 TO n
80 IF a (p) < a (p + 1) THEN GO TO 130
90 LET t = a (p)
100 LET a (p) = a (p + 1)
110 LET a (p + 1) = t
120 LET f = 1
130 NEXT p
140 IF f = 1 THEN GO TO 65
150 FOR p = 1 TO n
160 PRINT a (p)
170 NEXT p
```

The program doesn't quite do the job. (Key it in and try it.) In fact it gets into an endless loop.

So where to start looking? The first loop (20–50) looks harmless enough, and the final one (150–170) is just printing out the contents of the array, a. It seems sensible then to concentrate on the loop from 70 to 130. It is clear from line 80 that sometimes all the statements in the loop are executed, and sometimes those from 90 to 120 are ignored. So we'll have two profile counts, c1 and c2, which count the number of times the loop is entered and the number of times the last part of the loop is executed, respectively.

We can achieve this with:

```
67 LET c1 = 0
68 LET c2 = 0
75 LET c1 = c1 + 1
125 LET c2 = c2 + 1
132 PRINT c1, c2
```


While we are at it, we might as well print out the contents of the array at the end of each loop because it is obvious that numbers are being shovelled about inside it and that very few other variables are being used at all. So:

```
134 FOR q = 1 TO n           [because 1 TO n seems to be the
135 PRINT a (q);             relevant chunk of the array]
136 NEXT q
137 PRINT
```

Let's try a few data sets and see what happens: if we enter

3
6
1
8
5
0

we get:

6 4
316500
6 4
130056
6 2
100356
6 2
001356
6 2
001356
6 1
001356
6 1
001356

and so on, until memory runs out.

Well, it seems to be getting values in order but we've lost the "8" and where did the pair of zeros come from? Also, it goes round the main loop 6 times consistently but the number of times round the subloop decreases steadily until it reaches 1, where it stays.

One of the zeros is obviously the delimiter, and the other one is an element of the array which is not set during the run, but is initialized to zero by the system. In other words, the program is dealing with two values too many. So let's rewrite line 60:

```
60 LET n = p - 2
```

and try yet again. Hope beats eternal . . .

We get

```
4      2
3165
4      2
1356
4      0
1356
1
3
5
6
```

It's progress of a kind: we've got rid of those zeros. But we still haven't got our 8 back.

It's hard to see how it can have got lost. Perhaps it's still there, but not being printed out. Where do we PRINT it? Lines 150–170. The range 1 TO n must be too small. Let's increase it by 1:

```
150  FOR p = 1 TO n + 1
```

And, of course, while we're at it, the trace in line 134 is presumably having the same problems. So let's fix that up too:

```
134  FOR q = 1 TO n + 1
```

Ho hum; once more unto the breach, dear friends, once more . . .
This time we get (for the same data):

```
4      2
31658
4      2
13568
4      0
13568
1
3
5
6
8
```

Great: we've cracked it. It's doing the job perfectly. Or is it? Let's try:

```
3
5
2
1
5
0
```


This time we get:

```
4      3
32155
4      3
21355
4      2
12355
4      1
12355
4      1
12355
4      1
12355
etc.
```

It's getting the right answer, but it never comes out of the loop. We notice that *c2* never gets down to zero in this case so it's a fair bet that this is what terminates the program.

What decides whether the program enters the subloop or not? Line 80:

```
80  IF a (p) < a (p + 1) THEN GO TO 130
```

The difference between the two data sets is that the second has two identical values in it. Since 5 is not less than 5 the subloop will be executed whenever the pair of 5's is encountered. That's why the program always goes round the subloop once. Perhaps the question should be:

```
80  IF a (p) <= a (p + 1) THEN GO TO 130
```

This time everything works.

```
4      2
32155
4      2
21355
4      1
12355
4      0
12355
1
2
3
5
5
```

Now it works like a charm and we can take out the test lines.

I hope I have illustrated a couple of important points here. Firstly, we haven't needed to know exactly how the procedure works. If you've worked through this carefully, it's probably fairly clear by now; and a few dry runs would probably convince you that you understand it. (Dry runs are a good way of understanding how computer procedures work. I've often done a dozen or so on some obscure piece of code—someone else's, of course—before being really clear in my mind about what's going on.) Secondly, there is always a temptation to believe that when a program runs successfully for the first time, the job is done and there's time for a quick pint down the local. As we've seen, the job is *not* done, because there may be other sets of data for which the program fails; and anyway, the pub closed an hour and a half ago, if time goes as fast for you as it does for me when I'm writing code.

*Computers aren't just number-crunchers.
They can crunch characters too; that is, manipulate symbols.*

17 Strings

The postman cometh . . . and there is a letter for you. A very personal letter. "Dear Mr Slugshaver," it says, "You have been selected from among a small group of people living in Lower Pigpen to receive, absolutely free of charge, a magnificent pair of concrete-lined Wellington boots . . ." Very gratifying. But, next door, old Mrs Snagglechest has received *almost* the same letter. In fact, the whole of Lower Pigpen has, along with most of the West Midlands.

Here's how it's done.

```
10 INPUT "What is your name?"; n$
20 INPUT "What town do you live in?"; t$
30 INPUT "What street do you live in?"; s$
40 INPUT "What number is your house?"; h
50 PRINT n$
60 PRINT h; ", □"; s$
70 PRINT t$
80 PRINT "Dear □"; n$; ", "
90 PRINT "□ □ □ you have been selected from a"
100 PRINT "small group of people living in"
110 PRINT t$; "□ to receive"
120 PRINT "absolutely free of charge* a"
130 PRINT "magnificent pair of concrete-"
140 PRINT "lined Wellington boots. We are"
150 PRINT "sure, □"; n$; "□ that you"
160 PRINT "will want to take advantage of"
170 PRINT "this generous offer, and that"
180 PRINT "the other inhabitants of"
190 PRINT t$; "□ will be sick"
200 PRINT "as parrots with envy."
210 PRINT "□ □ □ Yours insincerely,"
220 PRINT "□ □ □ □ □ Milton F. Gnatbender"
230 PRINT "□ □ □ □ □ Dealer's Wry Jest."
240 PRINT, " * Postage £1043.22 extra."
```


RUN this, and INPUT (e.g.) "Gully Ball"; "Wayward Heath"; "Pristmas Crescent"; "666". Try other names and addresses. Hmmm . . .

Now imagine this automatically fed names and addresses from a data bank, and churning out thousands of letters an hour.

The interesting thing, apart from how blatant the whole exercise is, is that absolutely *no computation* is involved. Merely memory, and some very simple manipulation of written text. The computer can manage this because, as well as numbers, it can store *strings*. That's what those dollar signs "\$" signal, though there may be some Freudian significance too in the present context.

A *string* is a sequence of *characters*. The characters are listed on pages 183–188 of the *Manual*, and each has a CODE which we'll talk about in a moment.

Here's a string:

```
stringstringstringstringstring.
```

Here's another one:

```
ab334 * . / > > > > < < - b + + + + + qqj.
```

(If you type the full stops, they're part of the string too!)

A string can be only one character long, such as <, or even *no* characters long!

To assign a string variable, you have to put inverted commas around the string:

```
10 LET a$ = "stringstringstringstringstring"
```

Every string variable must be a single letter followed by the \$ sign. For a string no characters long, LET a\$ = "" works.

Strings will sit up and beg if you know how to ask them to. Consider a single character string like

```
3
```

This can be thought of as:

(a) a number, 3

(b) a string, "3"

and you can switch from one to the other in various ways. Suppose we really do think of it as a string, and assign it:

```
10 LET a$ = "3"
```

Suppose you want to work out $3 + 5$. It's no good asking the Spectrum to do this:

```
20 LET b = a$ + 5
```

```
30 PRINT b
```

It won't work—try it if you don't believe me. Why not? The foolish beast has been told that 3 is to be considered a *string*, and it doesn't know it's a *number* too. But—ahah—we can *convert* it to a number using VAL. Try

```
20 LET b = VAL a$ + 5
```

```
30 PRINT b
```

In general, given a string which happens to be an arithmetical expression:

```
10 LET a$ = "2 + 2 + 5 * 3"
```

the machine does not know it can be worked out as a number; it just thinks of it as a sequence of *characters*

```
2 + 2 + 5 * 3
```

For instance, you can pick out the 6th character in it:

```
20 PRINT a$(6)
```

and get * as your answer. (This can be useful: as an arithmetical expression it equals 19 which doesn't *have* a sixth character.) But if you *want* to convert it to a number, then

```
20 PRINT VAL a$
```

will come up with 19 as your answer.

You can convert a number to a string in two ways. First, by putting inverted commas around it, "3336" or whatever. BUT if you're inside a program, and are working out $a + b$ or whatever, which is 3336, it's no good writing " $a + b$ " and hoping to get "3336" in there; what you get is a three-character string

```
a + b
```

which is rather different.

CHR\$ converts a number between 0 and 255 into a single character according to the list of codes in the *Manual* on pages 183–188. For instance, CHR\$ 96 is the pound sign £. Some numbers aren't used.

CODE goes the other way: CODE "£" is 96. If you try CODE "£335/h" you still get 96: it only looks at the first character.

LEN tells you how long a string is.

By far the most interesting feature of strings is that you can chop bits out of them, called *substrings*. The instruction

```
a$(3 TO 7)
```

picks out the string consisting in turn of the 3rd, 4th, 5th, 6th, and 7th characters in a\$. So if a = \text{"stringstringstringstring"}$ then a(3 TO 7) = \text{"rings"}$. You can use any numbers in place of 3 and 7. And a(5)$ picks out just the 5th character—here "n".

For instance, suppose you want to input a number between 1 and 7 and say which day of the week that gives (taking 1 as Sunday, 2 as Monday, etc.) A clumsy way would be

```
10 INPUT n
20 IF n = "1" THEN PRINT "Sun"
30 IF n = "2" THEN PRINT "Mon"
....
```

until all seven days are accounted for.

But, using substrings:

```
10 LET a$ = "SunMonTueWedThuFriSat"
20 INPUT n
30 PRINT a$(3 * n - 2 TO 3 * n)
```

does the same job in 5 fewer lines; see DAYFINDER, page 121.

You can stick strings end to end using + to join them, like this:

```
"hot" + "dog" = "hotdog"
```

or order them "alphabetically" using <. Consult the *Manual*, and experiment. Careful use of strings can often save a lot of space.

This program takes a name and finds its initials.

```
10 INPUT "What is your full name?"; n$
20 LET n$ = " " + n$
30 FOR i = 1 TO LEN n$
```



```
40 IF n$ (i) = " " AND i < LEN n$ THEN PRINT n$ (i + 1); " ";  
50 NEXT i
```

RUN it, use your own name. INPUT "Ginormous Electronic Corporation" and see if it outputs "G.E.C."

As it stands, it's not perfect: if you put extra spaces between words it gives a rather messy output. All it does is tack an extra space at the front to make life easy; and then assume that each character *following* a space is an initial. For tidiness it inserts dots.

Project

Modify the program so that it ignores repeated spaces. One way is to search through, and delete any spaces that follow a given one. To delete the *j*th character from a string *n\$* you write

```
LET n$ = n$ (TO j - 1) + n$ (j + 1 TO)
```

If you omit the numbers before or after the "TO" the machine assumes they are the front and end, respectively.

*There can be more efficient ways
to define a lot of variables than
using LET statements:*

18 Data

Over the top of key D, in green, is the word “DATA”. By using this instruction you can avoid having to type in long series of statements of the type “LET a (37) = 242”, “LET a(38) = 243”, . . . etc. For example, and purely to familiarize yourself with the command, type in this:

```
10 DATA 1, 2, 3, 4, 5, 6, 7
20 FOR i = 1 TO 7
30 READ x
40 PRINT x
50 NEXT i
```

You should get, as output, the numbers 1–7 again. The READ command is essentially “LET x = the next number on the DATA list”. Every time a READ instruction is met, the computer looks along the DATA lines, finds the last item it read, and inputs the next one as the value of the variable after the READ statement. So first time around it reads x as 1; next time 2; and so on.

A useful feature of DATA is that the machine considers all of the DATA lines in a program as a single list, strung end to end. You can put these lines anywhere you like (though, since the computer searches through them, somewhere near the front tends to be quicker). Try rewriting the program, say as

```
10 FOR i = 1 TO 7
20 READ x
30 DATA 1, 2, 3, 4
40 PRINT x
50 DATA 5
60 NEXT i
70 DATA 6, 7
```

It works just the same—even if the DATA line is inside a FOR/NEXT loop. (DATA is an exception to the rule that lines are executed in numerical order.)

Now, if that were all you could do, it wouldn’t be very impressive. But here’s a more typical use.

```
10 DATA 1000, 500, 1500, 1000, 1000, 1500, 500, 1000, 1000, 500
20 READ x, y
```



```

30 PLOT x, y
40 FOR i = 1 TO 4
50 LET x0 = x: LET y0 = y
60 READ x, y
70 DRAW x - x0, y - y0
80 NEXT i

```

This should draw a diamond shape: the DATA list gives the coordinates of the corners, with the bottom corner repeated for the finish as well as the start.

Let's jazz that idea up into a real cool cat . . .

```

10 DATA 2, 0, 6, 0, 9, 1, 15, 0, 16, 1, 16, 12, 15, 13, 14, 12,
    14, 2, 10, 2, 11, 6, 8, 12, 10, 15, 9, 18, 8, 22, 7, 18, 3, 18, 2,
    22, 1, 18, 0, 15, 2, 12, 0, 6, 0, 3, 2, 0
20 READ x, y
30 PLOT x, y
40 FOR i = 1 TO 23
50 LET x0 = x: LET y0 = y
60 READ x, y
70 DRAW x - x0, y - y0
80 NEXT i

```

If you analyse what this is doing, you'll find it first PLOTS the point with coordinates 2, 0; and then it joins this to the point 6, 0; then 9, 1; and so on. The numbers are successive ones in the DATA list.

To put the picture more towards the middle of the screen so that it doesn't look cramped, change line 30 to:

```
30 PLOT 100 + x, 60 + y
```

The DATA list itself was worked out by making a rough drawing on graph paper, and reading off the coordinates. It's just like the pictures in those dot-to-dot drawing books, except that we have to specify where the dots are using coordinates.

Now, that's still a lot of work for just one cat. For the option of fat cats, thin cats, and even upside-down and back-to-front cats, you can *transform* the data. Change line 30, as above, to make enough room; add this line:

```
5 INPUT a, b
```

and change line 70 to

```
70 DRAW a * (x - x0), b * (y - y0)
```

Now, on RUN, you must input two numbers. Don't be too ambitious at first: try $a = 1$, $b = 2$; and $a = 2$, $b = 1$. Then try $a = 2$, $b = -1$; $a = -2$, $b = -1$. Once you've seen those, try what you like! But be warned: the program has no protection against going off the screen, and fractional values of a and b give funny effects due to round-off errors. For a trick to avoid this problem, see SPIRALS, page 125.

Now let's draw a whole line of cats. To do this, we can use a loop; but we need some way to reset the READ instruction to the front end of the data list. That's what RESTORE does. So we add the lines

```

15 FOR t = 1 TO 8
90 RESTORE
100 NEXT t

```

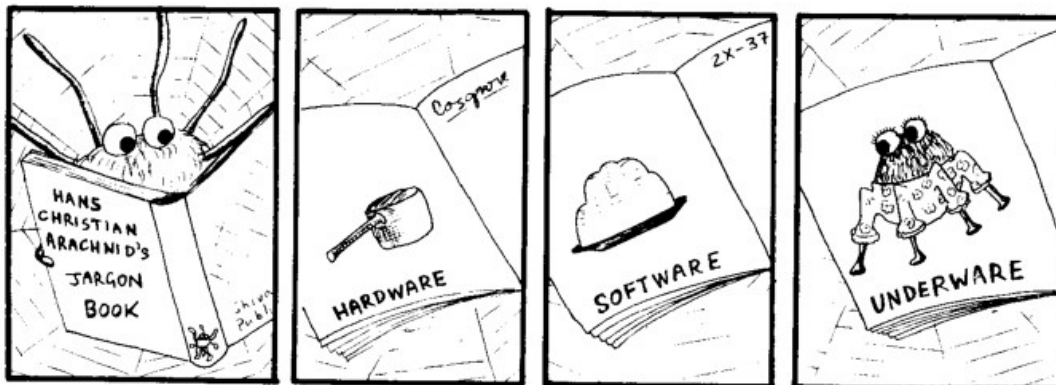
and change line 30 to

```
30 PLOT 50 + x + 20 * t, 50 + y
```

That's all.

Projects

1. Make the cats sit higher up the screen as they move to the right, as if they're sitting on a flight of stairs.
2. Add more DATA to draw the stairs.
3. Change the DATA list, with the aid of an atlas, so that the program draws a map of Australia. Find out what Australia looks like upside down. (Australians think it is.)
4. If you've half an afternoon to spare, set up DATA for a map of the world.



19 Debugging IV

How do we prove conclusively that a program does precisely what it was written to do? I don't want to get involved in too complicated a philosophical discussion (because that's where we are headed) but, broadly, it's a bit like asking an astronomer whether the sun will rise tomorrow. If he is very pedantic he might answer that the earth has been going round the sun for a long time now and we have a body of physical laws which suggest that it will continue to do so in a regular way, and that the smart money would be on this continuing to be the case tomorrow; but he would add that he has no way of knowing whether our physical laws are right and that what we have observed for thousands of years might in fact be a manifestation of a much more complex law whose effect, tomorrow, might be to reverse the direction of the earth's rotation or to take it out of orbit completely.

By analogy, because a program behaves correctly for the first thousand sets of data input to it, there's no absolute guarantee that it will work for the thousand and first. In fact, bugs often don't become apparent until months or even years after a program has been apparently successfully completed and has been run without problems dozens or even hundreds of times. This isn't really surprising; after all, it's the conditions which occur least often that the programmer is most likely to overlook.

Here's an example:

We are writing a suite of programs for the Nether Hopping Electricity Board to handle their customer accounts. They explain to us that there are two tariffs, A and B. On the A tariff the consumer pays a quarterly standing charge of £15 and then pays for units used at a rate of 4p per unit. On the B tariff the consumer pays no standing charge and pays 7p per unit. So we write a piece of code like:

```
100 INPUT t$
105 INPUT units
110 IF t$ = "a" THEN GO TO 300
120 IF t$ = "b" THEN GO TO 140
130 GO TO 5000
140 LET bill = 7 * units/100
150 PRINT bill
160 GO TO 100
300 LET bill = 15 + 4 * units/100
310 PRINT bill
```

```
320 GO TO 100
```

```
.....
```

```
5000 PRINT "Invalid tariff"
```

```
5010 STOP
```

OK. I know the code could be more efficient, and that we would actually need some more information like the consumer's name and account number, but you get the basic idea.

So we test this piece of code and it works fine and we go away muttering that it is a waste of our remarkable talents to be given Noddy programs like that to write.

And it does work fine; for years; and then one day it prints a bill for £0.00. Of course, nobody notices because it's one of thousands of bills and anyway it's probably enveloped automatically. The recipient is puzzled and probably amused by the bill because it shows how stupid computers are but there seems no point in taking any action so he throws the bill away. Unfortunately, we wrote another program in the same suite which stores the date that each bill was despatched, and if it does not receive confirmation that the bill has been paid within 28 days, it prints a final demand notice. This time the recipient is more irritated than amused but he just consigns it to the waste paper basket, as before. At this point things start to go visibly wrong. The routine which checks the delay between presentation of the account and payment issues an order to the maintenance department to cut off the consumer when he still hasn't paid after 60 days.

What's happened? Easy!! The consumer is an old-age pensioner who has taken advantage of one of the long winter break packages that the travel companies offer to senior citizens. He was out of the country for just over three months and used no electricity in a full account period. He is also an unusually frugal user of electricity so he's on tariff B. That's why the system printed out a request for zero payment, and of course it won't happen very often because very few people will be away from home for that long, and tariff B users are likely to be thin on the ground, too. For the problem to occur, the consumer has to fit both conditions.

Once seen, the bug is easily squashed:

```
145 IF bill = 0 THEN GO TO 100
```

so that the PRINT statement is avoided. This problem is supposed to have occurred in an early computer system, although whether it's a folk tale I wouldn't like to say. In any event, I think it illustrates neatly how a bug can lie dormant almost indefinitely.

The moral is: when you invent data to test a program, don't do so at random. Choose values at and close to branch values in the program. If a statement says:

```
305 IF u < 30 THEN GO TO 500
```

then run a test with u at 29.999 and another at $u = 30.0001$. You may have meant:

```
305 IF u <= 30 THEN GO TO 500
```

If you only test at $u = 15$ and $u = 160$ you won't notice the error.

Make sure test data have been chosen so that every section of the program gets executed at some time. And of course, make sure you know exactly what the answer should be for each set of test data.

For the more mathematically minded only!

20 Curve Plotting

You've probably met the idea of a *graph* specified by coordinates; but just in case, let's recap. Start with two lines, the x-axis and the y-axis, at right angles to each other. We can mark off distances x and y along these (marking negative numbers to the left on the x-axis and downwards on the y-axis) and use these two distances to specify a point with *coordinates* x and y . It's just like pixels (Figure 20.1).

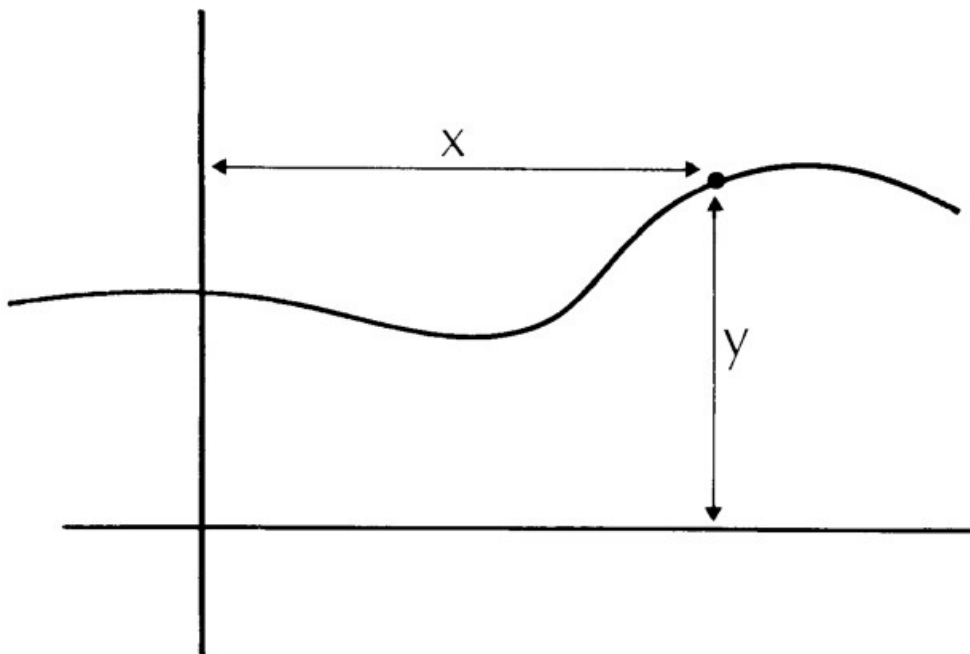


Figure 20.1 Coordinates for curve-plotting

If we now imagine x varying along the x-axis, and the value of y changing in some way that depends on x , then the point with coordinates (x, y) will move too; and it generally traces out a *curve*. If you give a formula for how y depends on x , say $y = x^2 - 3$, then you give a formula for that curve. (This idea, due to Descartes in 1637, lets you use algebra to study geometric curves: it's the starting-point for calculus.)

Using PLOT instructions we can draw similar curves:

```
10 FOR j = 0 TO 255
20 PLOT j, j/2
30 NEXT j
```

You should get a line of pixels climbing up the screen from lower left to top right. A mathematician would call this the graph of the function $y = x/2$.

You can modify this to give an endless variety of graphs, just by changing $j/2$ to some other expression involving j . For instance, to plot the square root of j as a function of j all you have to do is change the $j/2$ to $\text{SQR } j$, getting:

```
10 FOR j = 0 TO 255
20 PLOT j, SQR j
30 NEXT j
```

That's a bit more interesting, isn't it?

Experiment. Change that $j/2$ again. Try these.

```
(Parabola)      20 PLOT j, .002 * j * j
(Sine curve)    20 PLOT j, 80 * SIN (j/20) + 80
(Cosine curve)  20 PLOT j, 80 * COS (j/20) + 80
(Catenary)      20 PLOT j, (EXP (.02 * (j - 120))
                  + EXP (.02 * (120 - j)) ) * 10
```

... hang on, why are these so complicated? What *is* he up to?

There are problems if you try any old expression in j . Namely, what will fit on the screen. The value of the vertical coordinate has to be between 0 and 175, or else the Spectrum loses its paddy and shuts up shop. (The poor old thing can't PLOT anything else, and it resents being asked to.) So you need to adjust the *scale* to make things fit. You'll soon see why if you try

```
20 PLOT j, j * j
20 PLOT j, SIN j
20 PLOT j, COS j
20 PLOT j, EXP (j) + EXP (-j)
```

There are ways round this problem, of course—see below, under SCALE. But before you do, here are some more interesting functions that *do* fit, because I've carefully chosen them to.

```
20 PLOT j, EXP (-j/80) * SIN (j/8) * 80 + 80
20 PLOT j, 80 + 80 * COS SQR (2 * j)
20 PLOT j, ABS (j - 127)
20 PLOT j, 80 + 60 * LN (1 + ABS SIN (j * .125) )
20 PLOT j, 12 * ABS (j/4 - 30) ↑ .666
20 PLOT j, 40 * ABS (j/4 - 30) ↑ .25
20 PLOT j, 160 * EXP (-.1 * (j/4 - 30) * (j/4 - 30) )
```

To avoid writing complicated expressions, you can build up the formula in stages, like this:

```
20 LET t = j/24
25 PLOT j, 120 + .1 * t * (t - 2) * (t - 4) * (t - 6) * (t - 8) * (t - 10)
```


SCALE

For the moment we'll work just with functions defined for *positive* numbers and taking *positive* values: a good one to use is SQR, the square root. Take the graph-plotting program above, and change line 20 to the following in turn:

- (a) 20 PLOT j, SQR j
- (b) 20 PLOT j, 2 * SQR j
- (c) 20 PLOT j, 4 * SQR j
- (d) 20 PLOT j, 6 * SQR j
- (e) 20 PLOT j, 8 * SQR j
- (f) 20 PLOT j, 10 * SQR j
- (g) 20 PLOT j, 12 * SQR j
- (h) 20 PLOT j, 14 * SQR j

You'll notice pretty quickly that in successive graphs, everything goes higher up the screen—and in fact, in (g), the machine stops because graph points start to go off the screen altogether. The greater the “something” in (something) * SQR j, the more the graph is stretched in the vertical direction. This “something” is a *scale factor*, and by adjusting it you can make graphs fit the screen neatly.

If the scale factor is too small, you get graphs that are so squashed up you can't see anything. Try

```
20 PLOT j, .1 * SQR j
```

If the function you're plotting grows too large, you can get it back on the screen by adjusting the scale factor. For example

```
20 PLOT j, j * j
```

goes off the screen because $14 * 14 = 196$, which is already too big. In fact the largest number you need to plot is $255 * 255 = 65025$. If you divide this by 400 you get 162.5625, which fits neatly into the 175 that is permissible. So you get a nice graph provided you use $1/400$ as the scale factor:

```
20 PLOT j, j * j / 400
```

There is a fairly obvious general rule which makes sure the scale factor is chosen suitably. Suppose that the largest value the function takes as j runs from 0 to 255 is m (for maximum). Then with scale factor s, everything is fine provided $s * m$ is no larger than 175—and preferably close to it so that the graph isn't too squashed. In particular, you can make $s * m = 175$, by setting $s = 175/m$. (For round figures $160/m$ may be better; and you need a reasonable estimate for m rather than an exact number.)

In fact, you could write a program to work out m. Suppose we stick to the $j * j$ function; then this will do the trick:

```
10 LET m = 0
20 FOR j = 0 TO 255
30 LET q = j * j
40 IF q > m THEN LET m = q
50 NEXT j
```

As it stands this won't PLOT the graph, though; so now you add the plotting routine overpage:

```

60 FOR j = 0 TO 255
70 PLOT j, (175/m) * j * j
80 NEXT j

```

One defect: you have to do all the calculations *twice*. This is hard to avoid efficiently, unless you know *which* j gives the largest value for $j * j$. In this case $j = 255$ obviously does, but it's not always easy to see in advance. (You *can* dimension a vector $v(i)$ of size 256, store the values of $j * j$ as $v(j + 1)$, and use these for the PLOT; but vectors and arrays take up a lot of memory! See the *Manual* for further information on vectors and arrays.)

As well as scaling the vertical axis, you can scale the horizontal axis. The function $\text{SQR } j$ or $j * j$ doesn't show this very vividly, so I'll use $80 + 80 * \text{SIN } j$, which does. Try the following:

```

20 PLOT j, 80 + 80 * SIN (.05 * j)
20 PLOT j, 80 + 80 * SIN (.1 * j)
20 PLOT j, 80 + 80 * SIN (.15 * j)
20 PLOT j, 80 + 80 * SIN (.2 * j)
20 PLOT j, 80 + 80 * SIN (.25 * j)

```

This time the *horizontal* scale changes—but the scale factor acts rather differently (did you notice?). The greater the scale factor, now, the more the graph is squashed up in the horizontal direction: you get more wiggles in the curve. Why?

When j ranges from 0 to 255, the number $.05 * j$ ranges from $.05 * 0$ to $.05 * 255$, that is, from 0 to 12.75.

When j ranges from 0 to 255, $.1 * j$ ranges from $.1 * 0$ to $.1 * 255$, that is 0 to 25.5.

So, in the second case, *twice* the range of values gets squashed into the same horizontal space.

In fact, with a scale factor s —that is, using

```
20 PLOT j, 80 + 80 * SIN (s * j)
```

you plot the range from 0 to $s * 255$ in the width of the screen. The greater s is, the larger the range, so the more squashed it becomes.

So, if you want to plot over a chosen range, say 1000, then you want $s * 255 = 1000$, that is, $s = 1000/255$. In general, if you want the range 0 to n , you'll need a scale factor $n/255$.

To sum up:

$$\text{best vertical scale factor} = \frac{175}{\text{largest value being plotted}}$$

$$\text{best horizontal scale factor} = \frac{\text{top of range of values of variable}}{255}$$

Project

If you don't know what's going to look best, you can write an "interactive" program which lets you choose the two scale factors (via INPUT) and then plot the graph; if you don't like the result you run it again and change the scale.

Write such a program for the function $80 + 80 * \text{SIN } j$.

Hint: if h is the horizontal scale factor and v the vertical, the operative program line is

```
20 PLOT j, v * (80 + 80 * SIN (h * j) )
```


SHIFTING AXES

You may be wondering: “why all those $80 + 80 * \text{SIN } j$ ’s?” Or maybe, “That’s all very well, but what if the numbers are negative?” The answer is the same to both. The way to deal with negative numbers is to shift your axes, as Eric the Viking once said.

Try this program.

```
10 INPUT s
20 FOR j = 0 TO 255
30 PLOT j, s + SQR j
40 NEXT j
```

Try inputting $s = 0, 10, 20$, etc.

What you’ll see is the same graph, but moving higher up the screen according to the value of s . There are two ways to think of this.

One is that you’re plotting different functions, like $5 + \text{SQR } j$ or $10 + \text{SQR } j$.

The other is that you’re always plotting $\text{SQR } j$, but that the position of the x-axis on the screen changes. See Figure 20.2, which is pretty much self-explanatory.

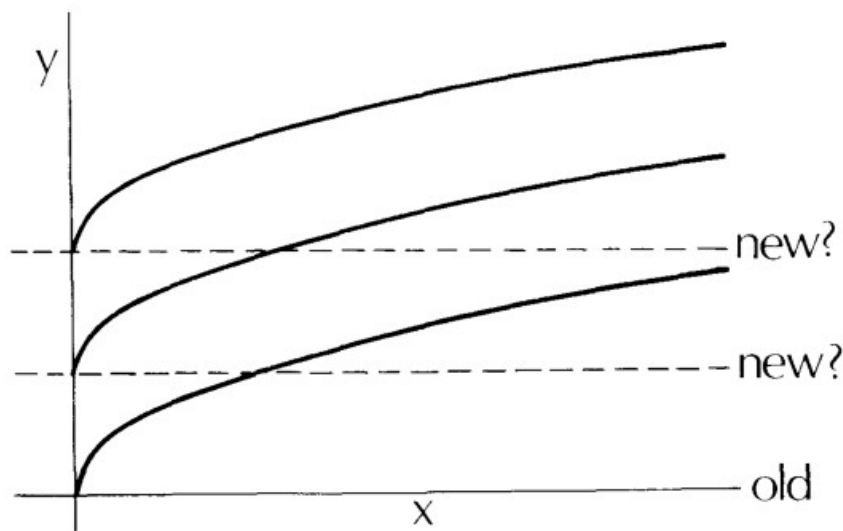


Figure 20.2 Adding a constant is the same as moving the x-axis

To get a clear sine curve, for instance, you need to replace the $\text{SQR } j$ above by $80 * \text{SIN } (.1 * j)$; and pick s to get everything central on the screen. $s = 80$ works beautifully; hence all those $80 + 80 * \text{SIN}$ s.

That gets the x-axis moving. To get the y-axis moving, you can change the range of j , say from -120 to 135 instead of 0 to 255 .

You can combine these shifts with scalings in both directions; and you don’t have to keep the axes central—it’s up to you. I did in fact write out a detailed specification of how to do it to get the best possible display on the screen for a given curve over a given range; then I took one look at the mass of algebra that I’d written, and chucked the lot away again. There are times when mathematics hides the wood behind the trees, and this is one of them.

Instead, I suggest you experiment, using the following program:

```
10 INPUT a, b, c, d
20 PLOT 0, d
30 DRAW 255, 0
40 LET u = -b/a
50 PLOT u, 0
60 DRAW 0, 175
100 FOR j = 0 TO 255
110 PLOT j, c * SIN (a * j + b) + d
120 NEXT j
```

Here a, b, c, d shift axes and scale, while the x-axis and the y-axis are drawn in. I haven't protected against crashes if the function goes out of range—apart from laziness, this drives home the need to take care in choosing scale and position of axes.

(Try $a = .1$, $b = -10$, $c = 80$, $d = 80$. Negative b is generally necessary.)

OTHER TECHNIQUES

There are other ways of using PLOT and DRAW to get curves. Rather than overload this chapter with technicalities, I've put examples of some of these in the Prepacked Programs. See LISSAJOUS FIGURES, SPIRALS AND ROSETTES, and GRAPHICS DEMONSTRATION 1 and 2.

Sometimes numbers that look the same . . . aren't!

21 Debugging V

The kinds of bugs we've looked at so far have been of our own making, and have been reasonably easy to cure once we have seen them. There's another kind of bug which is caused by the design of the machine itself. It's not a design fault but a consequence of the way all computers are organized. It's to do with the precision with which computers store numbers. If we think about any common way of holding numbers it's obvious that there is a limit to the number of digits that can be held. A car mileometer, for example, can only hold 5 digits because it only has 5 "windows". It's just the same with a computer. Each number can occupy no more than a fixed number of "windows". However, each window does not represent a decimal digit. The internal machine code for numbers is quite different from the way we think about them, and I will not bore you with the gory details. The fact that there is inherent inaccuracy *and* a code conversion being employed means that the external representation of a number (as it is displayed on the screen) may not be quite the same thing as the internal representation. I'll give you an example of what I'm talking about from school logarithms. If you multiply 2 by 2 using logs you get:

No.	Log.
2	0.3010
2	0.3010
3.999	0.6020 +

i.e. $2 \times 2 = 3.999!$

The combination of the fact that the logs are only accurate to 4 figures (i.e. they are only allowed to occupy 4 windows) and that a code conversion (number to logarithm, logarithm back to number) is taking place, creates the inaccuracy.

Here is a program which causes the same kind of problem:

```
10 FOR p = 0 TO .3 STEP .01
20 LET q = ATN (TAN (p) )
30 IF p < > q THEN PRINT p, q
40 NEXT p
```

At line 10, we take the tangent of p and then immediately invert the process by taking the arctangent of the result. In other words, q should contain the same value as p. So line 30 should never have the effect of printing p and q because they are always equal. When the program is run we get the following output (see over):

0.02	0.02
0.03	0.03
0.04	0.04
0.05	0.05
0.07	0.07
0.09	0.09
0.11	0.11
0.12	0.12
0.13	0.13
0.14	0.14
0.16	0.16
0.18	0.18
0.2	0.2
0.21	0.21
0.22	0.22
0.26	0.26
0.28	0.28

This is a very strange result indeed, because the machine is not only printing out values, and so claiming they are different, but it is also printing them as if they were the same! What has happened is that the complex mathematical processes involved have led to slight inaccuracies in the internal representation of the numbers, which have accounted for the differences between p and q . However, there are also inaccuracies involved in decoding the internal format back to the decimal numbers displayed on the screen so that they appear to be identical although the machine is adamant that they are not. Note that for some values the internal codes *are* identical: for 0.06 and 0.08, for example.

This kind of error can be extremely puzzling and sometimes the only way out of it is to allow a small error in the IF statement so that we have:

IF ABS ($p - q$) < 0.000001 THEN . . .

The ABS function is necessary because q might be greater than p . For example, if $p = 3$ and $q = 3.1$ then $p - q = -0.1$, which is less than 0.000001 so the condition would be met if the ABS function (which effectively chops off the minus sign) were not there. $\text{ABS}(-0.1) = 0.1$ which is greater than 0.000001 and so the condition is not met, which is what we wanted.

When you come to write longer programs, to do more complicated things, it's important to keep a clear head. This program, ANTI-MISSILE SCREEN, produces a playable game and illustrates

22 Programming Style

First things first: the program. You can treat it as a prepacked program if you wish: copy it and RUN.

It starts by displaying two cities and two anti-missile silos. The silos are referred to as "1" and "2" from left to right.

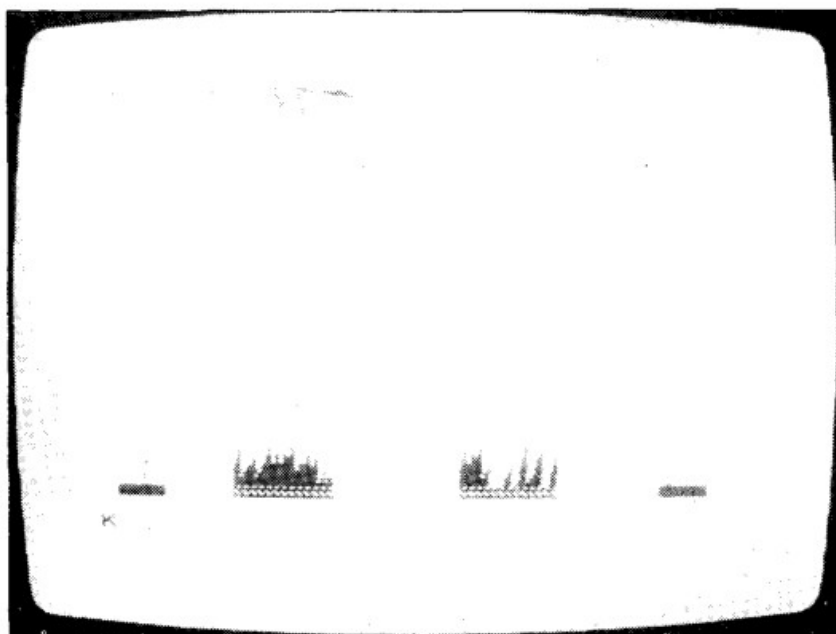
Missiles now rain down on you in a fairly unpredictable fashion, but the track of any given missile has a fixed angle.

Hit "1" or "2" on the keyboard to activate one of your defence silos. (The program can take a little while to respond to this, so hold the key down until it does.) You will then be asked for a range and angle for the intercepting missiles. If you hit an incoming missile, this is indicated by the warhead no longer flashing.

Sooner or later you will run out of missiles (there are 20 in each silo to start with), or your silos or cities will be destroyed. The object of the game is to shoot down as many missiles as possible before this happens.

The "range" and "angle" values you need to enter need explaining.

Firstly, the range value uses the same coordinate system as the Spectrum in high-resolution graphics mode (PLOT). So if you want to hit something at the top of the screen immediately above a silo, the range is 175. From bottom left corner to top right corner is a total distance of about 300.



Secondly, the angle (measured in degrees) is taken from the horizontal to the right of the referenced silo. This means that for silo 2 the most useful angles are between 90° and 180° , so be careful! (For silo 1, however, 0° to 90° is the useful sweep.) One of the features of the program is that it doesn't tell you if you make a mistake such as firing a missile off the screen. It will simply deduct a missile from your stock and not plot any trace of it.

Here's the listing, in full.

ANTI-MISSILE SCREEN

```
1  RANDOMIZE
10  LET display = 5000: LET genmis = 10000
20  LET movmis = 15000
30  LET fire = 25000: LET prnsr = 30000: LET hitest = 35000
40  DIM m (3, 20): DIM s (2)
50  LET city = 2: LET s (1) = 20: LET s (2) = 20
60  LET kilsilo = 40000: LET kilcity = 45000
70  LET score = 0
80  LET kbhit = 50000
100  GO SUB display
110  IF city < 1 OR s (1) + s (2) = 0 THEN GO SUB prnsr
120  GO SUB genmis
130  GO SUB movmis
160  GO TO 110
500  LET xs = 8
510  FOR I = 1 TO 2
520  FOR x = xs TO xs + 20
530  PLOT x, 0
540  DRAW 0, 4
550  NEXT x
560  LET xs = 224
570  NEXT I
580  LET xs = 56
590  FOR I = 1 TO 2
600  FOR x = xs TO xs + 36 STEP 3
610  LET h = INT (RND * 10) + 3
620  FOR y = 0 TO h
630  PLOT x, y
640  DRAW 3, y
650  NEXT y
660  NEXT x
```



```

670 LET xs = 144
680 NEXT I
690 RETURN
1000 FOR I = 1 TO 5
1010 LET x = INT (RND * 100)
1020 LET a = (RND * PI/2) + .01
1030 FOR p = 1 TO 20
1040 IF m (3, p) = 0 THEN LET m (1, p) = x:
      LET m (2, p) = 175: LET m (3, p) = a: GO TO 1060
1050 NEXT p
1055 GO SUB kbhit
1060 NEXT I
1070 GO SUB kbhit
1080 RETURN
1500 FOR p = 1 TO 20
1505 IF m (3, p) = 0 THEN GO TO 1580
1510 PLOT FLASH 0; m (1, p), m (2, p)
1520 LET xo = 20 * COS m (3, p)
1530 LET yo = -20 * SIN m (3, p)
1540 IF m (1, p) + xo > 255 OR m (2, p) + yo < 0
      THEN LET m (3, p) = 0: GO SUB hitest: GO TO 1580
1550 DRAW xo, yo
1560 LET m (1, p) = m (1, p) + xo
1570 LET m (2, p) = m (2, p) + yo
1573 PLOT FLASH 1; m (1, p), m (2, p)
1575 GO SUB kbhit
1580 NEXT p
1585 GO SUB kbhit
1590 RETURN
2500 INPUT "range, angle: "; rg, ag
2505 IF s (VAL d$) = 0 THEN RETURN
2510 IF d$ = "1" THEN LET xb = 18
2520 IF d$ = "2" THEN LET xb = 234
2523 PLOT xb, 0
2525 LET s (VAL d$) = s (VAL d$) - 1
2530 LET xf = rg * COS (ag * PI/180)
2540 LET yf = rg * SIN (ag * PI/180)
2545 IF xf + xb < 0 OR xf + xb > 255 OR yf < 0 OR yf > 175 THEN RETURN
2550 DRAW xf, yf
2560 FOR p = 1 TO 20

```

```

2570 IF ABS (xf + xb - m (1, p) ) > 15 OR ABS (yf - m (2, p) ) > 15
    THEN GO TO 2700
2610 LET score = score + 1
2625 PLOT FLASH 0; m (1, p), m (2, p)
2630 LET m (3, p) = 0
2700 NEXT p
2710 RETURN
3000 CLS
3010 IF city < 1 THEN PRINT AT 10, 2: "Cities destroyed"
3020 IF s (1) + s (2) = 0 THEN PRINT AT 10, 2: "No anti-missiles left"
3030 PRINT AT 12, 5; score; " missiles shot down"
3040 GO TO 9999
3500 LET xt = m (1, p) + xo
3510 IF xt > 7 AND xt < 29 THEN LET xs = 8: GO SUB kilsilo
3520 IF xt > 223 AND xt < 245 THEN LET xs = 224: GO SUB kilsilo
3530 IF xt > 55 AND xt < 93 THEN LET xs = 56: GO SUB kilcity
3540 IF xt > 143 AND xt < 181 THEN LET xs = 144: GO SUB kilcity
3550 RETURN
4000 IF xs = 8 THEN LET s (1) = 0
4010 IF xs = 224 THEN LET s (2) = 0
4020 PRINT AT 21, xs/8; "□ □ □"
4030 RETURN
4500 LET city = city - 1
4510 FOR r = 19 TO 21
4520 PRINT AT r, xs/8; "□ □ □ □ □"
4530 NEXT r
4540 RETURN
5000 IF INKEY$ = " " THEN RETURN
5010 LET d$ = INKEY$
5015 IF CODE d$ < 49 OR CODE d$ > 51 THEN RETURN
5020 GO SUB fire
5030 RETURN

```

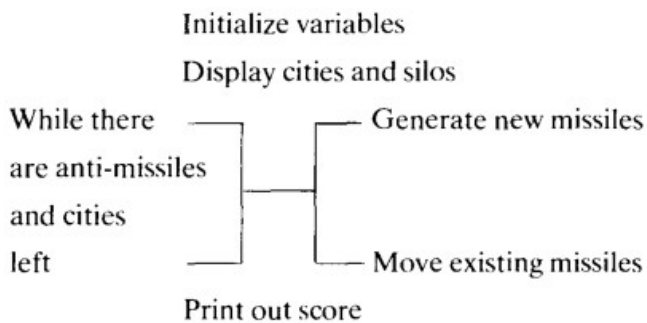
BREAKING IT DOWN

Most of the programs in this book are pretty short and fairly easy to follow, given practice. This one is a little more serious (in program content rather than intention) and it's worth a little study because the technique used to write it is a common and powerful one, known variously as *top-down analysis* or *step-wise refinement*.

The idea is that we start by breaking the program down into a sequence of major steps. Then we examine each step and decide whether it can easily be coded directly; or, alternatively, if it is worth breaking it down into smaller steps. We continue this process

until all the steps are small enough to code without difficulty. Each step at each level becomes a subroutine.

In this program, the first stage of the breakdown looks something like this:



Looking at the listing, you can see that the initializations are done between lines 1 and 80. You might then expect to find something like

```
100 GO SUB 500
```

to enter the display routine; but you'll see that it's actually written as

```
100 GO SUB display
```

and that "display" is a variable which is set to 500 in the initialization area. This makes the program more readable and is particularly handy when you're debugging. For example, once the program has been run once you can write

```
LIST display
```

instead of having to remember

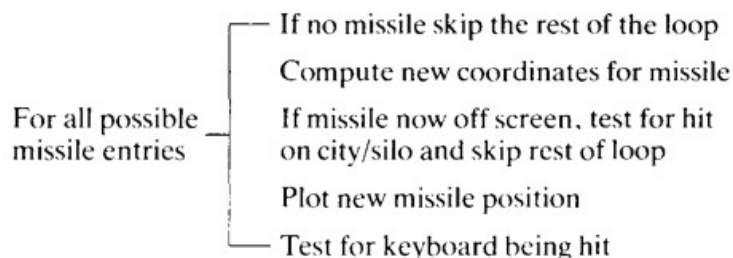
```
LIST 500
```

when you want to examine that particular routine.

The lines between 110 and 160 simply model the "while" loop in the skeletal program description. You can see that there are two subroutine calls in the loop; one to generate new missiles (genmis) and one to move missiles (movmis), as you'd expect.

Let's look at "movmis", for instance. In order to move a missile we have to know where it is. In fact there can only be 20 missiles at any given time, and 3 pieces of information are held about each (its x-coordinate, y-coordinate and attack angle) in an array called m. By convention, if an attack angle pigeonhole contains zero, then there isn't a missile there. That way we only need to test one out of the three parameters to see if there's a missile; and later on, when we want to destroy one, we need only reset the angle to zero to indicate its demise.

The breakdown inside "movmis" then looks like this:



If you look at the listing, you'll see that two of the blocks are written as subroutines: the "test for hit on city/silo" (called hitest) and the "test for keyboard being hit" (called kbhit). This last routine is there because we need to know whether the user has hit the keyboard recently to activate a silo.

Now look at kbhit:

If no key hit, then RETURN

If key hit not 1 or 2 then RETURN

Fire anti-missile

and you'll see that "fire" is another subroutine.

Can you see how this kind of approach and analysis allows us to worry about only a little at a time? That's what appeals to my natural lethargy.

I'll leave you to complete the job of re-creating the skeletons of genmis, fire, and prnsr.

REM: TELL THEM ABOUT REM

When developing a program, the REM statement is an important aid to memory. The computer ignores all REM statements, except during a LIST; so you can put little messages to yourself to remind you why a particular command is where it is. I haven't actually used many REMs in this book; the reason is that all of the programs are accompanied by detailed discussion anyway. But, when you're writing programs on your own, REMs are a godsend. You can use tricks to make REMs stand out in a listing: put in a line of stars REM*****, or use control characters to get them printed in a different colour (see the *Manual*, p. 114).

MODIFICATIONS AND IMPROVEMENTS

1. At the moment, when you activate a missile silo, the program does not confirm which silo it is. Arrange for the activated silo to flash (and to deactivate a few seconds after firing).
2. Missile and anti-missile tracks look identical. Give them different colours. (NB: colour peculiarities will mean that drawing new tracks may change the colours of parts of old tracks. There is no way round this, but it doesn't matter.)
3. In more sophisticated versions of this game, the incoming missiles are MIRVs, and split up into multiple warheads as they descend. It's not difficult to do this, but it needs some careful thought!
4. The destruction of the cities is rather clinical (they're literally wiped out!). How about displaying a mushroom cloud, and then some ruins?
5. Think about sound-effects. (But they will slow the game down a lot.)
6. Because kbhit is called fairly infrequently, you can hold down "1" or "2" for quite some time before getting a response. Experiment with other places to put "GO SUB kbhit" in the program, to improve the response time without slowing the rest of the display too dramatically.

The internal organization of the Spectrum is not something you usually need to worry about. But, if you want to, you can get the machine to tell you what it's doing —and you can change it to further your own ends.

23 Peek and Poke

I could probably write another book the size of this one on PEEK and POKE. And most of it would be far too technical to be useful. But it seems a shame to avoid these extremely important features altogether. By exploring PEEK and POKE you can really learn a great deal about the way computers operate. So, obviously, what you need is some idea of how to get started: how to talk the Spectrum into giving away some of its innermost secrets.

On page 3 I mentioned that computers store information as sequences of 0's and 1's. Each such 0 or 1 is called a *bit* (short for *binary digit*). You can think of a string of 0's and 1's as a number in the *binary system*, which is just like the decimal system except that in place of units, tens, hundreds, thousands, etc. we use units, twos, fours, eights, and so on. In the Prepacked Programs you'll find a program on binary/decimal conversion which explains this a little more.

We don't really need to know about binary; but we *do* need to recognize that computers work with *bits*. In fact, they generally carry their bits around in chunks, called *bytes*. One byte is a sequence of eight bits. So 10110001 and 00111011 are typical bytes.

There are 256 possible different bytes; if you convert to decimal you get the numbers from 0 to 255. You may recognize these numbers: the CODEs of Spectrum characters are numbers between 0 and 255. And indeed, each character is represented by exactly one byte.

A program (and some of the steps performed in carrying it out) is just a sequence of characters; so the machine can store it as a sequence of bytes. To make sure everything stays in the right order it gives each byte a reference number, called its *address*. So you should think of the program as living outside the machine in some form rather like

Address	Byte
1	11001100
2	01110000
3	00000000
4	11101111
.....	

Unfortunately, the machine "architecture" makes a scheme like this a little too simple.

The Spectrum consists of sets of microchips, known as the

ROM	(Read Only Memory)
RAM	(Random Access Memory)
CPU	(Central Processing Unit)
SCL	(Sinclair Logic Chip)

The ROM stores the BASIC interpreter; the RAM stores your program and anything that needs to be worked out while running it; the CPU does the arithmetic and the logic; and the SCL organizes how the others fit together.

We are not interested in the CPU or the SCL, but we *are* interested in the ROM and the RAM. And we can find out exactly which bytes are stored in which addresses in the ROM and RAM by using PEEK.

An instruction

PRINT PEEK 837

for example, will PRINT out the byte stored in address 837 (it's 40, or 00101000 in binary). It prints out the corresponding decimal, in fact; so there's no harm in thinking of a byte as just a number between 0 and 255.

For useful PEEKing, where should we look?

The ROM addresses run from 0 to 16383. There are certainly some good reasons for PEEKing the ROM—you can find out how the Spectrum tells the TV to PRINT a particular character, and play games (like printing it out four times the usual size) with this.

But more interesting are the RAM addresses. These start at 16384; and those up to 23754 (or more if the microdrive is attached) are set and used by the machine. Your program is placed in an address that is stored in the system variable PROG, which itself is in addresses 23635 and 23636. In fact the start address for the program is

PEEK 23635 + 256 * PEEK 23636

Without the microdrive, this ought to be 23755. Try it and see.

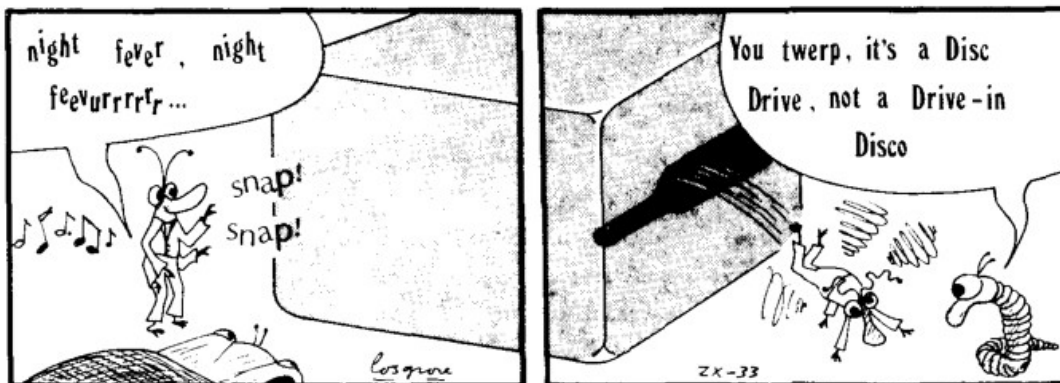
To find out what's stored in RAM, make the Spectrum do the work. (Always do this if you can!) Here's a program to do the job.

```
1000 LET q = PEEK 23635 + 256 * PEEK 23636
1010 LET r = PEEK q
1020 IF r > 23 THEN PRINT q; "□ □"; r; TAB 12; CHR$ r
1030 IF r <= 23 THEN PRINT q; "□ □"; r; TAB 12; "?"
1040 LET q = q + 1
1050 GO TO 1010
```

I've used large line numbers because the idea is to write a test program in front of this, and to see where and how it is stored by using GO TO 1000.

Type the above in; and also a test program:

```
10 REM start
20 PRINT AT 0, 10; "***"
```



Now hit GO TO 1000 and watch. You'll get a print-out like this:

23755	0	?
23756	10	?
23757	7	?
23758	0	?
23759	234	REM
23760	115	s
23761	116	t
23762	97	a
23763	114	r
23764	116	t
23765	13	?
23766	0	?
23767	20	?
23768	23	?
23769	0	?
23770	245	PRINT
23771	172	AT
23772	48	0
23773	14	?
23774	0	?
23775	0	?
23776	0	?

Now it's asking about scrolls, but let's take a look at what's on the screen.

We can see our program—or most of it—building up in the third column: REM, s, t, a, r, t, . . . , PRINT, AT, 0. . . . But there's some other junk too. The third column doesn't seem to help understand this; but the second does. For instance in address 23756 we find 10, and in 23767 we find 20: presumably these are the line numbers. (It's a bit more peculiar than it looks with the lower line numbers; and the 7 in 23757 and the 23 in 23768 are actually the number of characters in the relevant lines: see the *Manual*, page 166. But at least most of it makes good sense straight away.)

The question-marks in column three are there for good reasons: if you leave out "IF r > 23" in line 1020, and delete 1030, the program won't work. This is because those characters are *control* characters (e.g. setting colours) and the machine attempts to carry them out as commands, getting nonsense every so often, and stopping with an error message. But you can work out what they are from the character table in the *Manual*, pp. 183–188. For instance, in address 23765 we have character number 13, which is ENTER—oh yes, we *did* press it then, didn't we?

So, there are some details to puzzle out; but we can see where the program is stored, and it's really *in* there. To see more of it, hit "y" for a scroll (a feature that makes our PEEKing program extremely useful). Now we get

23777	0	?
23778	0	?
23779	44	,
23780	49	1

23781	48	Ø
23782	14	?
23783	Ø	?
23784	Ø	?
23785	1Ø	?
23786	Ø	?
23787	Ø	?
23788	59	;
23789	34	“
2379Ø	42	*
23791	34	”
23795	13	?

and you can go on listing, scrolling every so often, for a considerable time. Soon the routine on line 1ØØØ starts listing out itself . . .

Try a few other programs in lines 1Ø, 2Ø, 3Ø, etc. Leave lines 1ØØØ–1Ø4Ø as they are, and PEEK your programs using GO TO 1ØØØ. You’ll soon see the beginnings of patterns to the listing.

One thing that’s obviously peculiar is the way *numbers* are stored. The 1Ø in PRINT AT Ø, 1Ø, for example, seems to occupy addresses 2378Ø–23787. This looks a big space for such a small number; but it’s related to the fact that the Spectrum can handle *floating-point* arithmetic (decimals like 23.567) and use a code that is suitable. You can have hours of fun just trying to work out how a given number is actually stored.

But the real beauty of it all is that now you know *where* a given program byte lives, you can *change* it. The instruction that gives you this terrible power over the poor Spectrum is

POKE

Let’s see it in action.

What does our little program above do? It PRINTs out * at position Ø, 1Ø. RUN and see.

Let’s POKE it. Add two program lines:

```
3Ø POKE 2379Ø, 96
4Ø GO TO 1Ø
```

Now run it by typing

```
GO TO 3Ø
```

Instead of * you get £ printed out.

Why? Because line 3Ø POKes into address 2379Ø (where the original * was stored) the new character with code 96, which is £.

If you ask the machine to LIST you’ll find that line 2Ø of the original program now reads

```
2Ø PRINT AT Ø, 1Ø; “£”
```

Let’s experiment a little further. BREAK, and get rid of lines 3Ø and 4Ø. Change line 2Ø back to its original form using EDIT. At the same time, change the Ø to 1, so you have

```
2Ø PRINT AT 1, 1Ø; “*”
```

Now GO TO 1000 and PEEK it all again. Scroll to carry on when you run out of screen. You'll spot the change. Addresses 23772 and 23776 have become

23772	49	1
23776	1	?

Otherwise all is as before.

To test our new knowledge, add the following program lines.

```
30 POKE 23772, 49
40 POKE 23776, 1
50 GO TO 20
```

Change line 20 back to reading PRINT AT 0, 10; "*". Now press RUN.

The screen should display *two* asterisks, in screen lines 0 and 1. It runs through the first program, printing on screen line 0; then it POKES the values needed for screen line 1; then GO TO 20 makes it PRINT again in the new line.

Try LIST; once more you'll find line 20 has become PRINT AT 1, 10; "*".

You can't POKE the ROM, of course—"Read only" means what it says! And a good job too, otherwise an accidental POKE might ruin the BASIC interpreter. In consequence, you can POKE fearlessly as the whim takes you, to see what happens.

The possibilities opened up by POKE are vast. But to handle it properly you need to know more about the detailed internal codes used by the Spectrum. Chapters 24 and 25 of the *Manual* give the basic information you need. If you're keen enough, you can work the rest out using the PEEK routine above, lines 1000–1050. I'm not going to say more, because it's better to work it out for yourself than to read through a complicated description given by someone else. But, as in the other sections of this book, I wanted to set your mind working along useful lines. By the time you've reached here, you'll have seen that PEEK and POKE are not the awful mystery that they are often made out to be. It's just that they dig deeper into the *system* by which the Spectrum functions. Experiment with them: POKEing presents a fascinating challenge and a potentially rewarding one.

*There are one or two standard problems,
and one or two useful but obscure
tricks: here's the low-down on them . . .*

24 Tips

BAFFLING SNAGS

1. It's natural to try to square a number x by working out x^2 . This is fine for positive x ; but although the square of a negative number makes perfect sense, the up-arrow won't work then. (This is because x^a is worked out as $\text{EXP}(a * \text{LN } x)$ and the logarithm of a negative number isn't defined. Use $x * x$ instead if x is likely to be negative.

2. In the same order of ideas, if x is an integer, x^2 has a nasty habit of being not quite exact, and you can run into the sort of trouble discussed in Debugging V. Similarly for x^3 : it really *can* be better to write $x * x * x$.

3. If you use a REM in a multistatement line, remember that *everything* after it (in that line) is ignored by the computer. So

```
10 REM start: LET x = 99
```

has no effect whatever. However

```
10 LET x = 99: REM start
```

sets x to 99.

4. Any IF/THEN command works like this: the *entire* line after the THEN is carried out provided the IF condition holds, but *nothing* after THEN is carried out if the condition fails. So

```
10 IF x = 0 THEN LET y = 0: IF x <> 0 THEN LET y = 1
```

sets y to 0 if x is 0, but is *ignored entirely* if $x <> 0$. In contrast, as separate lines

```
10 IF x = 0 THEN LET y = 0
```

```
20 IF x <> 0 THEN LET y = 1
```

y will be set to 1 for any non-zero x . If your conditional statements are going haywire, check the multistatement ones first!

5. In an IF/THEN statement, the machine works out the *whole* expression between IF and THEN before checking to see if it's true. This can lead to crashes, and unintended error reports. For instance,

```
10 IF x$ <> "" AND VAL x$ = 5 THEN STOP
```

causes a crash if $x\$$ happens to be empty—even though the $x\$ <> ""$ part is false and hence forces the entire condition to be false, whatever VAL $x\$$ might be. The machine *still* insists on calculating VAL $x\$$ when $x\$ = ""$, which it doesn't like. You can easily *think* you've protected your program against crashes, but you can be mistaken . . .

6. There are two ways to get a blank space. One is the SPACE key, the other is graphics 8 on CAPS SHIFT. *These are not the same character* as far as the Spectrum is concerned. So searching for spaces with a command

```
10 IF n$(i) = " " THEN GO TO 5000
```

will fail if the blank at n\$(i) is not the same kind of blank as the one in line 10 of your program. Unfortunately, you won't see the mistake on a listing!

7. You can SAVE a picture using

```
SAVE "Picture" SCREEN$
```

It's terribly easy to try to load it back using

```
LOAD "Picture"
```

in which case the message "Bytes: Picture" comes up OK, but nothing actually happens. What you should have done, of course, is

```
LOAD "Picture" SCREEN$
```

8. The quality of the colour display is very sensitive to the quality of your TV set, and to the tuning. On a very cheap portable colour TV you may find the display just won't work satisfactorily at all; so DON'T buy a small colour TV specially to use as a monitor without checking that it's OK first. If the shop won't let you plug your Spectrum in before buying, go to another shop!

9. When trying to LOAD a program, there is a very distinct high-pitched Bleep through the speaker whenever a program name is encountered. If nothing shows up on the TV when you hear this bleep, something is definitely amiss.

10. If you've got a program in the memory, and want to SAVE it on a tape, but have forgotten whereabouts on the tape you are—and are worried about leaving too big a gap or overwriting something important—type

```
VERIFY "Rubbish"
```

where "Rubbish" is a name not used for any program. Then run the tape, and watch the messages to see whereabouts you are. When you reach the right place, stop the tape; BREAK; then LOAD your program as usual.

SYSTEM VARIABLE TRICKS

1. You can produce a much more satisfying keyboard bleep if you make the direct entry

```
POKE 23609, 50
```

Numbers other than 50 give slightly different effects, but 50 is about right.

2. You can speed up the auto-repeat by direct entry of

```
POKE 23562, 2
```

(say). Change the 2 to 1 or 3 for faster or slower auto-repeat; change it to 0 to disable the auto-repeat altogether.

3. You can shorten the time the machine waits before an auto-repeat by keying in

```
POKE 23561, 20
```

(say). Reduce the 20 to reduce the delay, increase it to increase it.

4. Some programs need to SCROLL automatically. (A lot of ZX81 games *use* the scroll, and you may want to transfer these to your Spectrum.) But there is no SCROLL command.

However, you can fool the beast into scrolling by using the routine:

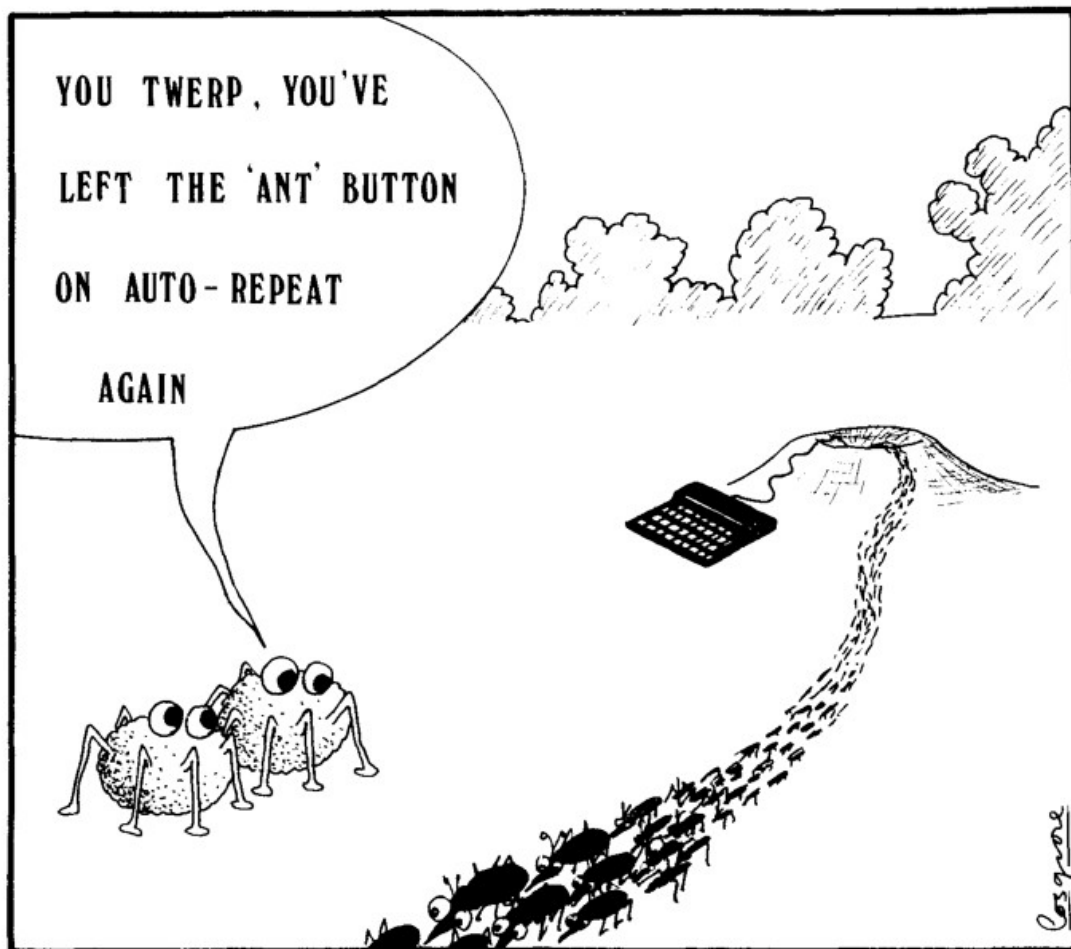
```
1000 PRINT AT 21,0
1010 POKE 23692,2
1020 PRINT
```

In fact, POKE 23692 with anything bigger than 1, at the moment the screen fills up, and then try to PRINT a new line: this produces a scroll without keyboard entry.

5. The DRAW x, y command, nice as it is, draws from the current position $x0, y0$ to the new position $x0 + x, y0 + y$. That is, x and y are the *offsets* needed, not the new coordinates. A way to draw from the current position to a new position x, y is to use the command

```
DRAW x - PEEK 23677, y - PEEK 23678
```

The advantage of this is that it works even if you've lost track of where the last PLOT position was (easily enough done) and it is not subject to cumulative round-off errors if you use it in a loop, say to draw curves. Addresses 23677 and 23678 hold the coordinates of the last point plotted, of course!



And now, the terrible truth is revealed . . .

25 What I haven't told you about

There is far, far more to your Spectrum than I have been able even to hint at in this book. I'd cheerfully have included more, but that would have made the book 17 volumes long, and each would have cost you about ten quid . . . However, by the time you've survived *this* book, the Sinclair *Manual* will make a lot more sense. (I'm not casting nasturtiums at the *Manual*—but, by definition, a manual has to tell you *everything*, and this means some pretty compact descriptions.)

For example, I've said nothing about the mathematical functions like EXP, COS, TAN, LN; I've told you nothing about user-defined functions DEF FN A (x, y, z, . . .) which are well worth knowing about; I've said next to nothing about the attributes ATTR; I've only given you one of the uses of USR (but the other leads into Machine Code, see Chapter 26, What Next?); I've used a few multidimensional arrays but never explained them; I've said nothing about INVERSE; and I've not explained about LOAD and SAVE because the *Manual* discusses them very clearly and the whole system is very nearly foolproof anyway.

But the main point I want to make here is that if, in a program listing, you see a command that you don't understand, *you can still copy out the listing and run it*. If you're feeling adventurous, you can change that puzzling command, and see what happens: that way, you may even find out what it does. Be bold.

This isn't the end: it's just the beginning. The question is:

26 What Next?

1. Read the *Manual* again.
2. In the UK there is a National ZX Usergroup. Its address is
Tim Hartnell
44-46 Earls Court
London W8 6EJ
3. If you are interested in educational programs, contact EZUG, the Educational ZX Usergroup. It lives at:
Eric Deeson
Highgate School
Birmingham B12 9DS

As we write, the potential for the Spectrum in educational use has increased dramatically, and it looks likely to become very widespread. (See Eric's book: *Spectrum in Education*; another Shiva title.)

4. There are local Usergroups, and National ones in other countries: contact the National Usergroup in (2) above for details.
5. There are swathes of magazines for home computers. Among those that carry a reasonable amount of stuff for the Spectrum are *Your Computer*, *ZX Computing*, *Sinclair User*, *Sinclair Programs* and *Computer and Video Games*. A good general computing magazine is *Personal Computer World*. All of these are widely available through newsagents.
6. Indubitably there will soon be swathes of software (tapes) and hardware (add-ons) for the Spectrum. See the magazines.
7. Read the hardware and software reviews in the magazines: most dealers are reputable, but there are a number of cowboy types, best avoided if you can spot them.
8. For more advanced BASIC programming techniques, or for Z80 Machine Code, we strongly recommend the book *Machine Code and Better Basic*, published by, er, Shiva Publishing Ltd. It's written by some blokes called Stewart and Jones, and it's excellent value for money. Although written as if the machine in question is the ZX81, almost all of it applies to the Spectrum—advanced computing depends less on the particular machine, and anyway the Spectrum can do almost anything the ZX81 can. (The display file is a bit different, though.)
9. If you haven't got it already, the extra 32K RAM will prove irresistible soon. And watch out for the microdrives . . .

Prepacked Programs

The programs that follow are intended to illustrate various features of the Spectrum, and to show you the sort of thing that can be achieved. Each is complete in itself, and need only be copied and RUN. It's not necessary to understand the commands in the listing.

However, by the time you've worked your way through this book, you ought to be able to analyse the way these programs work. This won't always be easy, though: it's often difficult to get used to somebody else's programming style. Good style requires clarity, partly because it's a soul-destroying task trying to modify programs that look like *Finnegan's Wake* written in Chinese, and partly because you tend to make fewer errors anyway.

I have deliberately left a lot of programs with untidy line numbers. It's amazing how often errors creep in when you renumber a program. And I wanted to emphasize that you do *not* have to use line numbers that are multiples of 10. The only reasons for doing this, apart from fashion or fussiness, are to leave room for changes to the program, or for inserting tracers when debugging.

In these listings, I've drawn the graphic characters as they appear on the keyboard. Boxes around a letter, such as **H**, mean inverse video; an open box□ denotes a space: obvious spaces aren't made explicit; but ones that are important and easily missed out are.

A word about *colour*. Before running any program, you can select BORDER, PAPER, and INK by direct keyboard entry—for instance

BORDER 3: PAPER 5: INK 1

I have *not* inserted such commands in the prepacked programs: you can put them in if you want. The aim is to save you time typing the program in, and to make it clearer—even if the display is not quite so spectacular. Similarly, some of the programs are fairly simple: this is because I hope you will try to see what they *do*, and not just copy them blindly (for that you should get *Games to Play on your Spectrum*, by Martin Wren-Hilton, and *Computer Puzzles: For Spectrum and ZX81*, by—you guessed it!—Stewart and Jones).

The programs are not listed in any special order. Dip in!

For my first prepacked program, I chose one whose internal workings are pretty close to the surface. It calculates:

The Area of a Triangle

If the sides of a triangle are a , b , c then its area is given by the formula $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{1}{2}(a+b+c)$. This program uses the formula to compute the area of a triangle whose sides are INPUT in turn.

```
10 INPUT a
20 PRINT "a = "; a
30 INPUT b
40 PRINT "b = "; b
50 INPUT c
60 PRINT "c = "; c
70 LET s = .5 * (a + b + c)
75 LET x = s * (s - a) * (s - b) * (s - c)
80 IF x < 0 THEN GO TO 110
90 PRINT "AREA IS "; SQR x
100 STOP
110 PRINT "IMPOSSIBLE TO FORM A TRIANGLE"
```

Program notes

1. Programs that evaluate formulae and print out the answers are so transparent that it would be cheating to count them as "real" programs! They're more like mechanized pocket calculator routines. But, at this stage, I felt that an obvious program wouldn't do any harm.
2. You can easily modify this type of program to work out any reasonable formula. Perhaps this is one way to make mathematical formulae more interesting, and get some educational value out of the Spectrum. Here are a few suggestions for possible programs:

- (a) The surface area of a sphere of radius r is $4\pi r^2$.
[π is PI on the screen: key M in extended mode.]
- (b) The volume of a sphere of radius r is $\frac{4}{3}\pi r^3$.
- (c) The volume of a cylinder of radius r and height h is $\pi r^2 h$.
- (d) The volume of a cone of radius r and height h is $\frac{1}{3}\pi r^2 h$.
- (e) The solutions of the quadratic equation $ax^2 + bx + c = 0$ are given by

$$x = (-b \pm \sqrt{b^2 - 4ac}) / 2a$$

(Work out the + and the - roots separately. If $b^2 - 4ac < 0$, the roots are imaginary, and you'll have to deal with this, either by PRINTing "IMAGINARY ROOTS" or concocting a piece of program to work them out if you know about complex numbers. You'll also have to deal with the possibility that $a = 0$.)

- (f) The sum $1 + 4 + 9 + \dots + n^2$ is equal to $\frac{1}{6}n(n+1)(2n+1)$. INPUT n and PRINT the sum. [Also, for comparison, work it out by summing the series, using FOR/NEXT, once you've read LOOPING, page 18.]

- (g) The period of swing of a pendulum of length l is $T = 2\pi\sqrt{l/g}$ where g is the acceleration due to gravity: 981 cm/sec/sec or 32 ft/sec/sec.
- (h) In fact g varies from place to place: a more accurate approximation is that it depends on latitude L and height h above sea level, according to the formula
- $$g = 980.616 - 2.5928 \cos(2L) + 0.0069 \cos^2(2L) - 0.0003h \quad \text{cm/sec/sec}$$

Write a program to work out g , and T , given L , h , and l .

If triangles aren't your thing, why not write a program to work out your bank balance?

Am I Overdrawn?

```

10 PRINT "PREVIOUS BALANCE IS ";
20 INPUT b
30 PRINT b
40 PRINT "LIST DEBITS"
50 INPUT d
60 IF d < 0 THEN GO TO 90
70 LET b = b - d
80 GO TO 50
90 PRINT "LIST CREDITS"
100 INPUT d
110 IF d < 0 THEN GO TO 140
120 LET b = b + d
130 GO TO 100
140 PRINT "CURRENT BALANCE IS "; b

```

Program notes

- Lines 60 and 110 are *delimiters* (see page 25). If you input a negative number the machine knows you've finished the listing. (This negative number is *not* included in your bank balance!)
- Debits are mostly chequebook stub figures. But don't forget standing orders. (An obvious first improvement is to build these into the program. See if you can manage it.) And withdrawals from cash-dispensers.
- Credits: don't forget your salary cheque! Maybe you can build this in too.
- Your younger children can play banks for hours using this.
- If you add two more lines:

```

55 PRINT d
105 PRINT d

```

the machine will list the items. If you have more than 20 items the screen fills, and it is necessary to scroll to continue.

*Deep in the depths of the dark Sinclair
Forest lurks a Terrible Tiger. Can you
find him?*

Tiger-hunt

```
PAPER 7: INK 8: BORDER 1
10 INPUT "Choose size of forest: max 16 □"; w
15 IF w > 16 THEN LET w = 16
20 FOR i = 0 TO 8 * w STEP 8
30 PLOT i + 64, 32
40 DRAW 0, 8 * w
50 PLOT 64, i + 32
60 DRAW 8 * w, 0
70 NEXT i
80 LET z$ = "01234567891111111"
90 LET y$ = "□□□□□□□□□0123456"
100 PRINT AT 19, 8; z$ (TO w) + "□" + "x"
110 PRINT AT 20, 8; y$ (TO w)
120 PRINT AT 16 - w, 6; "y"; AT 17 - w, 0
130 FOR i = 1 TO w
140 PRINT TAB 6 - (w - i > 9); w - i
150 NEXT i
200 LET mx = INT (w * RND)
210 LET my = INT (w * RND)
220 INPUT "Where is the Tiger? □"; x; "□"; y
225 IF x > w OR y > w THEN GO TO 220
230 LET d = ( (mx - x) * (mx - x) + (my - y) * (my - y) ) / w / w
240 LET d = d * 8: IF d > 4 THEN LET d = 4
250 PRINT PAPER d + 2; AT 17 - y, 8 + x; OVER 1; "□"
255 IF mx = x AND my = y THEN GO TO 300
260 GO TO 220
300 PRINT AT 17 - my, 8 + mx; OVER 1; "*"
310 PRINT AT 0, 0; FLASH 1; "Gotcha!"
320 FOR i = 1 TO 30
330 BEEP .1/i, i
340 NEXT i
```

Program notes

1. You can choose any size of wood from 1 to 16. If you input more, it gives you size 16. The wood is drawn with x- and y- coordinates along the bottom and down the side (using lines 20–70 for the wood, 80–150 for the coordinates). Notice how setting up the picture is the major part of the program!
2. When asked “Where is the tiger?” you must input two numbers between 0 and w – 1; these are the coordinates of your guess. If you input numbers outside this range they are ignored.
3. The machine then prints a coloured square at this position. The colour gives you a clue as to how close you are: red means you’re very close, magenta fairly close, green fairly far off, and yellow a long way off.
4. You keep going until you locate the tiger. You then get a “*”, a flashing message of congratulation, and a musical offering to mark the occasion. (The music is in lines 320–340: you can use it in your own programs to produce the same sounds.) PARENTS may delete these lines to preserve their sanity.
5. To try again, hit RUN.

Projects

1. When you are really close to the tiger, arrange for the red square to flash. You can do this in line 250. Insert after OVER 1; a piece of code like this:

FLASH (d < something or other)

and experiment to find what value something or other should be for a nice effect.

2. Change the message to something rude.
3. Change the music at the end: have a selection of tunes from which the machine chooses at random, to avoid boredom.

*Be your own Beethoven,
or Spectrum its own Sibelius . . .*

Composer

This program will allow the Spectrum to play music. The notes are entered one at a time, as single letters or single letters followed by a hash mark (#) to indicate sharps. Enter ** to terminate the sequence of notes and start the Spectrum playing the tune.

```
10 DIM s (7)
11 DIM n$ (2)
12 DIM t (1500)
20 LET s (1) = -3: LET s (2) = -1: LET s (3) = 0: LET s (4) = 2
30 LET s (5) = 4: LET s (6) = 5: LET s (7) = 7
35 FOR q = 1 TO 1500
40 INPUT "Enter a note": n$
42 IF n$ = "**" THEN GO TO 200
45 LET i = 0
50 LET p = CODE n$ - 64
```

```

55 IF p > 10 THEN LET p = p - 32
60 IF n$(2) = '#' THEN LET i = 1
100 LET t(q) = s(p) + i
110 NEXT q
200 FOR r = 1 TO q
210 BEEP 0.5, t(r)
220 NEXT r

```

The details of how this works are explained in the section on arrays.

Improvements and modifications

1. Replace line 40 with:

```
40 LET n$ = CHR$(INT(RND * 7) + 65)
```

The program now composes its own musical piece, with exactly 1500 notes in it. There are no sharps in the result, so it sounds pretty bland.

2. The length of each note is fixed at 0.5 seconds. Can you devise a routine which will allow the user to enter the desired length of each note, at the same time as the note itself? You'll need another array to hold the duration values, the same length as t. (In 16K, two arrays of length 1500 will run you out of memory. Make each array 750 long.)
3. Use a similar technique to that in (1) to generate random music with random note lengths.
4. Entering the music a note at a time can get a bit tedious. Can you devise a way in which it could be entered as a single string?
5. This could be a useful composing tool if individual notes could be altered easily. Can you invent a melody editor (meloditor?) which allows the entry of, say, 38, A# to mean: "alter the 38th note to A#"?
6. Break out of the single octave straitjacket. Again, you need more entries per note. For instance A, 0 could mean "A in the octave around middle C" and C# 2 could mean "C#, two octaves above middle C".

*Here's a program to keep the kids quiet
on a rainy afternoon . . .*

Sink the Bismarck

A routine patrol in the North Sea . . . suddenly from out of the fog, the enemy vessel appears, about a mile away. You tell your gunners to set the elevation and muzzle velocity of their gun. Will you be able to sink the *Bismarck*?

```

10 PRINT TAB 8; "SINK THE BISMARCK"
20 LET t = 15 * (1 + RND)
30 PRINT AT 21, 0; INK 5; INVERSE 1; "□□□□□□□□□□
   □□□□□□□□□□□□□□□□□□□□□□□□□□□□" [32 spaces]
40 PRINT AT 20, t - 2; INK 4; "■ ■ ■ ■"
50 INPUT "Elevation = □": e

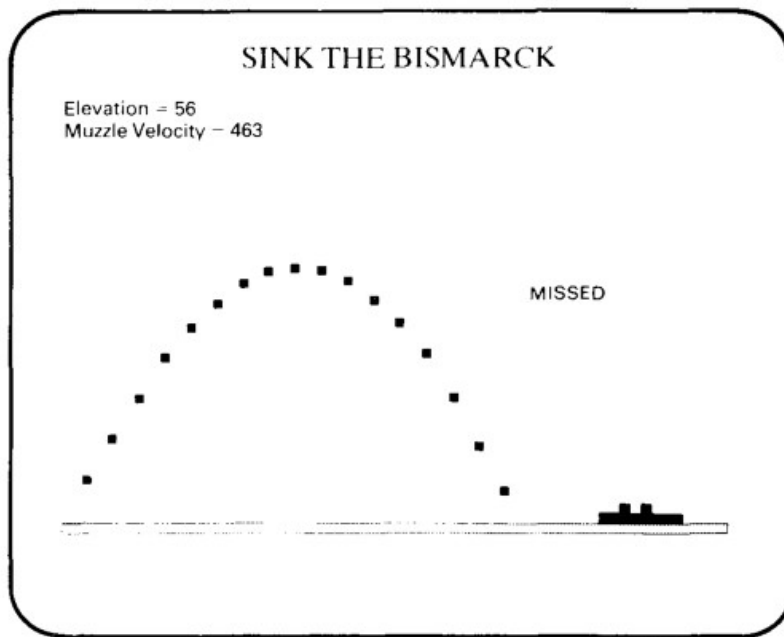
```



```

60 PRINT AT 3, 0; "Elevation = "; e
70 INPUT "Muzzle Velocity = "; v
80 PRINT "Muzzle Velocity = "; v
90 LET a = v * COS (PI * e / 180)
100 LET b = v * SIN (PI * e / 180)
110 FOR j = 0 TO b/16 STEP .3
115 LET c = .01 * (b * j - 16 * j * j)
120 IF a * j > 6200 THEN GO TO 190
130 IF c > 40 THEN GO TO 170
140 INK 2
150 PLOT .04 * a * j, 4 * c + 8
160 BEEP .005, c + 10
170 NEXT j
180 IF ABS (a * b / 3200 - t) < 3 THEN GO TO 210
190 PRINT AT 10, 20; "MISSED!"
200 STOP
210 FOR j = 0 TO 15
220 PRINT AT 20 - j, t - 2; "glug!"; BEEP .3, 6 - 3 * j
230 NEXT j

```



Program notes

1. It begins by drawing the sea, and a randomly placed silhouette of a ship. Line 20 chooses the random position, 30 PRINTs the sea, and 40 the ship.
2. 50–80 INPUT and display the elevation e and muzzle velocity v . The player when asked by the machine, must INPUT these from the keyboard. e is in degrees, and must be between 0° and 90° ; v is in feet per second and must be positive.
3. 90–170 compute and plot the trajectory of the shell at 0.3 second intervals—see mathematical notes, opposite.

4. 180 gives you a HIT if your shell hits the waterline within 600 feet of the ship's centre. Adjust the 3 to 2 or 1.5 for a more difficult game, or to 4 or 5 for an easier one.
5. 190-230 are output routines for a HIT or MISS.
6. To start or restart, hit RUN followed by ENTER, as usual.
7. The screen here is 6400 feet wide, as simulated.
8. The program is moderately "user-friendly" and goes on running even if the shell goes off the top of the screen (thanks to line 120).
9. The graphics in line 40 are graphics 3 and 1 in CAPS SHIFT.
10. "PI" in lines 90 and 100 is key M in extended mode, *not* keys P and I!

Mathematical notes

The shell's path is calculated using a mathematical formula accurate (in the absence of air resistance) for a body moving under gravity—assumed to be 32 ft/sec/sec. In line 110 j represents time; a is the horizontal component of velocity and b the vertical component; the $PI * e / 180$ converts from degrees to radians. Line 150 computes the height at time j and converts to screen coordinates (each PLOT pixel is 25 feet square.) In line 110, $b/16$ is the time taken to hit the waterline. In line 180, $a * b / 1600$ is the distance it has then gone; the 3200 happens because we are using a PRINT statement in line 40, so the distance to the ship is $200 * t$.

If any other feature of the program puzzles you, try changing it and see what happens.

Notice that, while *playing* a computer game like this has only minor educational value (pressing the right key, judging distances and angles), *writing* or *understanding* it requires some knowledge of both programming and mathematics, and could be used to advantage as motivation either in the classroom or at home.

*Here's a variation on the PLOT routine,
producing some beautiful curves:*

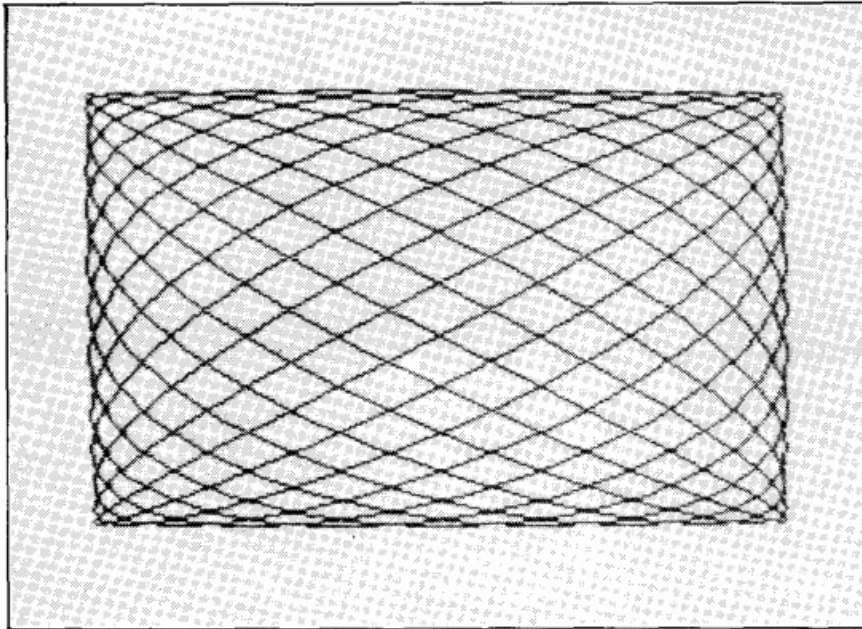
Lissajous Figures

These were invented by Nathaniel Bowditch in 1815, but named after Jules-Antoine Lissajous who reinvented them in 1857. They illustrate another way to PLOT interesting curves, using what mathematicians call a *parametrization*. What that means is that you make x and y depend on a variable t , and PLOT x, y for sequences of t 's.

```

10 INPUT "First number"; p
20 INPUT "Second number"; q
30 INPUT "Phase shift"; r
40 LET t = 0
50 LET m = p
60 IF q < p THEN LET m = q
70 LET x = 127 + 120 * COS (t * PI / 180 * p + r)
80 LET y = 87 + 80 * SIN (t * PI / 180 * q)
90 IF t = 0 THEN PLOT x, y
100 IF t > 0 THEN DRAW x-PEEK 23677, y-PEEK 23678
110 LET t = t + 10 / m
120 GO TO 70

```



Program notes

1. For pretty results, p and q should be whole numbers.
2. r can be anything, but somewhere between 0 and 3 is best.
3. A good start is p = 5, q = 7, r = 1.
4. PI is key "M" in extended mode.
5. To stop, BREAK + CAPS SHIFT.
6. For more spectacular results (but longer plot times) try p = 31, q = 29, r = 0.
7. Line 100 works out the offset from the old plot point to the new one, so that DRAW can put in a line between the two. The system variables in addresses 23677 and 23678, which are PEEKed, hold the last PLOT coordinates. This trick is extremely useful.

The Spectrum can do statistics, thanks to its random number generator RND. This program simulates

Monopoly Dice

If you've played *Monopoly** you'll remember that to find out how far your piece moves you throw two dice, and add the result. You may also have noticed that the total 7 is more common than anything else. In fact, out of 36 throws of the dice you should expect the total

- 2 to happen on average 1 time
- 3 to happen on average 2 times
- 4 to happen on average 3 times
- 5 to happen on average 4 times
- 6 to happen on average 5 times
- 7 to happen on average 6 times
- 8 to happen on average 5 times
- 9 to happen on average 4 times

* 'Monopoly' is a registered Trade Mark of John Waddington Ltd.

10 to happen on average 3 times
11 to happen on average 2 times
12 to happen on average 1 time

Of course, you won't usually get those numbers exactly, but in the long run you should get close . . . on average!

You can use the Spectrum to explore the world of statistics, by *simulating* this kind of thing using random numbers. This program "throws" two dice 144 times (4 times 36, to make life easy), counts how many times the total is 2, 3, 4, . . . etc.; plots out the results in a "bar chart"; and also compares the actual result with the expected theoretical numbers.

```
10 DIM a (11)
20 FOR j = 1 TO 144
30 LET d = 1 + INT (6 * RND) + INT (6 * RND)
40 LET a (d) = a (d) + 1
50 NEXT j
80 FOR j = 2 TO 12
90 LET q = a (j - 1)
100 LET q0 = INT (q/2)
110 LET q1 = q - 2 * q0
120 FOR t = 1 TO q0
130 PRINT AT 18 - t, 4 + 2 * j; "█"
140 NEXT t
150 IF q1 = 1 THEN PRINT AT 17 - q0, 4 + 2 * j; "[█]"
160 NEXT j
170 PRINT AT 19, 8; "2 █ 3 █ 4 █ 5 █ 6 █ 7 █ 8 █ 9 █ 10 █ 11 █ 12 █"
180 PRINT AT 20, 24; "0 █ 1 █ 2 █"
```

Program notes

1. The bar chart shows the relative number of occurrences of the totals 2, 3, 4, . . . , 12. The theoretical shape is triangular with the peak at 7. How close does it actually get? Do you get the same shape if you RUN it again? Why not?

*Here's an example of an "educational" program,
albeit not an ambitious one.*

Arithmetic Test

This program sets the operator (your seven-year-old) an addition sum involving two random two-digit numbers; checks if his answer is wrong or right; and, if wrong, explains how to get it right.

```

10 LET v = 10
20 LET c = 0
30 PRINT "Hi! I'm Sinclair Spectrum. Who are you?"
40 INPUT e$
50 PRINT "OK, "; e$; " can you answer?"
60 PRINT "this?"
70 LET x = INT (v * v * RND)
80 LET y = INT (v * v * RND)
90 IF x + y <= 12 THEN GO TO 7 * v
100 LET x1 = INT (x/v)
110 LET x2 = x - v * x1
120 LET y1 = INT (y/v)
130 LET y2 = y - v * y1
140 IF x2 + y2 >= v THEN LET c = 1
150 PRINT x; "+"; y; "= ? ";
160 INPUT a
170 PRINT a; "?";
180 LET b = x + y - a
190 PRINT ("right" AND b = 0) + ("sorry, wrong"
    AND b <> 0)
200 IF b = 0 THEN GO TO 190
210 PRINT x2; "+"; y2; " is "; x2 + y2 - c * v;
220 IF c = 1 THEN PRINT " carry 1"
230 PRINT " and then "; x1; "+"; y1; "+ the carry"
    AND c = 1; " is "; x1 + y1 + c
240 PRINT "giving "; x + y

```

Program notes

1. Line 30 is a confidence-builder. It asks the operator for his name. Line 50 uses the name to ask a question.
2. Line 160 is where the operator tells the computer what he thinks the answer is.
3. Lines 200 onwards explain to the operator how the sum should have been done, if he got it wrong.
4. The main difficulty in "educational" programs of this kind is that they tend to require lengthy displays of text. This program could be much improved—for example, setting various levels of difficulty, giving more advice if something goes wrong with the operator's calculation, protecting the program better against the operator pressing wrong keys by mistake, and so forth. And, of course, more interesting sums than plain addition would be an improvement. But most of the basic principles show up in this little program.
You can also buy prepacked "educational" programs, advertised in the magazines. The quality of these varies hugely, and some are not as good as mine!
5. A good program wouldn't need to be RUN each time, but would ask the operator if he wanted another go, and if he said yes, RUN automatically. It's easy to add a few lines to achieve this.

A sample of the Spectrum's hi-res graphic capability:

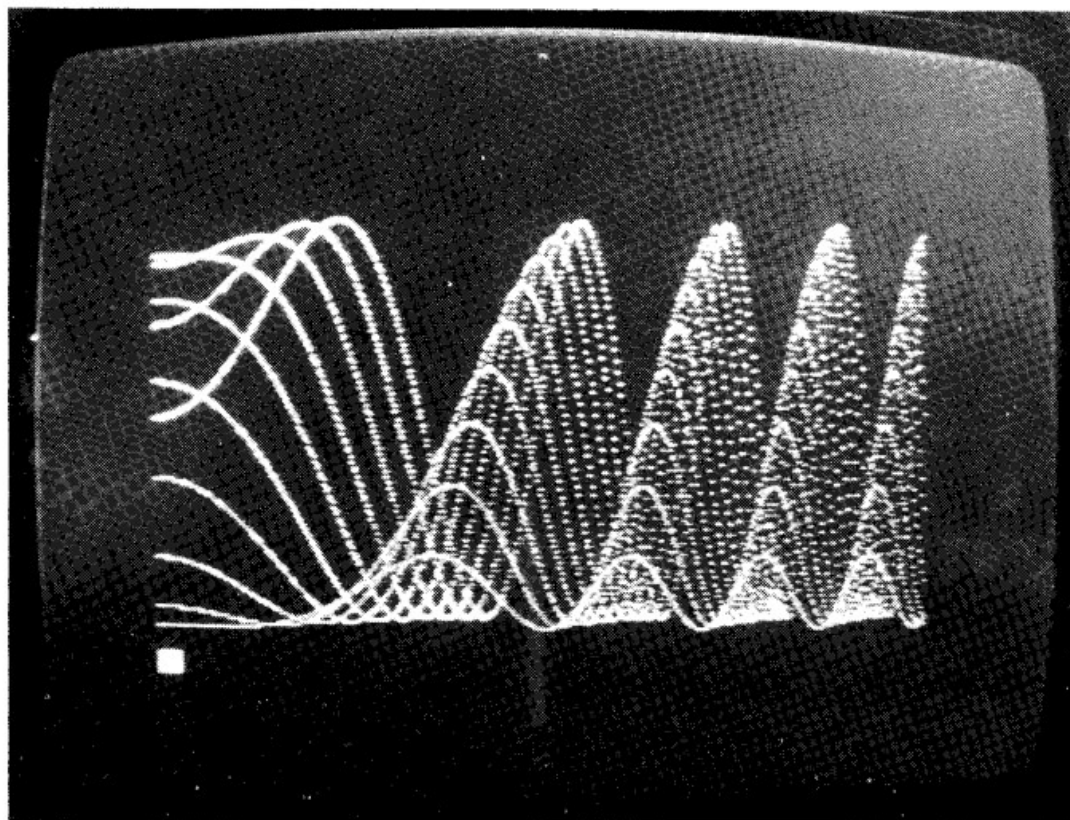
Graphics Demonstration I

For such a short program, this produces a remarkably pretty display.

```
BORDER 0: PAPER 0
10 FOR j= 0 TO 9
20 INK 7 - INT (j/2)
30 FOR i = 0 TO 510
40 PLOT .5 * i, (80 + 70 * SIN (i * i/100000) + j/2) * (1 - j * j/100)
50 NEXT i
60 NEXT j
```

Program notes

1. Although changes in INK colour affect entire 8×8 blocks of pixels, the effect here (with the lighter shades of colour) is quite nice, and you don't really notice the blocks.
2. This illustrates a good way to get interesting hi-res pictures: superimpose several (similar) curves. By "similar" I mean that just a few regularly varying changes to the formula are made at each step: notice here how the values of j change the curve.



An example of text-display:

Partridge Printout

This yuletide program displays, at singable speed, the words to a well-known carol.

```
10  FOR d = 1 TO 12
20  PAUSE 45 * d + 250
30  CLS
40  PRINT "On the "; d;
50  IF d = 1 THEN PRINT "st";
60  IF d = 2 THEN PRINT "nd";
65  IF d = 3 THEN PRINT "rd";
70  IF d > 3 THEN PRINT "th";
80  PRINT " day of Christmas my true love gave to me"
90  GO SUB 132 - d
100 NEXT d
110 STOP
120 PRINT "12 drummers drumming"
121 PRINT "11 pipers piping"
122 PRINT "10 lords a-leaping"
123 PRINT "9 ladies dancing"
124 PRINT "8 maids a-milking"
125 PRINT "7 swans a-swimming"
126 PRINT "6 geese a-laying"
127 PRINT "5 gold rings"
128 PRINT "4 calling birds"
129 PRINT "3 french hens"
130 PRINT "2 turtle doves and"
131 PRINT "a partridge in a pear tree"
140 RETURN
```

Program notes

1. The screen blanks out for a few seconds before anything happens.
2. Line 20 adjusts the tempo. The 45 adjusts the time for lines 120–131; the 250 the time for 40–80. Boxes denote important (non-obvious) spaces.

*The Spectrum is quite an effective number-cruncher.
To convince you of this, take an 8-digit number
and see how quickly this program will give its*

Prime Factors

```
10 INPUT k
30 PRINT k; "□ = □";
40 LET k0 = k
50 IF INT (k/2) * 2 = k THEN GO TO 140
60 LET n = 3
70 IF INT (k/n) * n = k THEN GO TO 110
80 IF n * n > k THEN GO TO 170
90 LET n = n + 2
100 GO TO 70
110 PRINT n; ".";
120 LET k = k/n
130 GO TO 70
140 PRINT "2.";
150 LET k = k/2
160 GO TO 50
170 IF k = k0 THEN PRINT FLASH 1; "PRIME"
180 IF k < k0 AND k > 1 THEN PRINT k
```

Program notes

1. $\text{INT}(k/n) * n = k$ if and only if k is divisible by n . So this tests for divisors.
2. The program tries dividing k by 2, and all odd numbers less than the square root of k . If no such divide k , then it is prime.
3. If a divisor is found, the program divides k by it, and hunts for a new divisor of the same size. If it doesn't find one it looks for a bigger divisor.
4. The machine takes about 35 seconds with an 8-digit prime and less otherwise. (The times quoted are for 11111117; obviously, they will change for different numbers.)
5. Modifications: Embed this in a loop, and produce a display of prime factors of k , $k + 1$, $k + 2$, . . . and so on indefinitely.
6. k has to be at most 8 digits, because the Spectrum won't handle larger numbers with enough precision.
7. This program could be made mathematically more efficient, for instance by excluding multiples of 3 or 5 from trial divisors. Can you use this to make a program run *faster*? The price for mathematical efficiency is increased program size: does the gain wipe out the loss if you're careful?

*Everybody has programs to solve quadratic equations.
What about this?*

Solving Cubics

This program finds all real roots of cubic equations $ax^3 + bx^2 + cx + d = 0$, to reasonable accuracy.

```
10 INPUT a, b, c, d
50 LET b = b/a
60 LET c = c/a
70 LET d = d/a
80 LET x = 0
90 LET g = 2 * x * x * x + b * x * x - d
100 LET h = 3 * x * x + 2 * b * x + c
110 IF h = 0 THEN GO TO 200
120 IF ABS (x - (g/h)) < 1.E - 8 THEN GO TO 300
130 LET x = g/h
140 GO TO 90
200 LET x = x + 1
210 GO TO 90
300 PRINT "x1 = "; x
400 LET a = b + x
410 LET b = x * x + b * x + c
420 LET d = a * a - 4 * b
430 IF d < 0 THEN PRINT "Others imaginary"
440 IF ABS d < 1.E - 7 THEN PRINT "Numerical instability possible"
445 IF d < 0 THEN STOP
450 PRINT "x2 = "; (-a + SQR d) / 2
460 PRINT "x3 = "; (-a - SQR d) / 2
```

Program notes

1. The program finds one root of the equation by an iterative process known as the *Newton-Raphson method*, which involves taking a trial value for x and improving it successively until it gets near enough to a root. Lines 90–150 perform this process.
2. If you want to see the way this iteration works, add

```
131 PRINT x
```

and watch the numbers converge on the answer.

3. Having found one root, the program divides out by this to get a quadratic and solves it by the formula in lines 400–460.
 4. If this quadratic has no real roots, line 430 applies; the program then crashes in line 450 but no harm is done.
 5. A fine point: for some cubics inaccuracies in the arithmetic build up too much, and the program may claim “Others imaginary” when this is not in fact the case. Line 440 warns the operator that this is on the cards. The problem is that the sign of d is crucial, and if d is close to 0, errors can have a disastrous effect.
Most numerical methods encounter similar problems, and a large part of the subject called numerical analysis spends a lot of time tackling these.
 6. For example, try INPUTting $a = 4$, $b = -8$, $c = 5$, $d = -51$. You should get $x_1 = 3$, other imaginary. Now try $a = 4$, $b = -16$, $c = -5$, $d = 51$: this time you get $x_1 = 3$, $x_2 = 2.6213203$, $x_3 = -1.6213203$.
-

You, too, can have an electronic gambling device:

Fruit Machine

```

10 LET c = 0
20 DIM a(3)
30 FOR t = 1 TO 3
40 LET r = INT(3 * RND)
50 LET a(t) = r
60 LET i = 5
70 LET j = 10 * t - 5
80 GO SUB 300
90 NEXT t
100 LET c = c - 1
110 IF a(1) = a(2) AND a(2) = a(3) THEN GO TO 700
120 GO SUB 1000
130 INPUT "Stop?"; b$
140 IF b$ = "s" THEN STOP
150 GO TO 30
300 PRINT INK 2 * r; AT i, j; "■ ■"; AT i + 1, j; "■ ■"
310 BEEP .1, 3 * r
320 RETURN
700 LET c = c + 9
720 GO TO 120
1000 IF c >= 0 THEN PRINT AT 18, 5; "You have won □"; c; "□ pounds □"
1010 IF c < 0 THEN PRINT AT 18, 5; "You have lost □"; -c; "□ pounds"
1020 RETURN

```

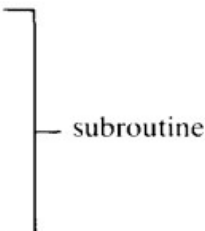
Program notes

1. This displays three decorated squares. If they are all the same, you win £9. If not, you lose £1. To play, push any key except "s"—we recommend ENTER as it saves a keystroke. Push "s" to stop.
 2. On average, you should break even. This is better than you'll get in the local pub.
 3. Note the use of subroutines to plot the three possible types of square.
 4. Note the proper use of the screen: you get a reasonable display in the middle, not a tiny little one bunched up in a corner.
-

The Spectrum will let you call a subroutine from inside itself. This is known as recursive programming. See if you can work out how this program does its job.

Obscure Factorials

```
10 LET f = 1
20 INPUT n
30 LET m = n
40 GO SUB 100
50 PRINT m; "□ Factorial is □"; f
70 STOP
100 IF n <= 1 THEN RETURN
110 LET f = f * n
120 LET n = n - 1
130 GO SUB 100
140 RETURN
```



What day of the week were you born on? What day did Anne Boleyn die? Now you can find out with the:

Dayfinder

This program accepts as input a date d. m. y. where d is the day number, m the month, and y the year (e.g. 23.7.1066) and works out which day it is (was, will be).

```
10 LET a$ = "033614625035"
20 LET b$ = "SUNMONTUEWEDTHUFRISAT"
```

```

30 INPUT d
40 PRINT d; “.”;
50 INPUT m
60 PRINT m; “.”;
70 INPUT y
80 PRINT y; “□ is □”;
90 LET z = y - 1
100 LET c = INT (z/4) - INT (z/100) + INT (z/400)
110 LET x = y + d + c + VAL a$(m) - 1
120 IF m > 2 AND (y = 4 * INT (y/4) AND y < > 100 * INT (y/100) OR
    y = 400 * INT (y/400) ) THEN LET x = x + 1
130 LET x = x - 7 * INT (x/7)
140 PRINT b$(3 * x + 1 TO 3 * x + 3)

```

Program notes

1. Line 10 stores a list of twelve “monthly correction numbers” in compact form as a string. Line 100 works out the month in the list as part of a computation explained in note 4.
2. Line 20 uses a similar trick to simplify the printing routine. See how line 140 selects the right group of three letters.
3. Line 100 computes how many leap years have elapsed since the year dot. Remember: multiples of 4 are leap years, but multiples of 100 are not unless also multiples of 400.
4. Line 110 is the guts of the computation. The idea is to count the number of days elapsed since some (essentially arbitrary) reference date, but to save space by throwing away multiples of 7 since these won’t affect the day of the week. So the number of years, *y*, gets multiplied by 365; but $365 = 7 * 52 + 1$ so instead of $365 * y$ we use *y*. The curiosities of month-lengths are taken care of by *a\$(m)*. Note the use of VAL to convert a single-digit character to an actual *number*. To calibrate the program I chose a date for which we knew the day of the week—30.9.1981 was a Wednesday—and that gave me the -1 correction on the end. Line 120 adjusts the computation in January or February of a Leap Year, where it would otherwise go wrong.
5. One calendric subtlety is missing. In 1752 Britain changed from the “Old Style” calendar to the “New Style”. The day after 2.9.1752 was 14.9.1752. My program assumes New Style.

Projects

Modify it to work with Old Style dates for pre-1752 periods.

6. The program accepts impossible dates like -37.12.-992. Modify it to accept only sensible dates.
7. It won’t work for dates BC. Modify it so it will. NB: There was no year 0 between 1 BC and 1 AD, even if logically there should have been!

Strings and numbers are sometimes closer than you might imagine—for example, in:

Binary/Decimal Conversion

Binary numbers only use 0's and 1's; instead of 0, 1, 2, 3, 4, 5, 6, . . . they go 0, 1, 10, 11, 100, 101, 110, . . . where in general something like 1101001 is worked out from right to left as

$$1 \times 1 + 0 \times 2 + 0 \times 4 + 1 \times 8 + 0 \times 16 + 1 \times 32 + 1 \times 64 = 105$$

each digit counting double what the previous one does. Computers, of course, use binary numbers for internal work (though modern computers use sophisticated variants, in practice).

The following two programs convert an INPUT number from binary to decimal, or conversely.

Binary to decimal

```
10 INPUT a$
20 PRINT a$;
30 LET l = LEN a$
40 LET s = VAL a$(1)
50 FOR j = 2 TO l
60 LET s = 2 * s + VAL a$(j)
70 NEXT j
80 PRINT "□ is □"; s; "□ in decimal"
```

Decimal to binary

```
10 LET c$ = ""
20 INPUT a
30 PRINT a;
40 LET d = INT (a/2)
50 LET r = a - 2 * d
60 IF r = 0 THEN LET b$ = "0"
70 IF r = 1 THEN LET b$ = "1"
80 LET c$ = b$ + c$
90 IF d = 0 THEN GO TO 120
100 LET a = d
110 GO TO 40
120 PRINT "□ is □"; c$ "□ in binary"
```

Program notes

1. The INPUT for binary/decimal conversion must be a sequence of 0's and 1's, like 11000101011. For decimal/binary it must be a whole number, like 3427006.
2. The two programs have been written to illustrate different approaches to the conversion problem. It would be possible to make both look very similar in structure.
3. In the decimal/binary conversion, note that lines 60–70 *cannot* be replaced by

```
60 LET b$ = "r"
```

You can avoid having two lines by using CHR\$ and being clever; or you can write

```
60 LET b$ = ("1" AND r) + ("0" AND 1 - r)
```

which works for reasons to do with how the Spectrum handles logic—see the *Manual*, p. 86.

Or, yet again:

```
60 LET b$ = "0"
```

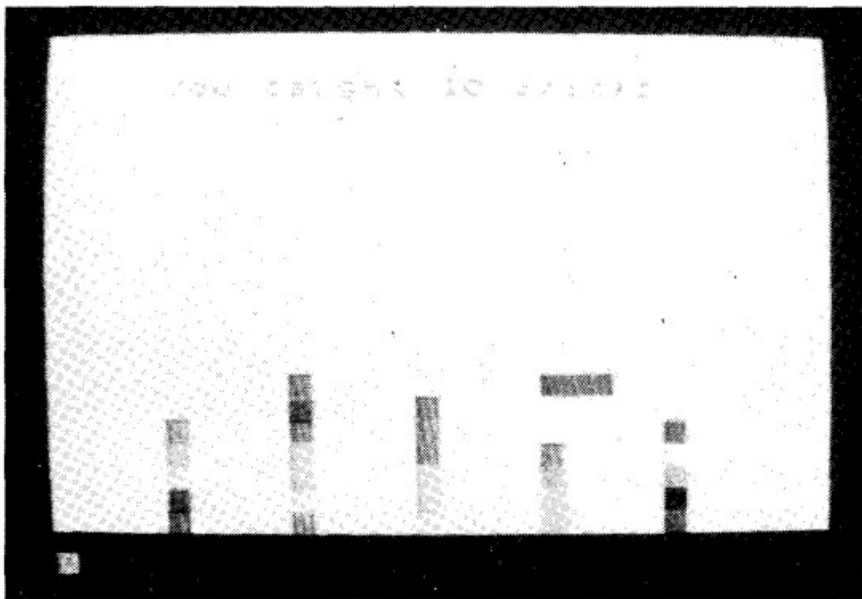
```
70 IF r = 1 THEN LET b$ = "1"
```

will do the trick and save a little space.

A graphic and moving experience:

Brickbat

Coloured bricks drop from the sky. If you catch them with your bat, they vanish. If you miss, they pile up. When the piles get too high, the game ends. How many can you catch?



BORDER 1: PAPER 7

```
10 LET h = 11
20 LET c = 0
30 DIM a (5)
40 LET p = INT (5 * RND + 1)
50 LET j = 5 * p
60 FOR i = 1 TO 21 - a (p)
70 PRINT AT i, j; INK 6 * RND; INVERSE 1; "□"
80 BEEP .02, 35 - 1.9 * i - j/2
90 PRINT AT i - 1, j; "□"
100 LET m$ = INKEY$
110 IF m$ = "5" THEN LET h = h - 1
120 IF m$ = "8" THEN LET h = h + 1
130 IF h < 1 THEN LET h = 1
140 IF h > 26 THEN LET h = 26
150 PRINT AT 15, h; "□"; INK 2; INVERSE 1; "□ □ □"; INVERSE 0; "□";
160 IF i = 15 AND ABS (h + 2 - j) <= 1 THEN LET c = c + 1: GO TO 40
170 NEXT i
180 LET a (p) = a (p) + 1
190 IF a (p) = 7 THEN PRINT AT 2, 5; INK 1;
    "You caught □"; c; "□ bricks": STOP
200 GO TO 40
```

Program note

1. To move the red bat left or right use the arrow keys 5 and 8.

*More graphics, and a useful trick
for using DRAW to plot curves:*

Spirals and Rosettes

BORDER 1: PAPER 2: INK 7

```
10 FOR t = 0 TO 4 STEP .5
15 PLOT 128, 88
20 FOR y = 0 TO 720
30 LET x = y * PI / 180
40 LET r = 1.5 * x
```



```

50 LET a = 128 + 5 * r * COS (x + t)
60 LET b = 88 + 4 * r * SIN (x + t)
70 DRAW a-PEEK 23677, b-PEEK 23678
80 NEXT y

```

Program notes

1. "PI" in line 30 is key M in extended mode, not the letters P and I.
2. Line 70 makes use of the system variable COORDS (see the *Manual*, p. 175) to calculate the correct offset for drawing from the old PLOT position to the new one. In general, if you replace the command PLOT p, q by DRAW p-PEEK 23677, q-PEEK 23678, the machine will draw a straight line to the next PLOT position. There are other ways to achieve this effect, but using COORDS avoids errors building up, which is not the case with some methods. You can experiment with this idea, even if you don't understand PEEK.
3. To draw ROSETTES instead of SPIRALS, change lines 20 and 40 to read

```

20 FOR y = 0 TO 360
40 LET r = 20 + SIN (7 * x)

```

4. Experiment, changing the "7" to other values, such as 3, 4, 5, 6, 8, 9, 10.

*This program may not help you to draw
like Picasso, but as long as you stick to colour 1,
you too can have a blue period!*

Picasso

The program starts by asking for a "mode". This is either "d" for draw, "e" for erase, or "f" for finish. In the latter case, the program is terminated.

If you enter "d", the program prompts for a colour. You enter this as the appropriate digit (1 for blue, 2 for red etc.)

For either "d" or "e", the program now asks for an option, which must be a number in the range 1 to 9. The effect of each option is shown below. Note that where the word "draw" is used "erase" should be substituted if the mode is "e".

Option Action

1. Draw horizontal and vertical scales for reference. A dot appears in every 5th column and row, a double dot (short line) in every 10th, and a slightly longer line in every 100th.
2. Rectangle. You will be asked for the left and right columns and bottom and top rows in which the edges are to appear. You will then be asked if the rectangle is to be drawn, or used as a frame (see option 7).
3. Circle. You will be asked for the column and row in which the centre appears, and the radius.
4. Segment. You will be asked for the coordinates of the two corners of the segment of a circle, and the largest distance from the curve to the straight line.
5. Straight line. You will be asked for the coordinates of the two ends of the line.

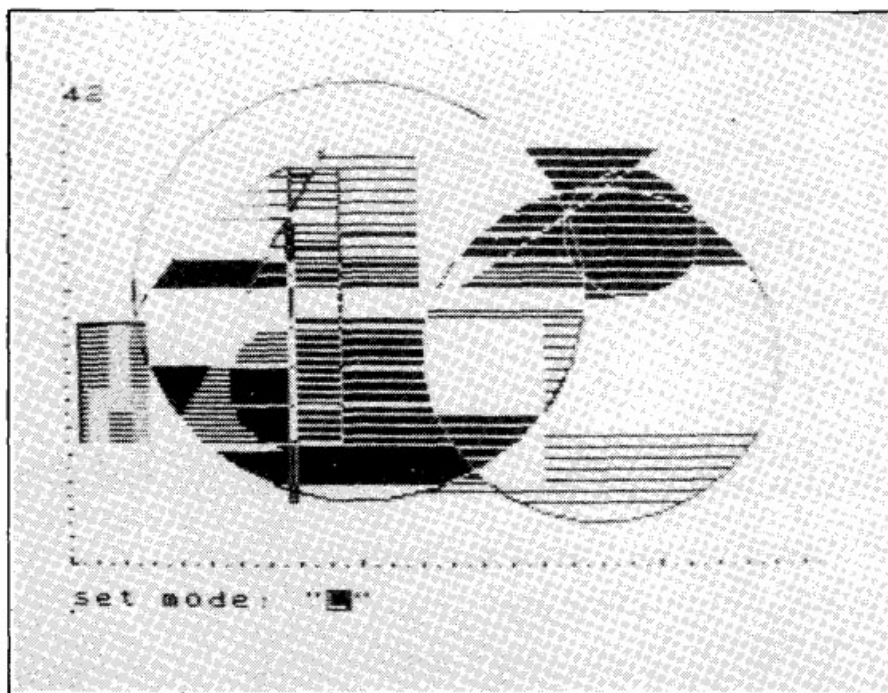
6. Curved line. You will be asked for the coordinates of the two ends of the line, and the largest distance from the curve to the imaginary straight line between these two points.
7. Shading. You will be asked whether the shape inside the current framing rectangle is to be blocked in or cross-hatched. Note that the current framing rectangle is the last one either drawn or set up as a frame. In this context, the word "inside" is used inclusively. In other words, if you draw a rectangle and then call up option 7, this rectangle will be shaded in, unless there is another figure inside it, in which case strange things will happen.
If you choose "cross-hatch" you will be asked for the distance between each line, and whether the hatching is to be straight or curved. If it is to be curved, you will be asked for the maximum distance between the curve and the imaginary straight line between the two end points. (Enter option 2 first.)
8. Save. This saves a picture on tape. You will be asked for a name.
9. Load. This loads a picture back from tape. You will be asked to supply its name.

Note: There is no need to implement the whole program all at once. The subroutines starting at lines 10000, 20000, 30000 and so on deal with options 1, 2, 3 etc. respectively. So if you want to start with something simple and just draw rectangles and circles you need not code lines 40000 onwards. Of course, if you do this, don't enter options 4-9 or you'll get an error message. There's one other thing to beware of: the routines starting at 40000, 60000 and 70000 all call the routine at 95000.

```

10 INPUT "set mode: "; m$
20 IF m$ = "f" THEN STOP
30 IF m$ = "c" THEN OVER 1
40 IF m$ = "d" THEN OVER 0: INPUT "colour: "; c: INK c
50 INPUT "option: "; op
60 GO SUB 10000 * op
70 GO TO 10

```



```

1000 FOR x = 0 TO 255 STEP 5
1010 PLOT x, 0
1020 IF x/10 = INT (x/10) THEN PLOT x, 1
1030 IF x/100 = INT (x/100) THEN PLOT x, 2
1040 NEXT x
1050 FOR y = 0 TO 175 STEP 5
1060 PLOT 0, y
1070 IF y/10 = INT (y/10) THEN PLOT 1, y
1080 IF y/100 = INT (y/100) THEN PLOT 2, y
1090 NEXT y
1100 RETURN

2000 INPUT "left column of rectangle:"; lc
2010 INPUT "right column:"; rc
2020 INPUT "bottom row:"; br
2030 INPUT "top row:"; tr
2040 INPUT "draw (d) or frame (f)"; m$
2050 IF m$ = "f" THEN RETURN
2060 PLOT lc, br
2070 DRAW 0, tr - br
2080 DRAW rc - lc, 0
2090 DRAW 0, br - tr
2100 DRAW lc - rc, 0
2110 RETURN

3000 INPUT "centre of circle; (column then row)"; cc, cr
3010 INPUT "radius:"; r
3020 CIRCLE cc, cr, r
3030 RETURN

4000 INPUT "one corner of segment; (column then row)"; c1, r1
4010 INPUT "other corner; (column then row)"; c2, r2
4020 INPUT "max. distance from curve to line:"; d
4030 GO SUB 9500
4040 PLOT c1, r1
4050 DRAW c2 - c1, r2 - r1
4060 DRAW c1 - c2, r1 - r2, a
4070 RETURN

5000 INPUT "one end of line; (column then row)"; c1, r1
5010 INPUT "other end; (column then row)"; c2, r2
5020 PLOT c1, r1

```



```

5030 DRAW c2 - c1, r2 - r1
5040 RETURN

6000 INPUT "one end of curved line; (column then row):"; c1, r1
6010 INPUT "other end; (column then row):"; c2, r2
6020 INPUT "max. distance from straight:"; d
6030 GO SUB 9500
6040 PLOT c1, r1
6050 DRAW c2 - c1, r2 - r1, a
6060 RETURN

7000 INPUT "block in (b) or hatch (h):"; m$
7010 IF m$ = "b" THEN LET s = 1: LET a = 0: GO TO 7070
7020 INPUT "width of hatch:"; s
7030 INPUT "straight (s) or curved (c):"; m$
7040 IF m$ = "s" THEN LET a = 0: GO TO 7070
7050 INPUT "max. distance from straight:"; d
7070 FOR r = br TO tr STEP s
7080 FOR c = lc TO rc
7090 IF POINT (c, r) = 1 THEN GO TO 7120
7100 NEXT c
7110 GO TO 7190
7120 LET c1 = c
7130 FOR c = rc TO lc STEP -1
7140 IF POINT (c, r) = 1 THEN GO TO 7160
7150 NEXT c
7160 LET c2 = c
7170 PLOT c1, r
7175 IF m$ = "c" THEN LET r2 = r: LET r1 = r: GO SUB 9500
7180 DRAW c2 - c1, 0, a
7190 NEXT r
7200 RETURN

8000 INPUT "enter name for picture to be saved:"; p$
8180 SAVE p$ SCREEN$
8020 RETURN

9000 INPUT "enter name of picture to be loaded:"; p$
9010 LOAD p$ SCREEN$
9020 RETURN

9500 LET l = 0.5 * SQR ( (c2 - c1) * (c2 - c1) + (r2 - r1) * (r2 - r1) )
9510 LET a = ASN (2 * l * d / (d ↑ 2 + l ↑ 2) ) * 2
9520 RETURN

```

Modifications and improvements

1. At the moment there are no tests to ensure that the user doesn't try to draw off the screen. Insert some.
 2. Some commands have the effect of destroying the horizontal scale, so that you have to regenerate it using option 1. How could you avoid this problem?
 3. How about writing a routine to generate vertical hatching? (Or even diagonal hatching?)
-

*At Harold Hustler's Vegas Venue
they don't play ordinary roulette;
they play:*

Linette

This is roulette played in a straight line.

```
10 LET w = 100
20 INPUT "Faites vos jeux "; b$
30 CLS
40 INPUT "Size of bet "; a
50 LET w = w - a
60 FOR n = 0 TO 9
70 PRINT AT 10, 2 * n + 6; CHR$(48 + n)
80 NEXT n
90 LET r = INT(5 * RND) + 5
100 LET d = INT(10 * RND)
110 LET r = r * 10 + d
120 FOR n = 0 TO r
130 LET x = n - 10 * INT(n/10)
140 PRINT AT 10, 2 * x + 6; CHR$ 143
145 BEEP .05, x
150 PRINT AT 10, 2 * x + 6; CHR$(48 + x)
160 NEXT n
170 PRINT AT 10, 2 * x + 6; FLASH 1; CHR$(48 + x)
180 IF b$(1) = "e" THEN GO TO 210
190 IF b$(1) = "o" THEN GO TO 220
200 IF VAL b$(1) = d THEN LET w = w + 10 * a
205 GO TO 240
210 IF INT(d/2) = d/2 THEN LET w = w + 2 * a
215 GO TO 240
```

```

220 IF INT (d/2) < > d/2 THEN LET w = w + 2 * a
240 PRINT AT 16, 9; w; "□ chips □"
250 IF w > = 0 THEN GO TO 20
260 PRINT "Harry's Mob will be round in themorning!"

```

Program notes

1. After RUN, you can bet either on EVEN, ODD, or a number between 0 and 9, by typing in your bet. (Any word starting e will be read as "even", anything starting o as "odd".)
 2. You then say how much you are betting by inputting a number. You start with £100, set in line 10.
 3. If you win you get back twice the stake on e or o; ten times the stake on a number. This theoretically makes it a "fair" game.
 4. You can bet more than you have in the kitty; but beware if you lose!
 5. In line 260 there is *no* space in themorning. Why? Put one in and see.
 6. To stop, on a character input, hit DELETE and then STOP. On a numerical input STOP will do.
-

*Meanwhile, two blocks up The Strip,
Harold's arch-rival Samuel Hammerhead
has had an original idea . . .*

Circular Linette

This is Linette played in a circle. Hmmm . . .

```

10 LET w = 100
30 CLS
40 CIRCLE 123, 91, 84
50 CIRCLE 123, 91, 60
60 FOR n = 0 TO 9
70 PRINT AT 10 + 9 * COS (n * PI / 5), 15 + 9 * SIN (n * PI / 5); CHR$ (48 + n)
80 NEXT n
82 INPUT "Faites vos jeux, kiddo!"; b$
84 INPUT "How much ya wanna bet?"; a
86 LET w = w - a
90 LET r = INT (5 * RND) + 5
100 LET d = INT (10 * RND)
110 LET r = r * 10 + d
120 FOR n = 0 TO r
130 LET x = n - 10 * INT (n/10)
140 PRINT AT 10 + 9 * COS (x * PI / 5), 15 + 9 * SIN (x * PI / 5); CHR$ 143
145 BEEP .05, x

```



```

150 PRINT AT 10 + 9 * COS (x * PI / 5), 15 + 9 * SIN (x * PI / 5); CHR$ (48 + x)
160 NEXT n
170 PRINT AT 10 + 9 * COS (x * PI / 5), 15 + 9 * SIN (x * PI / 5);
    FLASH 1; CHR$ (48 + x)
180 IF b$ (1) = "e" THEN GO TO 210
190 IF b$ (1) = "o" THEN GO TO 220
200 IF VAL b$ (1) = d THEN LET w = w + 10 * a
205 GO TO 240
210 IF INT (d/2) = d/2 THEN LET w = w + 2 * a
215 GO TO 240
220 IF INT (d/2) < > d/2 THEN LET w = w + 2 * a
240 PRINT AT 10, 10; w; "□ chips □ □"
250 IF w > = 0 THEN GO TO 82
260 PRINT FLASH 1; "Steer clear of Sammy the Shark!"

```

Program notes

1. Instructions for use are just like LINETTE, previous program.
2. PI is key M in extended mode.
3. Samuel Hammerhead is a bit more flash than Harold Hustler.

*Dot-dot-dot; dash-dash-dash;
dot-dot-dot . . .*

Automatic Morse

This is an example of the way to use BEEP. It accepts a message from the keyboard, and turns it into Morse code. The message, and the code, are displayed on the screen; simultaneously the dots and dashes are produced from the Spectrum's loudspeaker.

AUTOMATIC MORSE

```

5 CLS
10 INPUT "Enter message □": m$
20 FOR i = 1 TO LEN m$
30 LET c = CODE m$ (i)
40 IF c < 32 OR c > 32 AND c < 65 OR c > 90 AND c < 97 OR
    c > 122 THEN GO TO 120
50 IF c > 96 THEN LET c = c - 32
60 LET c = c - 64

```

```

100 IF c = -32 THEN PAUSE 40: PRINT "□"
110 IF c > 0 THEN PRINT CHR$ (c + 64); "□"; : GO SUB 200
120 NEXT i
130 STOP
200 LET k$ = a$ (c)
210 LET t = 1
220 IF k$ (t) = "1" THEN PRINT INK 6; "."; : BEEP .1, 10
230 IF k$ (t) = "2" THEN PRINT INK 3; "-"; : BEEP .3, 10
240 IF k$ (t) = "□" OR t = 4 THEN PRINT "□": RETURN
250 LET t = t + 1: GO TO 220
500 REM Input Routine
510 DIM a$ (26, 4)
520 FOR r = 1 TO 26
530 INPUT a$ (r)
540 NEXT r

```

Program notes

1. To set up the array a\$ which stores the Morse code, you must start with GO TO 500. You then INPUT 26 strings of 1's and 2's: these are the Morse codes for the letters A–Z, with 1 standing for a dot and 2 for a dash. Here are the strings to INPUT, in order:

Letter	INPUT this string	Letter	INPUT this string
A	12	N	21
B	2111	O	222
C	2121	P	1221
D	211	Q	2212
E	1	R	121
F	1121	S	111
G	221	T	2
H	1111	U	112
I	11	V	1112
J	1222	W	122
K	212	X	2112
L	1211	Y	2122
M	22	Z	2211

Don't INPUT the letters: they're just there to remind you how far you've got.

2. After setting up a\$, press GO TO 5. Do NOT press "RUN"—you'll clear out the variables you've just laboriously typed in. Never use RUN on this program.

3. When asked for a message, type one in. If it's more than 22 characters long, you'll need to SCROLL. The program treats lower and upper case letters as the same, and ignores everything else. For example, you could type in "Hello".
4. The computer will now display the message, in letters and Morse; and it will BEEP the dots and dashes.
5. Note the way a string array is used to hold the Morse code. This is a very common use for arrays: as a "look-up table" to convert from one system of code to another one.

Project

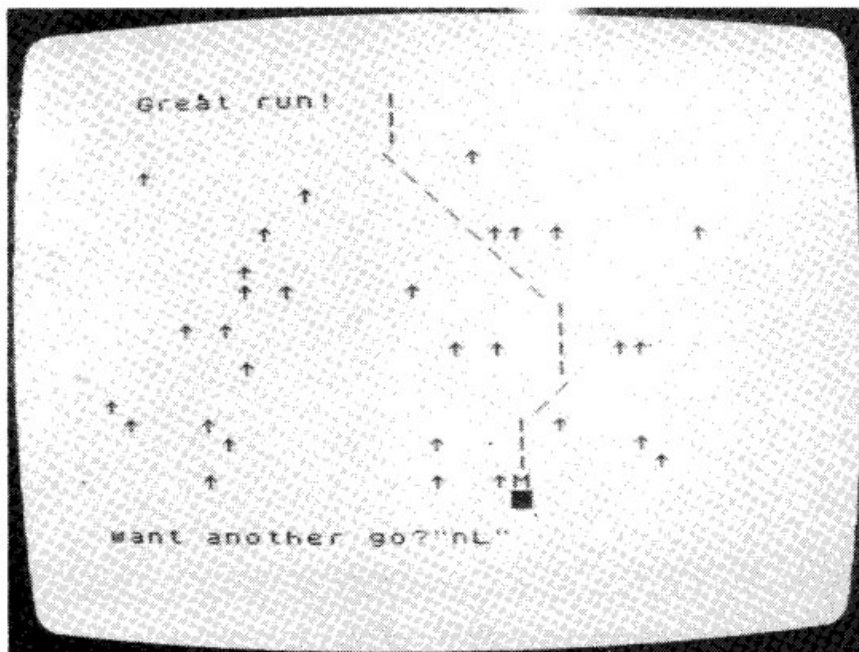
Modify the program so that it can cope with numbers as well as letters. The Morse code for numbers 0-9, in order, is:

-----, .- - - -, .. - - -, ... - -, -, - -, -----, -----

*Go skiing with the Spectrum
in the safety of your own home:*

St. Moritz

You have to ski through the forest, avoiding the trees to arrive at the ski-lift. You are an "M" (bird's-eye view of a crouching skier, get it?) and the ski-lift is a black blob (I don't know why, either!). It's downhill all the way, so if you do nothing you'll simply fall from top to bottom, or until you hit a tree. You hit "z" to veer left and "m" to go right.




```

1  CLS: RANDOMIZE
2  LET n = 30: LET sc = 14
10  FOR i = 1 TO n
20  LET r = INT (RND * 18) + 3
30  LET c = INT (RND * 32)
40  PRINT AT r, c: "↑"
50  NEXT i
60  PRINT AT 21, 20: "■"
70  PRINT AT 0, sc: "M"
80  PAUSE 200
90  FOR r = 1 TO 20
100  LET d$ = INKEY$
103  IF d$ = " " THEN PRINT AT r - 1, sc: "|"
104  IF d$ = "m" THEN PRINT AT r - 1, sc: "\": LET sc = sc + 1
105  IF d$ = "z" THEN PRINT AT r - 1, sc: "/": LET sc = sc - 1
120  IF SCREEN$(r, sc) = "↑" THEN PRINT AT r, sc: "* crash": GO TO 200
130  PRINT AT r, sc: "M"
135  PAUSE 2
140  NEXT r
150  IF sc < 22 AND sc > 18 THEN PRINT FLASH 1: AT 0, 2: "Great run!":
    GO TO 200
160  PRINT AT 0, 3: "You've got a walk to the ski-lift"
200  INPUT "want another go?": q$
210  IF q$ = "yes" THEN GO TO 1
220  CLS: PRINT AT 10, 2: "Et maintenant l'apres-ski . . ."

```

Modifications and improvements

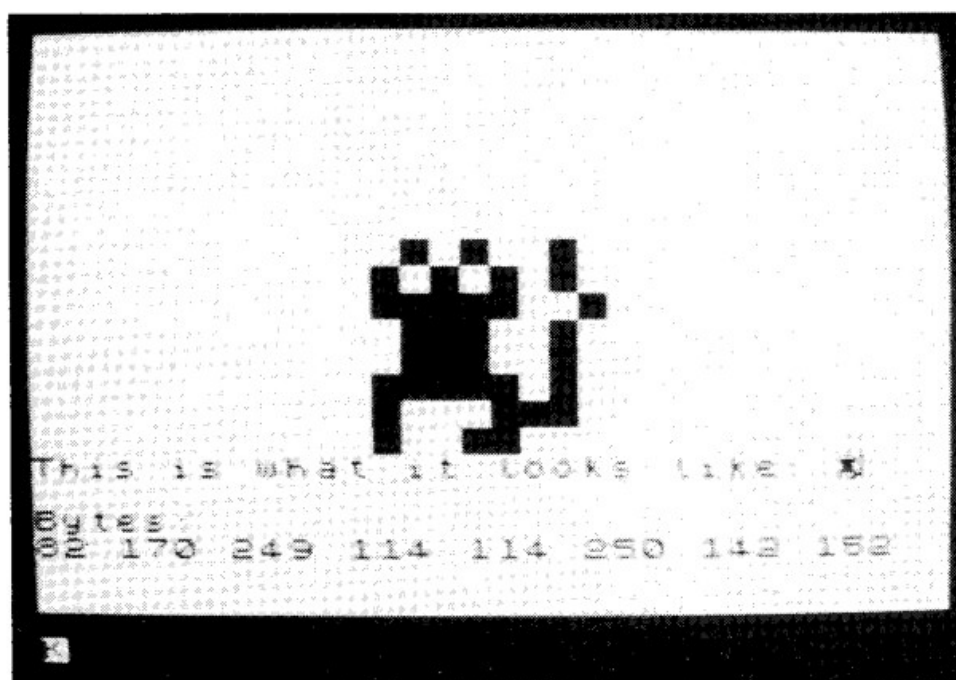
1. The trees could look more tree-like. Experiment with CHARACTER BUILDER to produce a spruce or concoct a conifer. While you're at it, how about giving the skier twin tracks; and do *something* about that awful ski-lift.
2. The number of trees is fixed at 30. Allow the user to vary it to get an easier or more difficult game.
3. There's a "PAUSE 2" at line 135 in the loop to give a reasonable speed of descent. You could allow the user to alter this. (I don't mean directly: set up a series of possible PAUSE values and allow the user to enter, say, a difficulty rating of 1–5, each of which selects a different PAUSE value.)
4. Why not modify the difficulty automatically, by looping the whole program and increasing the number of trees and/or descent speed every time the user makes a successful run?

User-defined graphics made easy:

Character Builder

This program lets you design a graphics character on the TV screen, large scale, and then enters it into the computer as the user-defined character corresponding to a letter of your choice. It also prints out the contents of the eight bytes giving the rows of the character, for re-use in subsequent programs.

```
10 DIM k (8, 8)
20 FOR i = 8 TO 15
30 PRINT AT i, 12; "....."
40 NEXT i
50 LET x = 0: LET y = 0
55 GO SUB 500
60 INPUT "Pixel Value ☐"; v
65 IF v <> 0 AND v <> 1 THEN GO TO 60
70 LET k (x + 1, y + 1) = v
80 PRINT AT x + 8, y + 12; INVERSE v; "☐"
90 LET y = y + 1
100 IF y = 8 THEN LET y = 0: LET x = x + 1
110 IF x = 8 THEN LET x = 0: LET y = 0: GO TO 200
120 GO TO 55
200 INPUT "Is this right?": q$
```



```

210 IF q$ = " " THEN GO TO 200
220 IF q$ (1) = "y" THEN GO TO 600
230 INPUT "Use arrow keys to move cursor"; j$
235 GO SUB 500
240 IF INKEY$ < > " " THEN GO TO 240
250 IF INKEY$ = " " THEN GO TO 250
260 LET i$ = INKEY$
270 IF i$ = "0" OR i$ = "1" THEN LET k (x + 1, y + 1) = VAL i$:
    PRINT AT x + 8, y + 12; INVERSE VAL i$; "□": GO TO 200
275 PRINT AT x + 8, y + 12; INVERSE k (x + 1, y + 1); "□"
280 IF i$ = "5" THEN LET y = y - 1 + (y = 0)
290 IF i$ = "6" THEN LET x = x + 1 - (x = 7)
300 IF i$ = "7" THEN LET x = x - 1 + (x = 0)
310 IF i$ = "8" THEN LET y = y + 1 - (y = 7)
320 GO SUB 500
330 GO TO 240
500 PRINT AT x + 8, y + 12; FLASH 1; INK 2; "*": RETURN
600 INPUT "Which Letter?"; f$
610 FOR n = 1 TO 8
620 LET t = k (n, 1)
630 FOR j = 2 TO 8
640 LET t = 2 * t + k (n, j)
650 NEXT j
660 POKE USR f$ + n - 1, t
670 NEXT n
700 PRINT "This is what it looks like: □"; CHR$ (CODE f$ + 47)
710 PRINT, , "Bytes:"
720 FOR n = 1 TO 8
730 PRINT PEEK (USR f$ + n - 1); "□";
740 NEXT n

```

Program notes

- Initially, the screen shows an 8×8 array of dots, and a flashing cursor; it asks for a Pixel Value. You input 0 or 1: if you input 0, a dot is blanked out; if a 1, it is blacked in as a square. The cursor runs automatically through the entire array, row by row; your inputs build up your first try at the character.
- It then asks "Is this right?". If you input "y"—or anything starting with y—it goes on to load the character into the machine (see note 3. below). If you input anything else, it prints the message "Use arrow keys to move cursor". You must now press ENTER, and the cursor reappears. It can be controlled from the keyboard using keys 5, 6, 7, 8 to move it in the directions of the arrows. Once you have moved it to the right place, an input of 0 or 1 from the keyboard will blank out or black in the corresponding square. The message "Is this right?" reappears, and you can make new changes if you wish.

3. Once it is right, and you've typed "y", the machine will ask you which letter you wish your character to correspond to. Recall that each user-defined graphic character is accessed from the keyboard by using a letter (apart from v, w, x, y, z): you have to decide which one. For example, you might choose "s": now your new character is stored in the right place for graphics-mode s to produce it. You can also call it by code, as CHR\$(162).
 4. The computer also lists the eight bytes corresponding to the rows of the character: this makes it easy to set it up at any later date, if you note the numbers down, by POKEing them into place as described on page 94.
 5. From now on, until you switch off, your new character will be stored as graphics-mode "s". You can load other characters into other positions, by RUNning the program again.
 6. So that you can check if everything is correct, you get a sample print-out of your new character. If you don't like it, press GO TO 200. The character isn't printed out, but as you sweep the cursor through the area, the squares reappear; and you can make new changes.
-

Up at the rifle range on a windy day: how high can you score?

Target Practice

```

10 LET d = 100
20 LET w = 40 * (.5 - RND)
30 LET v = 1000
40 LET s = 0
50 LET c = 0
100 FOR i = 1 TO 8
110 CIRCLE 127, 95, 8 * i
120 NEXT i
200 IF c = 11 THEN STOP
205 PRINT AT 20, 0; "Elevation = ";
210 INPUT e: PRINT c
220 PRINT "Deviation = ";
230 INPUT f: PRINT f
240 LET e = e * PI / 180
250 LET f = f * PI / 180
300 LET t = d / (v * COS e)
310 LET h = v * t * SIN e - 4.9 * t * t
320 LET k = w * d / 100 + d * SIN f
330 LET h0 = 8 * h + 95 + 40 * (.5 - RND)

```

```

340 LET k0 = 8 * k + 127 + 40 * (.5 - RND)
350 IF h0 < 0 OR h0 > 175 OR k0 < 0 OR k0 > 255 THEN GO TO 500
360 GO SUB 390
370 PRINT AT 20, 0: "□□□□□□□□□□□□□□", ., [16 spaces]
    "□□□□□□□□□□□□□□" [16 spaces]
375 LET c = c + 1
380 GO TO 200
390 INK 3
400 PLOT k0 - 8, h0: DRAW 16, 0: PLOT k0, h0 - 8: DRAW 0, 16
405 BEEP .1, 5
410 CIRCLE k0, h0, 6
415 IF c < 1 THEN GO TO 480
420 LET q = SQR ( (k0 - 127) * (k0 - 127) + (h0 - 95) * (h0 - 95) )
430 LET q = INT (q/8)
440 LET q = 100 - 10 * q
450 IF q < 30 THEN LET q = 0
460 LET s = s + q
470 PRINT AT 0, 25: c; TAB 28: s
480 INK 0
490 RETURN
500 PRINT AT 0, 0: "Off screen"
510 PAUSE 50
520 PRINT AT 0, 0: "□□□□□□□□□□" [10 spaces]
530 PRINT AT 20, 0: "□□□□□□□□□□□□□□", ., [16 spaces]
    "□□□□□□□□□□□□□□" [16 spaces]
535 PRINT AT 0, 25: c
540 LET c = c + 1
550 GO TO 200

```

Program notes

1. After RUN, you are asked to input an elevation (in degrees). Somewhere close to zero is best, say between -2 and +2.
2. Next you are asked for a sideways *deviation* (to compensate for a side-wind). This should be between about -10 and +10. NB: you are not told which way the wind is blowing!
3. You get one ranging shot, which doesn't count; and then 10 shots. The wind varies slightly between shots.
4. The score, and the number of shots used, is printed out at the top right—number of shots first, then score.

Other titles of interest

PEEK, POKE, BYTE & RAM! Basic Programming for the ZX81

Ian Stewart & Robin Jones

'Far and away the best book for ZX81 users new to computing'—*Popular Computing Weekly*

'... the best introduction to using this trail-blazing micro'—*Computers in Schools*

'One of fifty books already published on the Sinclair micros, it is the best introduction accessible to all computing novices'—*Laboratory Equipment Digest*

The ZX81 Add-On Book

Martin Wren-Hilton

Machine Code and better Basic

Ian Stewart & Robin Jones

ComputerPuzzles: For Spectrum and ZX81

Ian Stewart & Robin Jones

Games to Play on Your ZX Spectrum

Martin Wren-Hilton

Available from October '82

Further Programming for the ZX Spectrum

Ian Stewart & Robin Jones

Spectrum in Education

Eric Deeson

Easy Programming for the BBC Micro

Eric Deeson

Further Programming for the BBC Micro

Alan Thomas

Plus lots more! Keep your eye on the magazines for up-to-date news.

Tapes available soon!

If you've just bought — or are still thinking of buying — a ZX Spectrum, then this is the book for you!

We'll show you, in easy steps, how to get started on writing your own programs. Topics include:

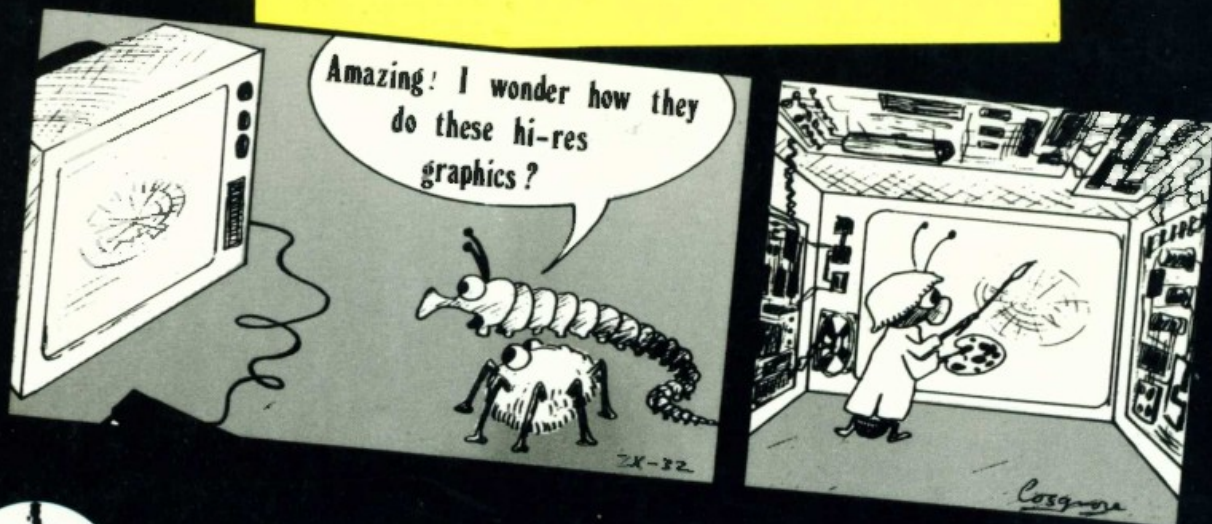
- graphics
- strings
- data
- debugging techniques
- son et lumière
- programming style

And, in case you're impatient to have a game or two on your new toy, why not have a go at

Brickbat
Fruit machine
Picasso
Automatic Morse

or any of the other 26 "Prepacked Programs" listed at the end of the book. They are all ready to copy and RUN, and you'll find that, by working through the text, you'll come to understand how these programs were put together.

Happy computing — it's easy when you know how!



Shiva Publishing Limited

ISBN 0 906812 23 2

UK price £5.95 net