Further Programming for the ZX SPECTRUM

SHIVA's friendly micro series

Ian Stewart and Robin Jones

Further Programming for the ZX SPECTRUM

Ian Stewart
Mathematics Institute, University of Warwick

Robin Jones
Computer Unit, South Kent College of Technology

Shiva Publishing Limited

Contents

Preface

You own a Sinclair ZX Spectrum and you feel pretty confident about using it. You know what the keys do and you can string twenty or thirty lines of BASIC together and make them work. You've worked your way through the *Manual* and an introductory book. You've typed in dozens of programs from the magazines and discovered that the shorter ones all do the same thing and the longer ones, if they haven't got errors all over them, take hours and hours of careful work—and for a mere five pounds you can buy a cassette which produces more impressive results. Which would be fine except you don't want to keep spending five pounds to buy other people's software—you want to produce your own.

So what next?

You've still got some way to go before you can write Machine Code arcade-quality games or programs to display the Night Sky at any time between 4000 BC and 6000 AD; and while this book may start you along that road, it certainly won't take you all the way. What it *will* do is help you to expand both your own capabilities and those of the machine.

There are three main directions to explore.

One might be described as "Theory of Computation": how to develop techniques for improving your programs. As far as *this* book goes, we've taken a fairly practical view of what constitutes theory: that is, we've concentrated on specific features of the Spectrum, such as its colour facilities and its graphics, and dug a little deeper into the machine. You'll find out more about control characters, user-defined functions, user-defined graphics, the systems variables, and the display and attributes files—*and* how to make good use of them.

Second is "Machine Enhancement". By writing suitable utility programs you can equip your Spectrum with facilities that the bare machine does not possess. Quick line-renumbering of BASIC programs (our routine lets you choose a block out of a program and renumber that on its own—great for tidying up subroutines). Plotting graphs without having to worry about points going off-screen. Automatic block-fill of line-drawings. An effective system for handling large quantities of data held on cassette tape as *files*, which could be used as the basis of a practical record-keeping system for business or the home. In easy stages we take you from a simple Cassette File System to a Data Management System.

Third . . . well, have you noticed that whenever you ask a computer enthusiast "very nice, but what can you *do* with it?" he tends to change the subject? It's as if the major aim of computing is to do *more* computing. Art for art's sake, Computers as a Way Of Life. But wouldn't it be nice to actually *use* the computer to do *something else*? You'll find some suggestions here: maps, star charts, psychological experiments, simple statistics, cryptography and cryptanalysis, symbol manipulation.

Two areas we don't go into here are Machine Code and "pure" theory—topics like data structures and structured programming. We deal with those elsewhere, in *Machine Code and Better Basic* and in *Spectrum Machine Code*.

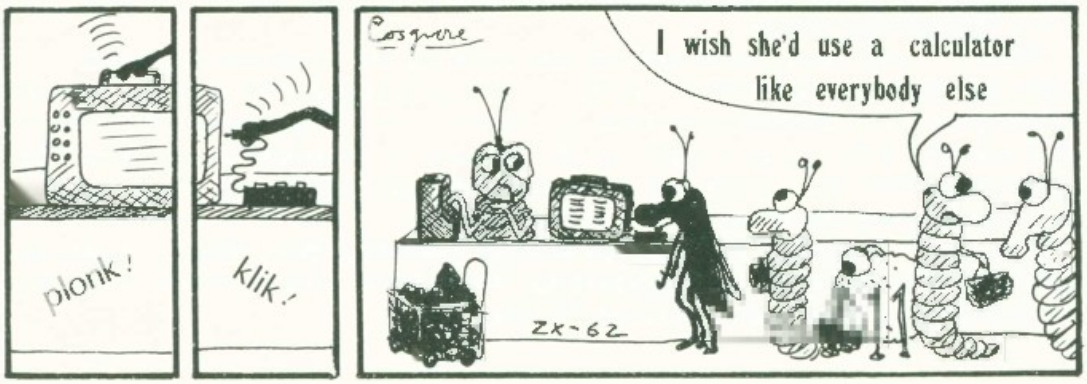
Our primary objective is not to produce highly polished "oven-ready" programs. The main emphasis is on the painful but satisfying process of developing an initial idea into a program that *works*. Instead of just presenting the final result, we sometimes describe routines that are then modified, rewritten, revised, or scrapped entirely and replaced. After all, that's how any non-trivial program gets written, and it's misleading to pretend otherwise. We're not trying to give you the impression that writing programs can or should be painless and easy. The important point is that *everybody* makes mistakes, so there's no reason to lose heart when *you* do. The trick is to recognize the errors and to put them right. Of course, any methods that help cut down the chances of making mistakes are well worth having.

Additionally, however, some of the generally applicable utility routines are also listed separately in appendices, so that it's not necessary to wade through the descriptions of their construction to be able to use them. If all you want to do is copy the listing and run the program, you can do it.

As in all our books, we've tried to keep the explanations clear and simple. This book is *not* a rigidly structured course: it's designed for you to dip in at random. Some chapters do depend on earlier ones to some extent, but it's always obvious when this is the case. So start by thumbing through to see which items have a particular appeal to your own tastes, and have a go at those first. You'll find them very instructive.

Fun, too.

So far, we've referred to ourselves as "we"—but as in *Easy Programming* we found this didn't always work out later. So from now on, we'll refer to ourselves in the singular, as "I". Whenever we say "we", we'll mean "I and the reader". It may sound a silly idea, but it's actually more informative that way.



The more work you're prepared to put in, the more your Spectrum can be trained to perform. Spend a couple of hours, and you can have a . . .

1 Map of the World

That's one possibility. The same technique will let you draw, and SAVE, hi-res line drawings such as pictures of Isaac Newton or Olivia Newton-John, or Martian landscapes for use in Space Inveiglers programs.

It's easy; but it takes time. Here's a picture of a Spectrum World Map, just to convince you that the results will amply justify the time spent.

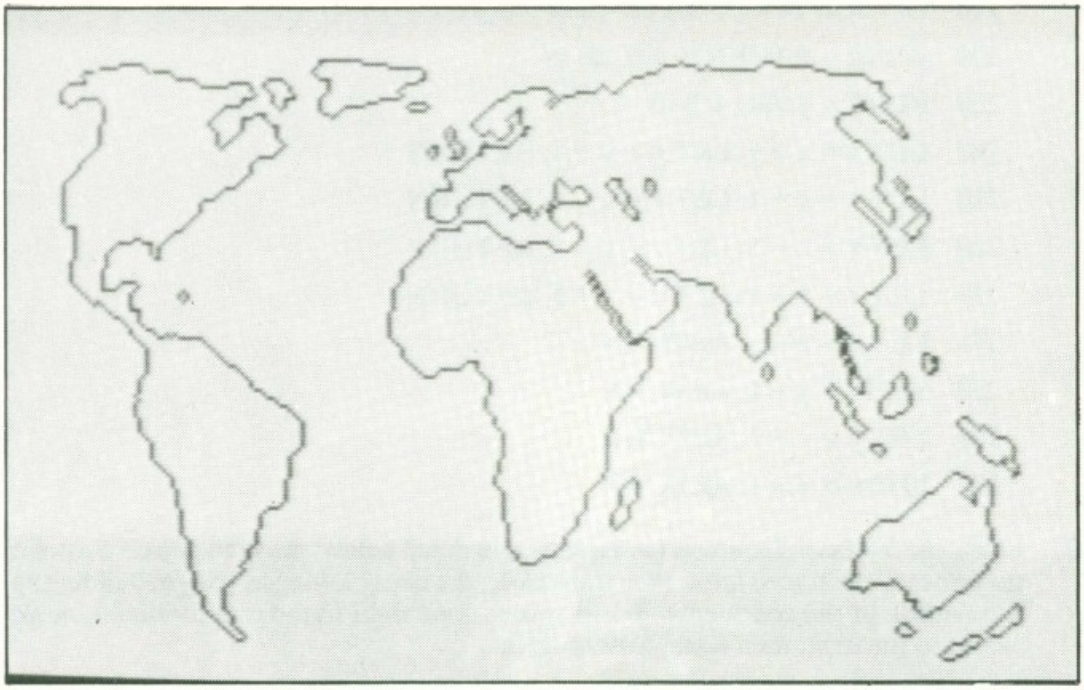


Figure 1.1 An outline map of the world, produced on a Spectrum.

The hard way to get this into the machine is to pore over grids of latitude and longitude, copying out coordinates, and feeding them into a drawing routine.

The way I actually did it is crude, but effective.

1. Cut a piece of transparent plastic—such as part of a polythene bag—to the size of your TV screen.
2. Mark on it an outline of the central area of the Spectrum display—the part you can PRINT or PLOT to. The easy way to find this is to type BORDER Ø.
3. Find a map of the world the right size to fit into this.
4. Trace it on to the polythene using a felt-tipped pen. A fairly crude image will do.
5. Let the ink dry, and being careful not to rub it off, stick the map to the front of the TV, lining it up with the central area, using sticky tape.

That's the "hardware" requirement for this method. Now for the software . . .

SKETCHPAD

6. Type in a Sketchpad program that lets you control a moving pixel on the screen, from the keyboard, so that it either PLOTs or moves. Some means of erasing mistakes is worth having too. Here's one that will do the job: you can of course make it more sophisticated if you feel like it.

```
10  LET x = 0: LET y = 175
20  OVER 1
30  LET flag = 0
40  INPUT d$
50  LET x0 = x: LET y0 = y
60  IF CODE d$ < 60 THEN GO TO 100
70  IF d$ = "m" THEN LET flag = 0
80  IF d$ = "p" THEN LET flag = 1
90  GO TO 40
100 REM Keyboard response
110 GO SUB 10 * CODE d$ - 290
120 IF flag = 0 THEN PLOT x0, y0
130 PLOT x, y: GO TO 40
200 LET x = x + 1: LET y = y + 1: RETURN
210 LET x = x + 1: LET y = y - 1: RETURN
220 LET x = x - 1: LET y = y - 1: RETURN
230 LET x = x - 1: LET y = y + 1: RETURN
240 LET x = x - 1: RETURN
250 LET y = y - 1: RETURN
260 LET y = y + 1: RETURN
270 LET x = x + 1: RETURN
```

7. Using the keyboard controls (as explained in detail below) move the pixel to a point underneath the traced lines, then trace along the lines plotting as you go, building up the outline of the continents. When you've gone right round one continent, move across to the next, then start plotting again.

See, it really is easy. But the results can be magnificent, if you take the time and trouble to be careful.

USING SKETCHPAD

The program can be in one of two "modes":

m: MOVE the pixel to a new position;
p: PLOT the current position as you move to it.

It starts in "m". At any stage you can change the modes by typing in the symbol "p" or "m".

The moves are controlled by keys 1–8, as follows:

4	7	1
5	*	8
3	6	2

The pixel moves from its current position (*) one place up, down, sideways, or diagonally, according to the numbers. (The order may look odd: the idea is that the "arrow" keys 5–8 work as usual, and the "diagonal" moves 1–4 start at 1 o'clock and move clockwise.)

The program as it stands requires you to ENTER each number or mode symbol. You can use INKEY$ if you prefer; but this way there's a chance to check you've hit the right key before doing any damage.

Experiment with the moves. Because of the OVER 1 command, if you PLOT twice in the same place, the pixel blanks out. This lets you erase mistakes. However, you should bear two things in mind:

1. To move on to a new region, make one move *away* from your finished curve (in a direction that won't run into it) *before* changing to mode "m".
2. On arrival at the new piece of curve, do not press "p" until your pixel is exactly aligned with it.

If you make mistakes in choosing modes, you tend to get isolated pixels sitting on the screen. To get rid of these, go into mode "m"; move until you hit them (and obliterate them); *go into mode* "p"; move one space; go back to mode "m". Try it.

Don't expect a polished result in ten minutes.

SAVE IT

Once you're happy, SAVE the map on tape: you won't need to spend two hours glued to the screen ever again. Just STOP the Sketchpad program; then key in as a direct command:

SAVE "map" SCREEN$

To LOAD it back in, go through the usual routine and use

LOAD "map" SCREEN$

Some of the later chapters of this book assume that you've drawn a map (it can be a lot simpler than the one in my photo) and saved it on tape. So get cracking!

Sometimes the main problem in writing a program is deciding just what it is that the machine has to do . . . as in this graphics utility program that shades in regions of the screen—subject to a few clauses in the fine print.

2 Block Filling

The original idea was: "Wouldn't it be nice to have a World Map screen display in which the land areas were *blacked in*?" And the immediate thought was "Using the Sketchpad program, it'll take *weeks*!" So of course the idea was to get the Spectrum to do all the work.

It sounds easy, at first. And in simple cases, it is. But you can't tell the Spectrum "Find the closed curves and block in the inside", because it doesn't know what a closed curve is, nor does it have any Inside Knowledge. Nor, indeed, does that approach look workable on the computational level.

Let's start with an easy case, and work up to the map in stages. The simplest task is to fill in a single closed region, such as a circle or a polygon. Here's a working title:

POLYFILLER

Suppose we have a *single* polygon drawn on the screen. Ignore practicalities for the moment: consider the theoretical question "What steps must the computer carry out in order to fill the outline in?"

The answer's easy:

1. For a given horizontal row, find the left-hand point of the polygon by searching along from the left.
2. Find the right-hand end by searching from the right.
3. Join them up by a horizontal line.
4. Repeat for each row.

The way to see if a point has been PLOTted is to use the function POINT (*Easy Programming**, page 36). The value of POINT (x, y) is 1 if (x, y) has been PLOTted, 0 if not. So the program we want is this (where I'm using an italic *l* to distinguish from the numeral 1):

```
10  FOR y = 0 TO 175
20  LET xl = 0
30  IF POINT (xl, y) = 1 THEN GO TO 60
40  LET xl = xl + 1
50  IF xl < = 255 THEN GO TO 30
60  LET xr = 255
70  IF POINT (xr, y) = 1 THEN GO TO 100
80  LET xr = xr − 1
```

* *Easy Programming for the ZX Spectrum* by Ian Stewart and Robin Jones, Shiva.

```
90  IF xr >= 0 THEN GO TO 70
100 IF xl >= 256 THEN GO TO 120
110 PLOT xl, y: DRAW xr - xl, 0
120 NEXT y
```

For a test, feed this in; then enter by direct command:

```
CIRCLE 127, 87, 87
```

(say) and then

```
GO TO 10
```

(*not* RUN, which erases the screen!).

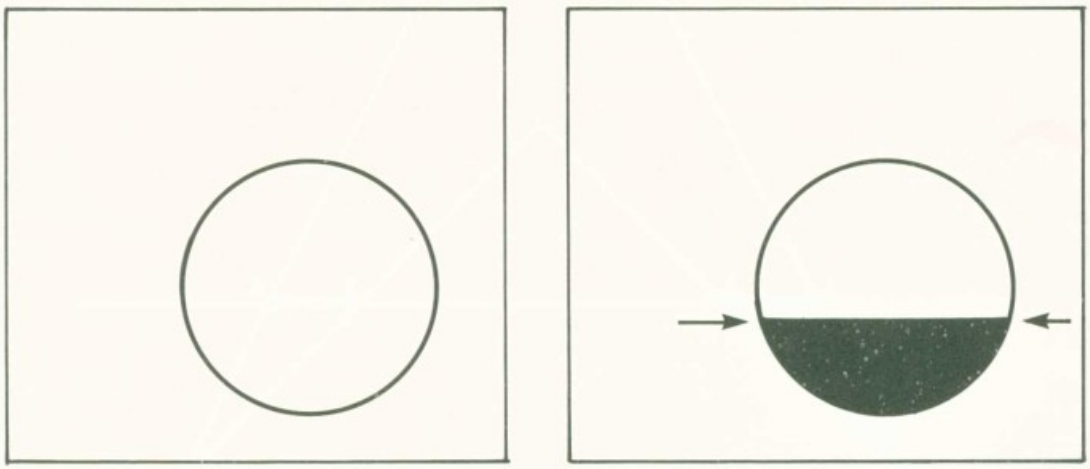


Figure 2.1 On easy shapes, shading from the leftmost point to the rightmost will work.

It's slow, to be sure (any shade-in routine is going to be, the amount of computation is bound to be lengthy because there are 45,056 points on the screen to worry about) but it works.

If you combine it with the polygon-drawing routine in Chapter 13 so that it first draws a single polygon, then shades it in, you'll find it continues to work.

THE SNAGS

However, it fails dismally when there are several regions that need shading (and indeed in other cases too). Try:

```
CIRCLE 50, 50, 49: CIRCLE 160, 50, 49
GO TO 10
```

That's not what's intended, is it?

What it's doing is finding the left-hand point of one circle, then the right-hand point of *the other*, and joining *those*. One way out is to try to work out which closed curve is which, but that's messy and long. And, in any case, we've still got problems even if there's only one curve. Try this:

```
PLOT 100, 50: DRAW 50, -50: DRAW -50, 100:
DRAW -50, -100: DRAW 50, 50:
GO TO 10
```

It's a single closed polygon, shaped a bit like an arrowhead; and the program shades in too much.

The reason here is that certain horizontal lines meet the shape in more than just two points. For example, a line like the one in Figure 2.2 will meet it in *four* points. And we only want to shade between the 1st and 2nd; and the 3rd and 4th: *not* between the 2nd and 3rd.

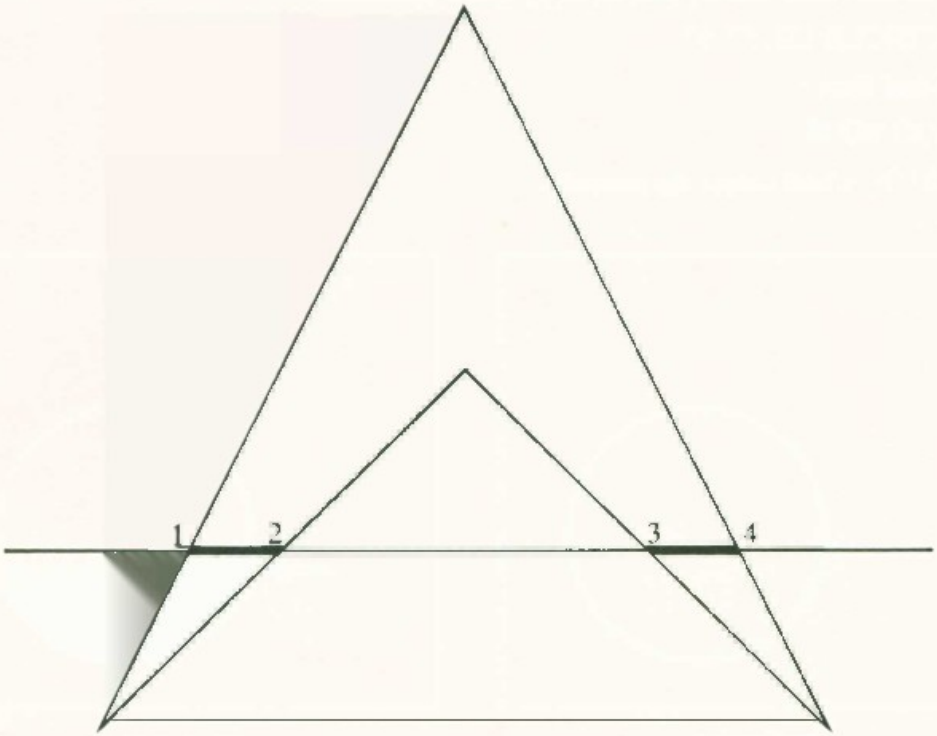


Figure 2.2 On more complicated shapes, it won't!

Which suggests we do this:

1. Track along the row looking for points on the curve, and list them all.
2. Draw from the 1st to the 2nd, the 3rd to the 4th, . . . and in general from the odd-numbered ones (2 * i + 1) to the even-numbered ones (2 * i + 2) as i runs from 1 to whatever it is.

If you think this will do the trick, try writing a program to implement it. Then test it out on the arrowhead shape above.

Whoops.

It goes wrong in the first line. There are only two points here; but you don't want to shade between them, because each is the tip of a bit of polygon that juts out, and between them is a "bay".

By drawing on bits of paper, you should be able to convince yourself that apart from this "tip of the horn" problem, the idea would work. For instance, on the shape shown in Figure 2.3 it shades correctly on rows A and B, but not on C or D which meet tips. Notice it works even though there are several closed curves drawn.

So now the problem is: how to recognize a tip?

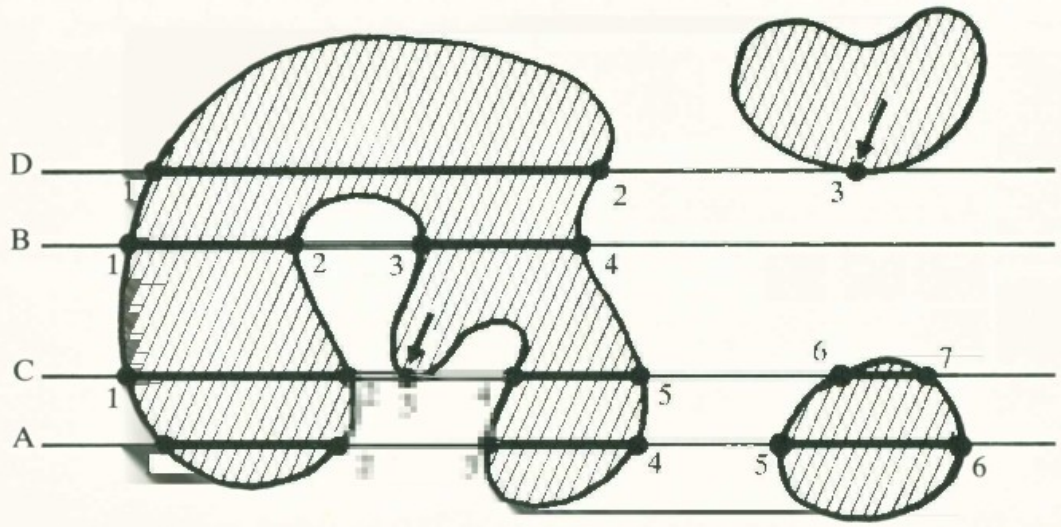


Figure 2.3 Shading between odd and even intersections will do the trick, except at the tip of a peninsula, as in lines C and D.

Obviously the characteristic feature of a tip is that the curve does not actually *cross* the horizontal line. It enters it from one side; possibly runs along it for a while; but then leaves on the same side it entered. Compare the two cases shown in Figure 2.4.

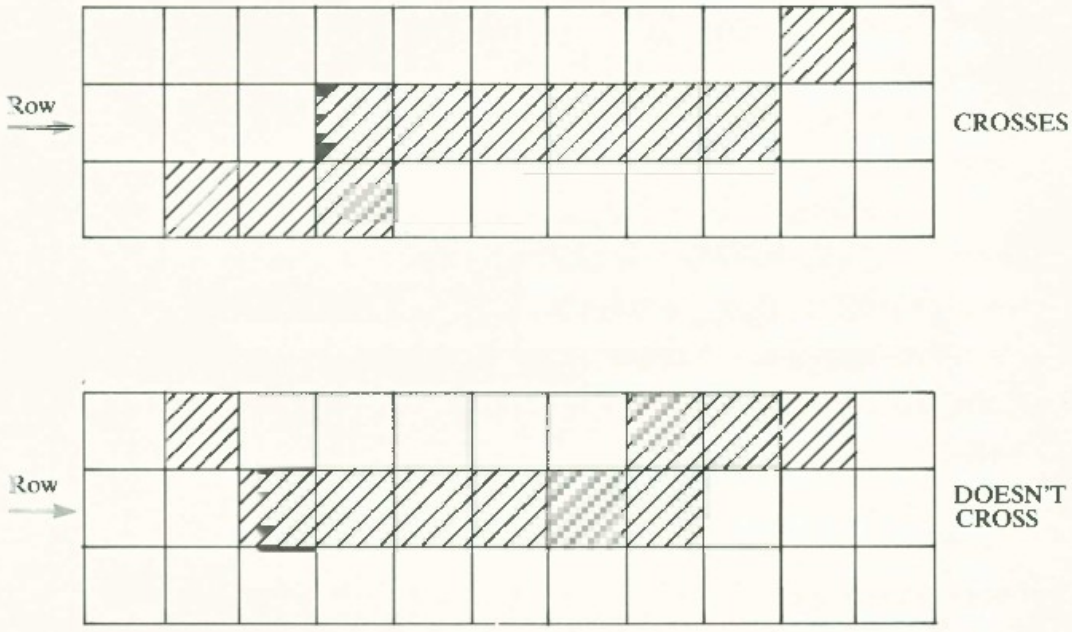


Figure 2.4 To locate tips of peninsulae, see if the line crosses the row or not.

So what we seem to need is a routine to see whether the curve crosses the row or not; and if it doesn't we ignore it. In order to see if it crosses, we also need to be able to recognize the part that runs *along* the row we're interested in. Then we look at the pixels surrounding the two ends, and see whether they are suitably filled in (see Figure 2.5).

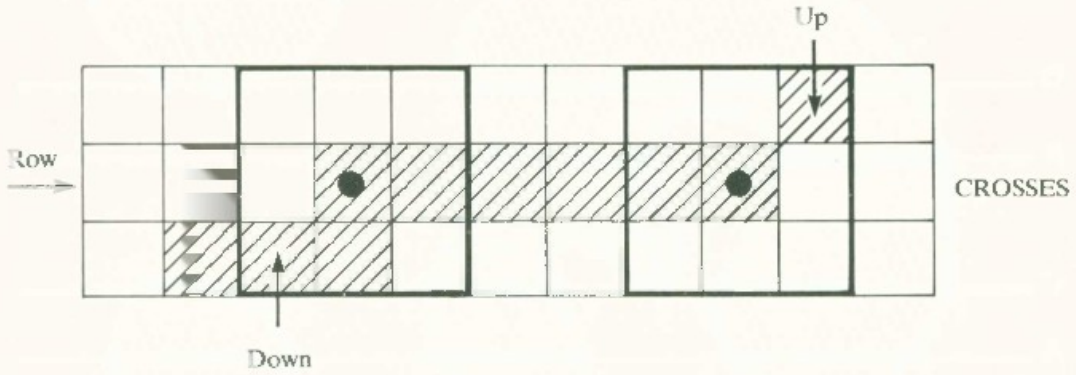


Figure 2.5 Detection of crossings by searching two 3 × 3 squares of pixels.

In fact we're still being a little naive; but we've got a workable idea, and it leads to the following program.

SUPERFILLER

```
10   LET track = 1000
20   LET test = 2000
30   LET list = 3000
40   LET shade = 4000
50   DIM a(20)
60   DIM b(20)
200  FOR y = 1 TO 174
205  LET q = 0
210  LET x = 1
220  IF x >= 255 THEN GO TO 400
230  IF POINT (x, y) = 0 THEN LET x = x + 1: GO TO 220
240  LET xl = x: GO SUB track
250  GO SUB test
260  LET x = xr + 1: GO TO 220
400  GO SUB shade
500  NEXT y
1000 REM track
1010 LET c = 0
1020 IF xl + c >= 255 THEN RETURN
1030 IF POINT (xl + c, y) = 0 THEN GO TO 1060
1040 LET c = c + 1: GO TO 1020
1060 LET xr = xl + c - 1
1070 RETURN
```

```
2000  REM test
2005  LET ll = 0: LET lu = 0: LET rl = 0: LET ru = 0
2010  FOR e = -1 TO 1
2020  IF POINT (xl + e, y - 1) = 1 THEN LET ll = 1
2030  IF POINT (xl + e, y + 1) = 1 THEN LET lu = 1
2040  IF POINT (xr + e, y - 1) = 1 THEN LET rl = 1
2050  IF POINT (xr + e, y + 1) = 1 THEN LET ru = 1
2060  NEXT e
2070  IF ll + rl = 0 OR lu + ru = 0 THEN RETURN
2080  GO SUB list
2090  RETURN
3000  REM list
3010  LET q = q + 1: LET a(q) = xr
3020  RETURN
4000  REM shade
4010  IF y < = 1 THEN RETURN
4020  FOR t = 1 TO 20 STEP 2
4030  IF b(t + 1) = 0 THEN GO TO 4100
4040  PLOT b(t), y - 1: DRAW b(t + 1) - b(t), 0
4050  NEXT t
4100  FOR t = 1 TO 20
4110  LET b(t) = a(t)
4120  NEXT t
4130  DIM a(20)
4140  RETURN
```

Before describing some of the peculiarities of this routine, I suggest you try it out. Key it in, and add:

```
1  CIRCLE 50, 50, 48: CIRCLE 50, 50, 44:
   CIRCLE 200, 40, 37: CIRCLE 205, 38, 20
```

to provide something to shade in. Now RUN. It's fairly slow, but it seems to work, as Figure 2.6 demonstrates.

Now for the explanations. Lines 200–500 set up the main program: a loop that checks along each horizontal row (except the top and bottom ones) looking for bits of curve. If it finds a bit it goes to a subroutine *track* which follows the curve along the line to find the ends (as marked in Figure 2.5); then it goes to *test* which decides, by examining the 3×3 region around each end, whether the curve crosses or not. If it does, the routine *list* notes down the relevant coordinate.

After a row has been scanned in this way, it is filled in by joining the 1st point to the 2nd, then the 3rd to the 4th, and so on as suggested above. However, there is one tricky feature. If you fill in the line too soon, it interferes with the *test* routine on the next line, and you get nonsense. So the list of points is first stored in a *buffer*—the array a—and then transferred to another array b on the next scan, ready to be plotted. Lines 4100–4130 perform this task.

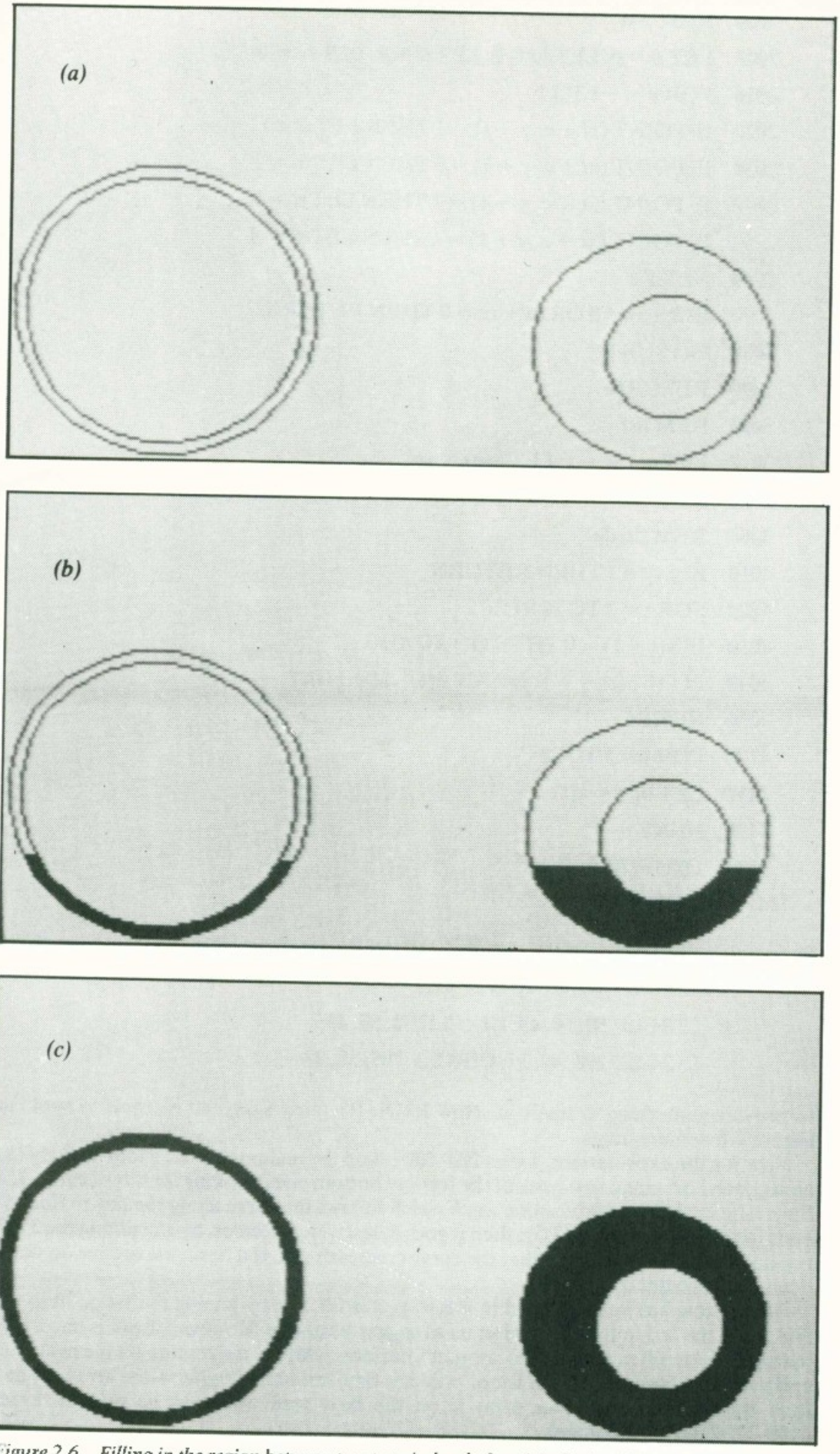


Figure 2.6 Filling in the region between two test circles: before (a), during (b), and after (c).

To keep the program simple, it is assumed that the curves being shaded in *do not touch the border* (rows 0 and 175, columns 0 and 255). So it scans rows 1 to 174 from columns 1 to 254 only.

THE MAP

So far so good, but will it pass the acid test? Will it shade the world map?

Load in your map, using

```
LOAD "map" SCREEN$
```

and then hit

```
GO TO 10
```

(*not* RUN, which wipes the map!).

Now, your map may not be quite the same as mine was. What I got was Figures 2.7 and 2.8.

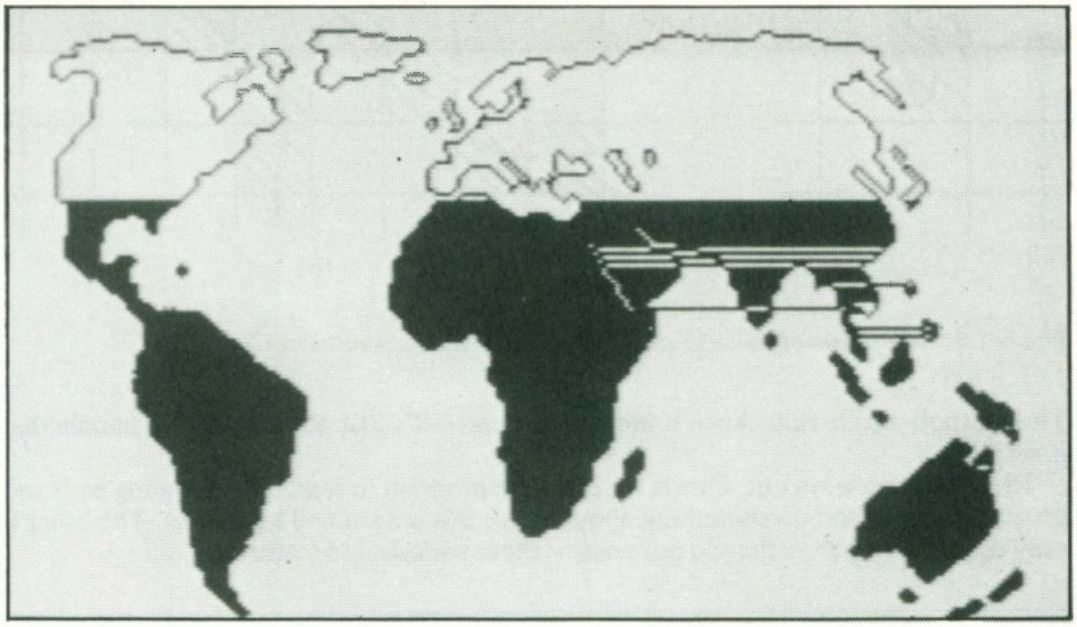


Figure 2.7 Shading in an outline map of the world—a few bugs . . .

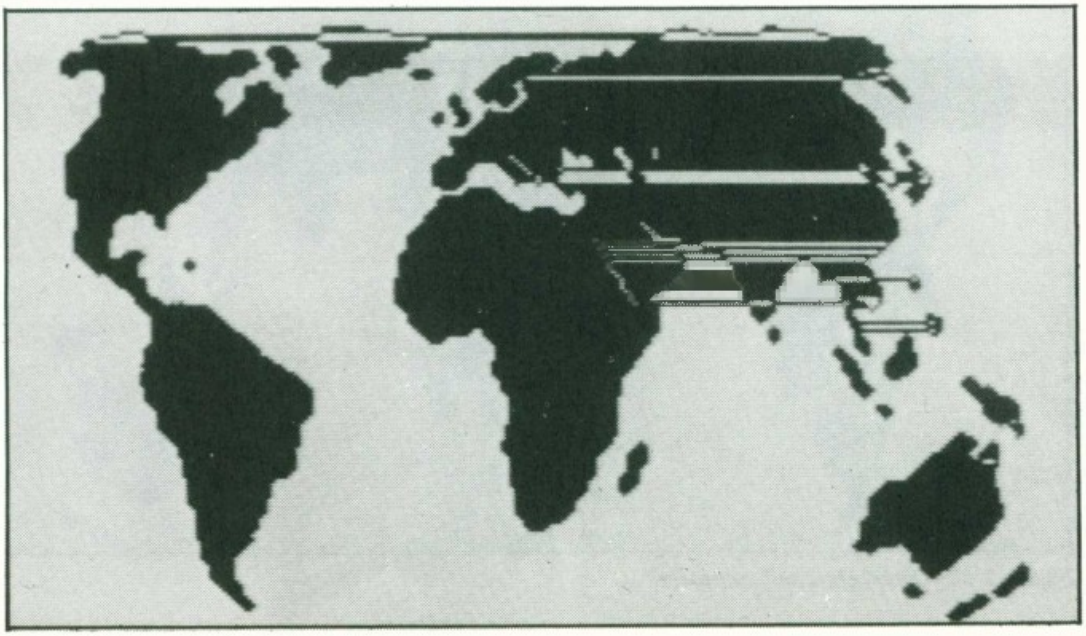


Figure 2.8 . . . and a few more!

So: if the routine fails on your map, as it did on mine, here's what you do. Use the sketchpad program to *modify* the map, removing dangling ends and touching curves. *Then* shade it in.

Possibly a few flaws will still remain: I don't guarantee that there aren't other undesirable features. (Breaks in curves cause havoc, but those are really just dangling ends again.) But you'll be able to spot where they arise, and use Sketchpad to eliminate them. After a few shots, you'll eventually end up with something like Figure 2.10. When you do, save it under a new name, say

```
SAVE "solidmap" SCREEN$
```

and it's ready to use in other programs.

If you think the shape is a little odd, it's because the map I started from was a Hammer equal-area projection, not the more usual Mercator projection.

THE PRAIRIE FIRE METHOD

There's a totally different approach to the shading problem that you might like to think about. The idea is to give the Spectrum a clue by "lighting a fire" on one of the dry-land regions. To do this, just tell it (via keyboard or a moving pixel) the coordinates of such a point. For example, the point 125, 80 is in the middle of Africa.

Now let the fire spread: that is, fill in all neighbouring pixels *unless you hit the coastline*. Using these as new starting points, spread the fire still further. It will only stop when it reaches the coast. In fact, the African fire will spread into Asia and Europe eventually. It will never reach the UK, the USA, or Australia; so you will have to input suitable "sparks" to start fires there too . . . in fact, you need one ignition point for each connected land-mass.

It's not a difficult method to program, but the above sketch has to be made a lot more precise. If you want an interesting project, this is a good one.

SET FIRE TO THE SEA

Since the sea is rather more connected than the land, a better way might be to set the *sea* alight (you'll have to light up the Caspian and Aral Seas separately), having set up the land colour as PAPER.

If the same expression keeps turning up,
with different values for its variables,
don't use a subroutine—try:

3 User-defined Functions

A function is a kind of "black box". You give it some numbers or strings, it gives you some back. For example, if you give the function LEN the string "gthily" it gives you back the number 6—the length of the string—because it works out

LEN "gthily" = 6.

The Spectrum has a lot of built-in functions like VAL, COS, TAN, EXP, and so forth. But sometimes you find you keep on using a particular expression, over and over again, on different variables. You *can* implement this as a subroutine; but that usually involves lots of LET a = 39: LET b = 21 . . . commands before you can call the routine.

The DEF FN key (E-mode/SYMBOL shift/1) lets you set up your own functions; and the FN key (E-mode/SYMBOL shift/2) calls them. Each must be followed by a single letter: DEF FNa, and FNa; or DEF FNb and FNb; and so on through the alphabet. If its value is a string, then you have to use FNa$, FNb$, etc. You can *use* capitals too; but the Spectrum takes no notice. That is, FNa and FNA are considered to be identical. So you've got 26 functions of each type at your disposal.

For example, suppose we find that our program keeps wanting to add three numbers together and then divide by three, getting an average. We have lots of expressions like (x + y + z) / 3 and (price1 + price2 + price3) / 3 scattered around the listing. Then we set up a function like this:

```
10  DEF FNa (p, q, r) = (p + q + r) / 3
```

Once this is done, we can replace the above expressions by:

```
FNa (x, y, z)
FNa (price1, price2, price3)
```

wherever they occur.

Let's test it out:

```
10  DEF FNa(p, q, r) = (p + q + r) / 3
20  INPUT x, y, z
30  PRINT FNa (x, y, z)
```

RUN this, and input things like:

```
 1     2     3     (result 2)
 4     4     4     (result 4)
77    91     3     (result 57)
```

to make sure it's working OK.

Various points are worth noting. First, the variables p, q, r that occur in the function definition are simply place-holders: you can use those letters elsewhere in the program without any harm being done, and you can define the *same* function using other letters, for example:

```
10  DEF FNa (a, b, c) = (a + b + c) / 3
```

It doesn't even matter if the letter used to name the function (here "a") occurs as a variable inside the brackets that define it. The Spectrum can tell which is which.

Next, the letters inside the brackets must be single: you can't use variables like *price1* in the *definition*. There is no harm at all, though, in using price1 when the function is calculated somewhere: FNa (price1, price2, price3) is fine.

You can have string variables in the definition, but they also must be only one letter, followed by $. You can mix numbers and strings; and the computer is quite happy with, say

```
10  DEF FNb (b, b$) = b * LEN b$
```

even though the "b" occurs in three places with three different meanings. Then FNb multiplies the length of a string b$ by the number b. If you ask for

```
FNb (7, "cat")
```

you'll get the answer 21, which is found by working out

```
7 * LEN "cat" = 7 * 3 = 21.
```

The definition of a function does *not* have to come in the program before the function is used; but it must be in there somewhere (rather like DATA definitions).

As another example, try the string-valued function

```
10  DEF FNm$ (u$, v$) = u$ + u$ + v$
```

and find out what you get if you ask for

```
FNm$ ("B", "C")
FNm$ ("a", "gh!")
FNm$ ("co", "nut")
FNm$ ("Zsa□", "Gabor")
```

Every function-definition *must* include the brackets. But it *can* have no variables! If you write

```
10  DEF FNk ( ) = 77
```

then FNk will give you the number 77 whenever you call it. (There are more subtle occasions where this kind of thing can actually be useful: here it just looks perverse.)

Further, the function-definition can include variables that are not enclosed in the brackets, provided these are assigned in the program. So

```
10  DEF FNa (x) = x + q
20  LET q = 5
30  PRINT FNa (10)
```

gives the result 15; but

```
10  DEF FNa (x) = x + q
20  PRINT FNa (10)
30  LET q = 5
```

gives an error message. (WARNING: press CLEAR before you check this out: the value of q will still be in the machine from the first trial.) To check that the definition

really can go anywhere, try

```
10  LET q = 5
20  PRINT FNa (10)
30  DEF FNa (x) = x + q
```

SOME USES

It's only worth defining a function this way if (1) you keep using the same expression over and over again but with different variables, or (2) your program manipulates a function which is not always the same one—such as a graphics program to plot out the graph of a given function. Then you may prefer to write it to manipulate, say, FNa; and then have the user edit in the desired function value. (Or sneaky tricks with VAL, and it can be INPUT—at a price: a slower program. See Chapter 16.)

For example, the distance between points (a, b) and (c, d) on the screen (in pixel-sized units) is given by:

```
10  DEF FNd (a, b, c, d) = SQR ( (a − c) * (a − c) + (b − d) * (b − d) )
```

which is the Spectrum's version of Pythagoras' Theorem. If you have a lot of distances to deal with, this may be useful. (Not just in maths: you may have set up a map of the USA and want to know how far it is from Los Angeles to Oklahoma City.)

Bear in mind the possibility of using *logic values*. Recall that a logical statement like "x < > 4" is considered by the computer to have a *numerical* value: 1 for true, 0 for false. So

```
10  DEF FNo (u, v) = u > = 0 AND u < = 255 AND v > = 0 AND v < = 175
```

may look to you like nonsense; but to the Spectrum it's clarity itself. In fact, FNo tests whether a pixel position (u, v) is on screen or not:

FNo (u, v) = 1 if (u, v) is on screen,
FNo (u, v) = 0 if (u, v) is off screen.

So, if you keep needing to test for this, it's a function worth thinking about.

Or, take the current rates for newspapers and periodicals by Air Mail outside Europe. These depend on the weight, and zone (A, B or C) as follows:

	Zone A	Zone B	Zone C
First 10 g	24	26	29
Each additional 10 g or part thereof	11	14	15

Let's set up a user-defined function FNp to give us the price for any weight w grammes and any zone z$ (= "a", "b", or "c"). It will take a numerical value, so we don't have to call it FNp$. And it will look like:

```
DEF FNp (w, z$) = something nasty . . .
```

Let's take it stage by stage. First, thinking just of zone A. For simplicity, assume that the weight is always greater than 0. Then the "first 10 grammes" always applies, so we'll certainly need to pay our 24p. The weight left over is w − 10. If this is 0 or less, then we're done; but if not we must round it up to the *next* 10 grammes.

To round up a number n to the next multiple of 10, we can use the expression

```
−10 * INT (−n/10)
```

Try it: if n = 43 then we have:

```
−n = −43
```

$-n/10 = -4.3$

INT $(-n/10) = -5$ (yes, try it! INT rounds *down*)

$10 * \text{INT}(-n/10) = -50$

$-10 * \text{INT}(-n/10) - 50$

which is what we want.

We'll need to use this process several times, so let's *define a function*:

DEF FNr (n) = $-$INT $(-n/10)$

which is the "round-up" function divided by 10. Great!

Now, in zone A, the price we pay is

$24 + 11 * \text{FNr}(w - 10)$

provided $w > 10$, and only 24 if $w <= 10$. Hmmm . . . logic values! We have to pay

$24 + (w > 10) * 11 * \text{FNr}(w - 10)$

because $(w > 10)$ takes value 1 when $w > 10$, adding on the extra bit; but value 0 when $w <= 10$, leaving just the 24.

Zones B and C give similar expressions, but with different values in place of the 24 and 11. How do we work them in?

If we use lower-case letters "a", "b", "c" for the zone variable z$, then logic values come to our aid again. The number

$24 * (z\$ = \text{"a"}) + 26 * (z\$ = \text{"b"}) - 29 * (z\$ = \text{"c"})$

takes value 24 when z$ = "a", 26 when z$ = "b", and 29 when z$ = "c". (Why?) And we can deal with the 11 – 14 – 15 bit the same way.

All of which leads us to the definitions:

```
10  DEF FNr (n) = -INT (-n/10)
20  DEF FNp (w, z$) = 24 * (z$ = "a") + 26 * (z$ = "b") + 29 * (z$ = "c")
      + (11 * (z$ = "a") + 14 * (z$ = "b") + 15 * (z$ = "c") ) *
      (w > 10) * FNr (w - 10)
```

and now FNp (w, z$) does give the price of a newspaper, weight w, to zone z$.

Projects

1. Set up FNt so that FNt (x) is the cost of x cans of beer at 65 cents apiece.
2. Set up FNu so that FNu (x, p) is the cost of x cans of beer at p cents apiece.
3. Set up FNj$ so that FNj$ (a$, b$, c$) gives whichever of a$, b$, or c$ comes earliest in alphabetical order. (Note: two strings a$ and b$ are in alphabetical order if a$ <= b$, in the Spectrum's notation for ordering strings.)
4. For newspapers registered at the post office, the prices table above becomes:

	Zone A	Zone B	Zone C
First 10 g	13	15	16
Each additional 10 g or part thereof	3	4	5

Define FNq (w, z$) to give the price of a registered newspaper of weight w to zone z$.

5. Combine FNq and FNp (in the text) to give a function FNr (w, z$, y) where w is weight, z$ zone, and y = 0 for unregistered papers, 1 for registered.

. . . then there was the programmer who always used magenta ink because he liked his programs to remain inviolate . . .

4 Control Characters

You've no doubt discovered that the top row of keys on your Spectrum behave differently from the rest, as regards modes and suchlike. If you haven't, try the following experiment. Hit, in turn,

NEW
CAPS SHIFT and SYMBOL SHIFT [for extended mode]
Key 4 in the top row
"fingers"

You'll find you have green fingers . . .

By using the top row of keys *in extended mode* (with or without CAPS SHIFT) you can, from the keyboard, set colours, FLASH, BRIGHT, change the PRINT position, and so on. The *Manual* gives full details of the effects of particular combinations of keys and modes, on page 115; the important part for us is:

Key	Effect in extended mode	
	Without CAPS SHIFT	With CAPS SHIFT
1	PAPER blue	INK blue
2	PAPER red	INK red
3	PAPER magenta	INK magenta
4	PAPER green	INK green
5	PAPER cyan	INK cyan
6	PAPER yellow	INK yellow
7	PAPER white	INK white
8	BRIGHT off	FLASH off
9	BRIGHT on	FLASH on
0	PAPER black	INK black

EFFECT ON LISTINGS

For test purposes, input a few lines of program:

10 REM
20 REM
30 REM

Now hit, in turn,

(a) 1
(b) REM

(c) extended mode/CAPS SHIFT and 9
(d) ENTER

Watch the cursor carefully to make sure you actually get into extended mode. You'll find that the *whole* program, except for line 1, flashes at you. LIST it: it still flashes.

Repeat this, but use different keys in the top row, and sometimes leave out the CAPS SHIFT, in (c). Hmmm . . . pity it affects the *whole* listing . . . or does it?

Go back to the flash version of line 1; and add

```
11 REM [extended mode/4]
```

Now everything after line 11 has gone green (colour 4). But it's still flashing . . . Wasn't there a FLASH off in the table? So maybe we need:

```
11 REM [extended mode/4] [extended mode/CAPS SHIFT and 8]
```

to get rid of the flash but keep the green, from line 20 onwards.

Try it and see.

CONTROL CHARACTERS

What *is* going on?

If you look at the list of character codes in the *Manual*, page 183, you'll find a bunch at the front (numbers 6 to 23) that cannot be printed out (even as ? marks). These are *control characters* which affect the behaviour of the system. Here's the list:

Code	Character
6	PRINT comma
7	EDIT
8	cursor left
9	cursor right
10	cursor down
11	cursor up
12	DELETE
13	ENTER
14	number (used in program organization)
15	(not used)
16	INK control
17	PAPER control
18	FLASH control
19	BRIGHT control
20	INVERSE control
21	OVER control
22	AT control
23	TAB control

These live in the memory like any other character, but they won't PRINT or LIST When you use the top row of keys in extended mode, you input certain of these characters. For example, key 4 in CAPS SHIFT has the effect of the INK control character, with the colour green.

Although you won't *see* a control character in a listing, you will see its *effect*. And you can check it's really in memory, either by PEEKing the relevant addresses (*Easy Programming*, page 93) or by the following experiment. Hit, in turn,

(a) 1
(b) REM

(c) E-mode/1

(d) E-mode/2

(e) E-mode/3

(f) E-mode/4

(g) E-mode/5

(h) E-mode/6

where E-mode refers to extended mode. Note that, unlike graphics mode, you have to go into E-mode each time.

You'll see that the cursor doesn't move, after the REM: it just keeps changing colour. Now use DELETE, held down for auto-repeat: see how long it takes to work its way past all those control characters? Try again, pressing DELETE repeatedly in single steps: watch the changes in the display. Obviously there are a lot of characters in memory that aren't getting PRINTed.

FLASHY LISTINGS

You can actually *use* this facility: it's not just a pretty trick. For example, you can make REM statements stand out in a listing, for easy visibility:

```
1  REM [E-mode/CAPS SHIFT/9] This will stand out [E-mode/CAPS/8]
10 REM
20 REM
etc.
```

LIST this program, and check that the REM statement continues to flash. Now SAVE it on tape, NEW, LOAD back . . . yes, it's still flashing.

Similarly you can colour-code sections of program, for example subroutines. Put the control characters into the start of the first line of the routine (after the line number—or else at the *end* of the previous line: anything before a line number is ignored). All subsequent lines will be LISTed in that colour.

To make a listing invisible, set its INK and PAPER colours to the same thing. (But it still LLISTs correctly; or it will list from a line after the one with the control character, so you can't protect against pirates this way. *Nothing* gives completely foolproof pirate-protection, but there are tricks, of which this is the simplest, to discourage amateurs.)

USE IN PROGRAMS

You can make use of control characters in programs, to avoid having to set INK, PAPER, etc. all over the place. This is especially useful in creating colourful displays, title pages for programs, and suchlike.

For example, to print out the French tricolour at any position r (row) and c (column), use this:

```
10 PAPER 0: INK 7: BORDER 0: CLS
20 INPUT r, c
30 FOR i = 0 TO 2
40 PRINT AT r + i, c; "(E-mode/1) □ □ (E-mode/7) □ □
   (E-mode/2) □ □"
50 NEXT i
```

where the boxes are SPACE characters.

To get the Italian flag, change the number 1 in line 40 to 4.

You won't have failed to notice that the string in line 40 is LISTED in its blue-white-red glory.

Which leads us to something a tiny bit more ambitious: Old Glory as a single string. You can PRINT an approximation to the American flag using control characters, by using Figure 4.1 and the graphics mode. It takes time and care; but at the end you'll understand the control characters from keyboard pretty well!

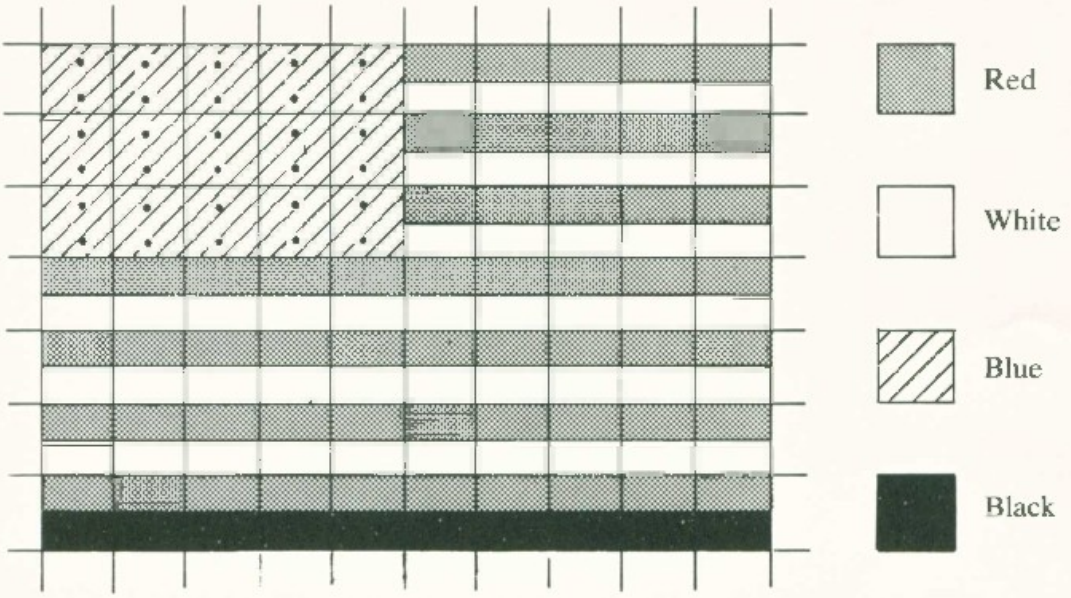


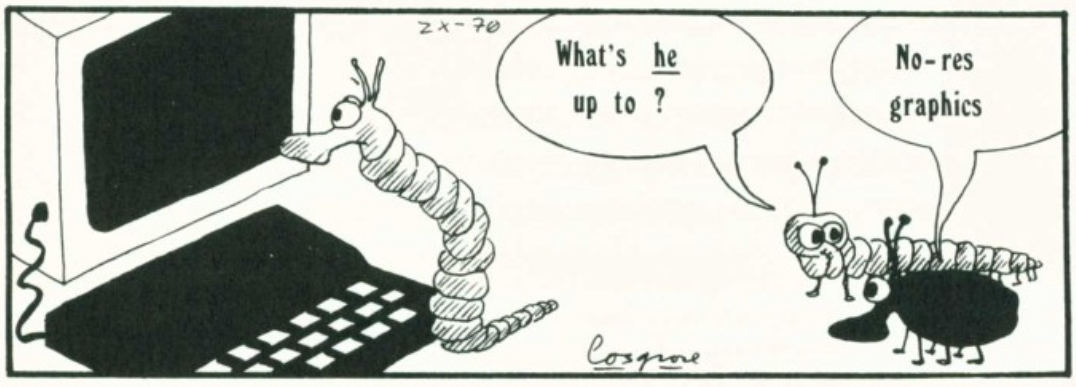
Figure 4.1 Old Glory entered from the keyboard as a single string.

Here's a blow-by-blow account: you'll soon see the flag building up as you work, and you'll be able to anticipate what's needed next. With practice, you can do this kind of thing from scratch without any initial sketches. (g3c is key 3 in graphics mode with CAPS SHIFT.)

```
10  PRINT "(E-mode/1) (E-mode/CAPS/7) : : : : :
    (E-mode/2) g3c g3c g3c g3c g3c (E-mode/0)
    (22 spaces) (E-mode/1) (E-mode/CAPS/7) : : : : :
    (E-mode/2) g3c g3c g3c g3c g3c (E-mode/0)
    (22 spaces) (E-mode/1) (E-mode/CAPS/7) : : : : :
    (E-mode/2) g3c g3c g3c g3c g3c (E-mode/0)
    (22 spaces) (E-mode/2)
    (g3c ten times) (E-mode/0)
    (22 spaces) (E-mode/2) (g3c ten times)
    (E-mode/0) (22 spaces) (E-mode/2) (g3c ten times)
    (E-mode/0) (22 spaces) (E-mode/2) (E-mode/CAPS/0)
    (g3c ten times) (E-mode/CAPS/7) (E-mode/0)"
```

Phew! Now, set the PAPER and BORDER to black, INK to white, and RUN. The stars part could be improved (think about user-defined graphics), but it's clear what it's meant to be. And all in one string . . .

For fast, colourful, lo-res graphics, you can use this technique to turn an entire screenful of coloured graphics characters into a single string, and PRINT it almost instantly. So any picture you can design on a 64 × 44 grid can be input, from the keyboard, as a string with 7Ø4 characters. It takes time, and patience; but it's worth it for an attractive display, and it's an efficient use of memory. (In practice, for ease of keying in and editing, I suggest 5 or 6 strings of 128 or 16Ø characters.)



"It's not what you do, it's the way that you do it." The same data, displayed in different ways, can be crystal clear or clear as mud. Graphics and colours add comprehensibility. But have you ever thought of using a formula to play a tune?

5 Display Techniques

Computing is not just a matter of generating vast swathes of numerical print-out, even if this does impress visiting bureaucrats. It's important to find suitable ways of presenting data too. I explore several of these at various points in this book. Here I'll take a look at five possible ways of presenting a series of numbers produced by a rather interesting mathematical process. The process itself is discussed at the end, because I know a lot of people are less entranced by mathematics than I am . . .

The program asks you to choose the type of display required from a menu of five options: then you must input a number between 0 and 1000. The listing is so simple that I'll give it all in one go:

```
10  DIM a(5)
20  LET a(1) = 40: LET a(2) = 255:
    LET a(3) = 704: LET a(4) = 1000:
    LET a(5) = 704
200 PRINT "Choose type of display:
    1. Numeric
    2. Graphic
    3. Colour
    4. Sound
    5. Both"
210 INPUT d
300 PRINT ' "Choose a number between 0 and □□□□ 1000"
310 INPUT k: CLS
320 LET k = k/250: LET x = .7
350 FOR t = 1 TO a(d)
360 LET x = k * x * (1 − x)
370 GO SUB 1000 * d
380 NEXT t
390 STOP
```

```
1000  REM numeric
1010  PRINT x,
1020  RETURN
2000  REM graphic
2010  IF t = 1 THEN PRINT k * 250
2020  PLOT t, 0: DRAW 0, 170 * x
2030  RETURN
3000  REM colour
3010  PRINT PAPER INT (8 * x); "□";
3020  RETURN
4000  REM sound
4010  BEEP .05, 20 - 40 * x
4020  RETURN
5000  REM both
5010  GO SUB 3000: GO SUB 4000
5020  RETURN
```

Before going on you might like to try this out. Any number in the 0–1000 range is allowed; but numbers bigger than 750 produce more interesting results. Option 1 prints out rather meaningless lists of numbers; option 2 produces some quite pretty spiky things; option 3 draws coloured bars and blocks all over the screen; and option 4 plays quite striking tunes, sometimes rhythmic and repetitive, sometimes more complex. Option 5 combines 3 and 4.

A SYSTEMATIC APPROACH

Let's take a more systematic look, by picking a value of the number to be input, and comparing the types of display. In fact we'll take four standardized values,

766 880 897 985

which between them illustrate the crucial points.

RUN the program with option 1, and input 766. You'll get a table of numbers in two columns. Reading along the rows in turn these give successive values of a number calculated by line 360 of the program. It's not easy to see anything significant (which is the trouble with numerical, tabulated output); but the trained eye will notice that in each column the numbers become more and more alike as you read down the screen. The left-hand one is close to 0.58, the right-hand one to 0.75. So the values are alternately flipping from one value (or near it) to the other. The sequence of values is (tending towards) something that is *periodic*, that is, repeats the same values over and over again; and the *period* is 2.

Repeat, with option 1, but input the next test number, 880. Now the result is different: the numbers don't settle down at all. However, each column is trying to alternate between two values: 0.82 and 0.87 on the left; 0.51 and 0.37 on the right. The whole sequence repeats every *four* goes, so it's periodic of period 4.

Now try option 1 with 897. Hmmm, well . . . is there a pattern or not? There are quite a few 0.89s on the left; and several things near 0.33 on the right; but it's not very clear.

Not to worry; option 1 with 985 is even worse. In fact, it looks a total mess.

Option 2 makes life much easier. For the number 766 it gives Figure 5.1, and the period-2 behaviour is very clear from the way the spikes go up-down, up-down. For 880 the period 4 is also very apparent (Figure 5.2).

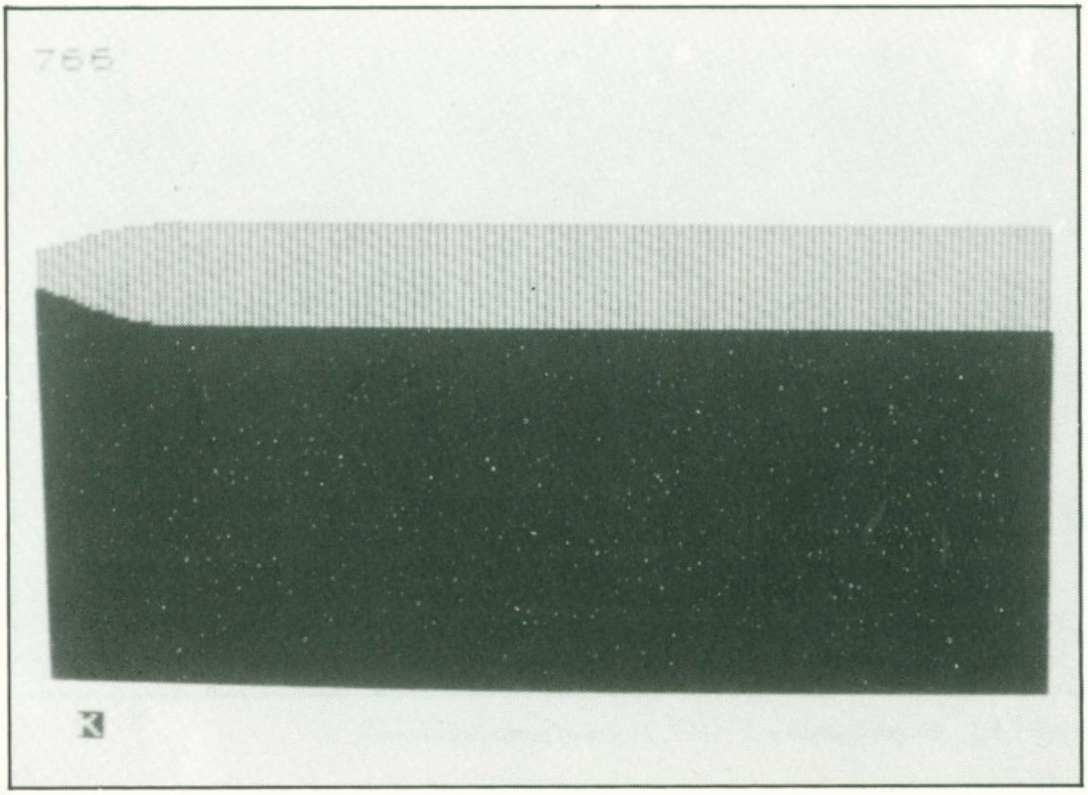


Figure 5.1 Graphic display: k = 766. Period 2.

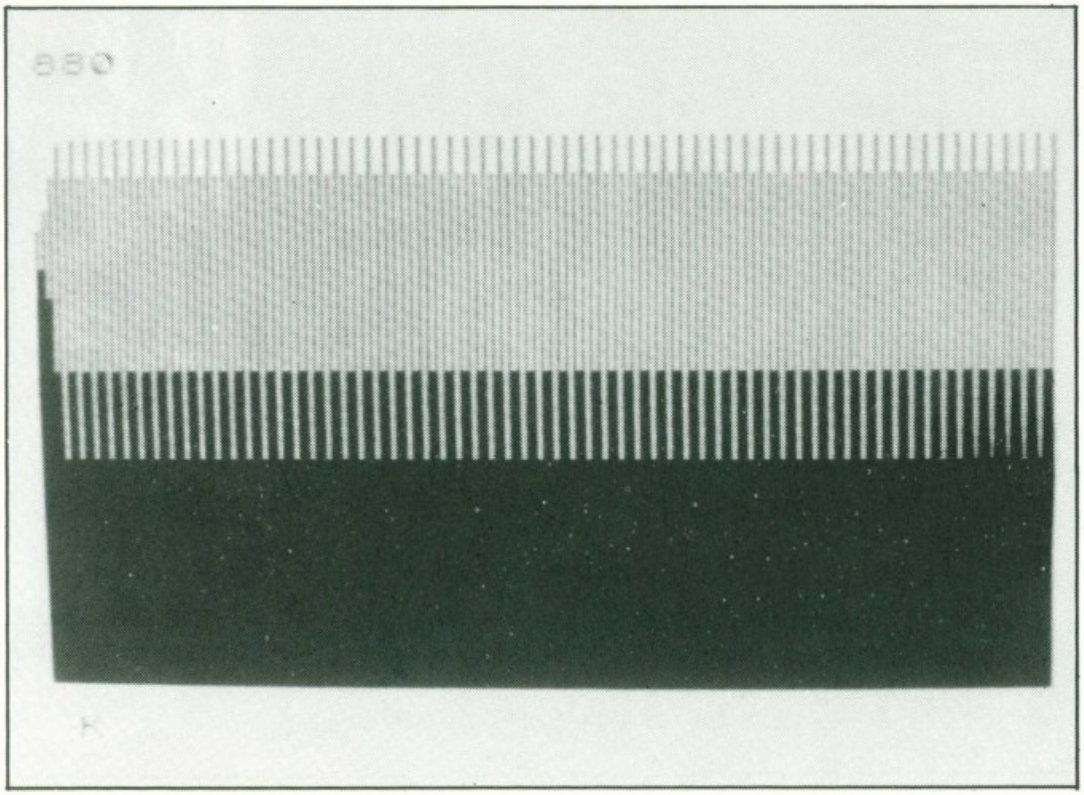


Figure 5.2 Graphic display: k = 880. Period 4.

For 897 there are definite traces of periodicity, but it's a little irregular (Figure 5.3).

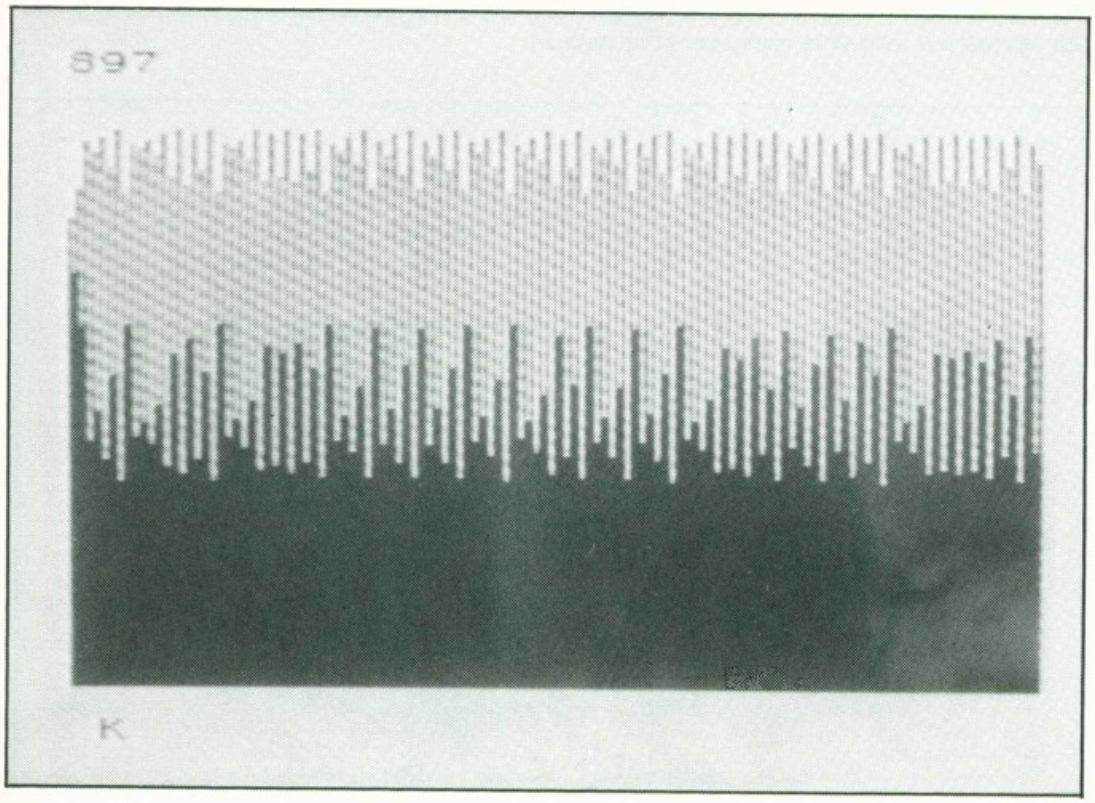


Figure 5.3 Graphic display: k = 897. Traces of periodicity remain.

For 985 the spikes are pretty much random (Figure 5.4).

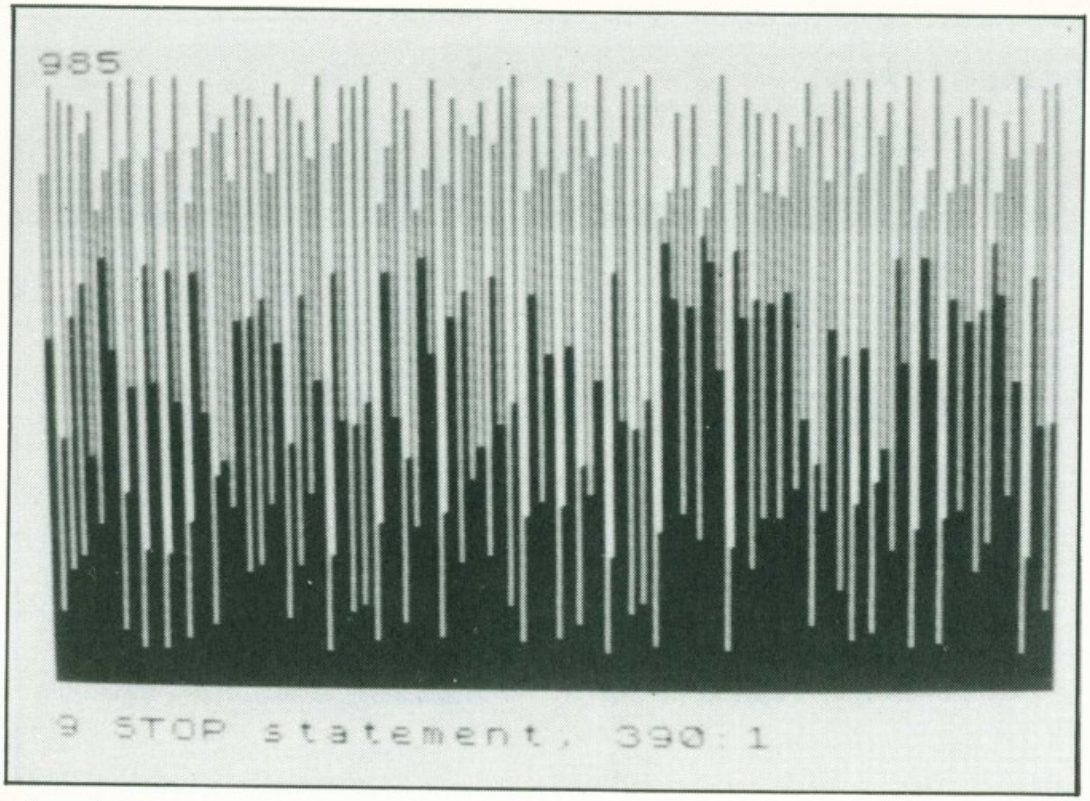


Figure 5.4 Graphic display: k = 985. Chaos!

The colour plot (option 3) brings out the periodicities even more strikingly—partly because the periods 2, 4 etc. all divide the number of characters in a row (32), a fortunate

coincidence. For 766 you will see vertical green and cyan stripes (except near the very beginning), period 2. For 880 (Figure 5.5) the stripes repeat the 4-fold pattern red-yellow-green-white. For 897 (Figure 5.6) there is a definite striped effect, with colours chosen from yellow-magenta-white-green-red; but there are occasional lapses where a square gets a different colour from the stripe. And for 985 (Figure 5.7), you just get random-looking squares.

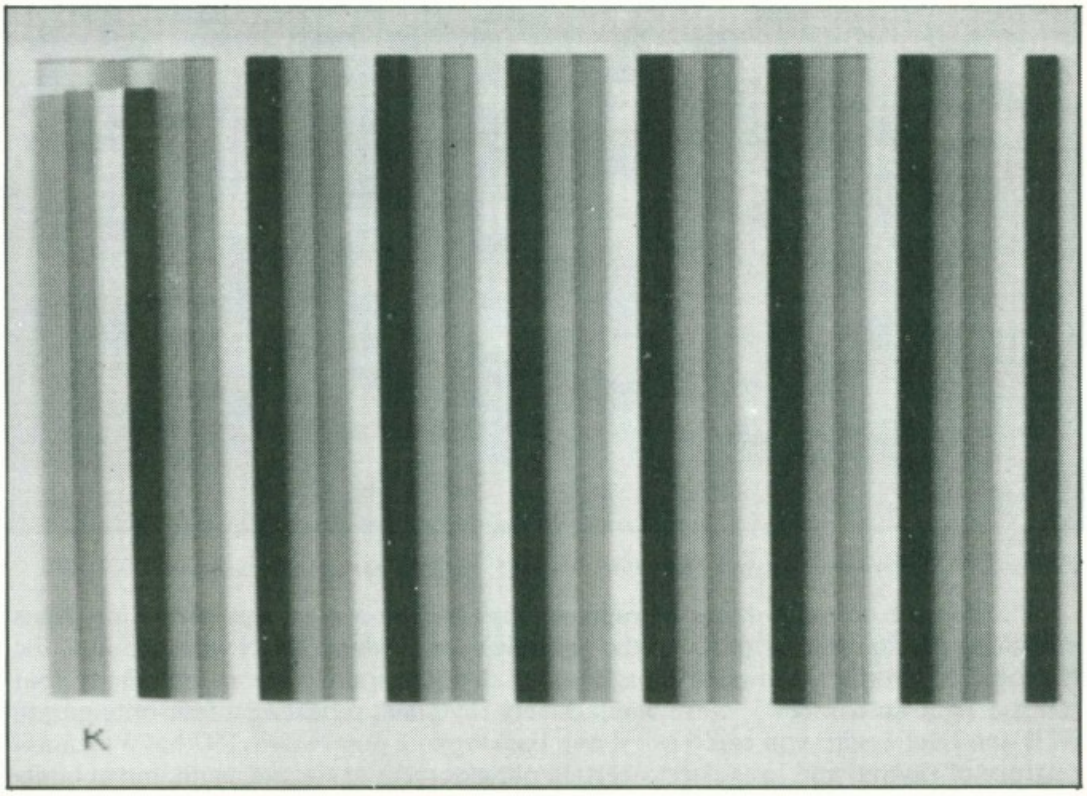


Figure 5.5 Colour plot (here in black and white). Pattern of bars shows periodicity when k = 880.

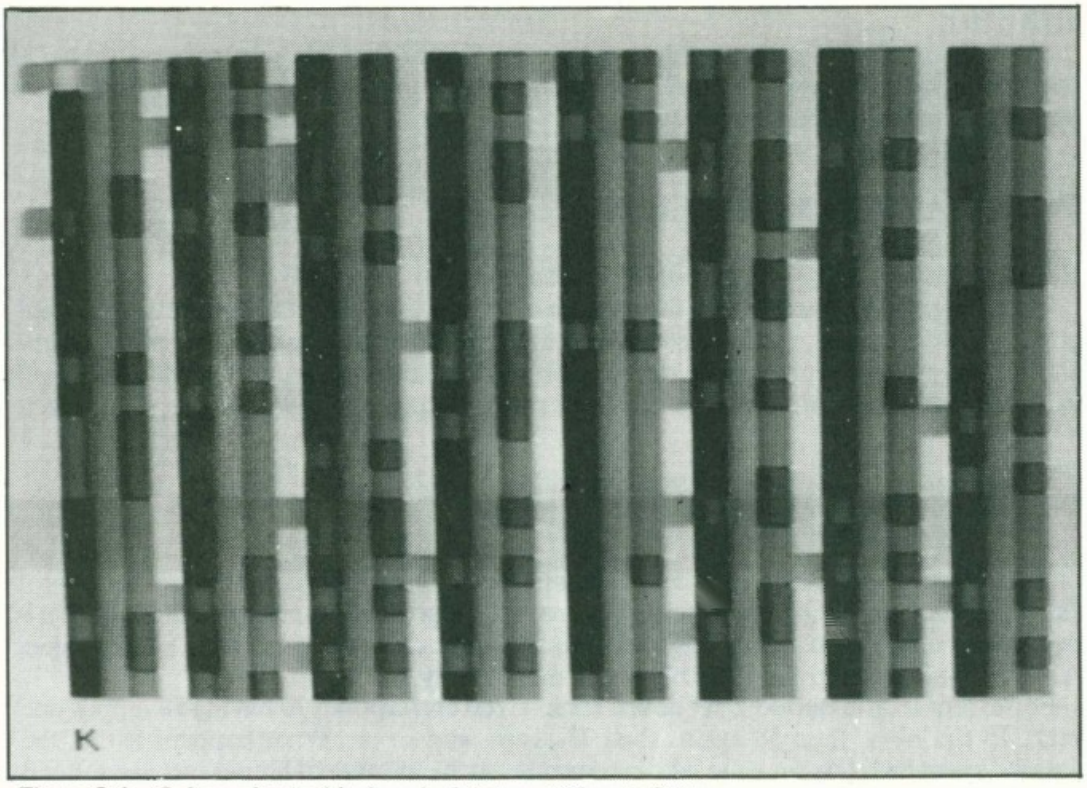


Figure 5.6 Colour plot (in black and white): partial periodicity with occasional lapses when k = 897.

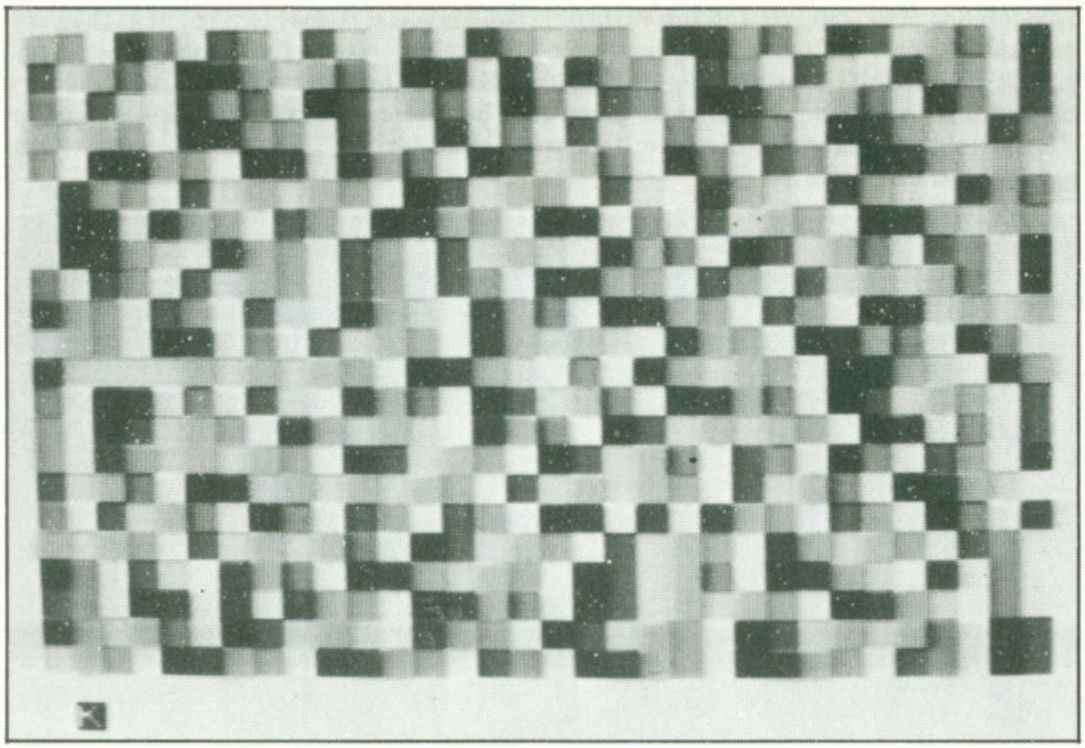


Figure 5.7 Colour plot (in black and white): k = 985, random pattern of squares showing chaos.

One doesn't normally think of representing data by a *tune*; but here it produces intriguing results: the ear picks up the periodicity as *rhythms*. RUN with option 4, the number 766 produces a monotonous doo/dah doo/dah sound, like an ambulance (but not the right notes). 88Ø is more compulsively rhythmic, repeating a four-note phrase over and over again: you could use it as a backing to a pop record. 897 has a pleasant mixture of rhythm and irregularity: certain phrases recur again and again, but at oddly spaced intervals. 985 sounds like bad Schönberg.

Option 5 lets you see the colours and hear the sound together: they tend to reinforce each other.

Try other values for the number, now. You'll find that for numbers less than 75Ø everything settles to a single value (very dull) and that the behaviour gets more and more peculiar, the larger the number gets.

POPULATION MODELLING

Very fascinating, no doubt: but what's it all about?

The program is based on a formula that is used in theoretical models of animal populations. Imagine a lake large enough to support 1ØØ hippopotami (at most). Depending on the reproductive rate per generation, how does the number of hippos vary? With this particular program, the number x that is output or plotted (or beeped) is given by

$$\frac{\text{number of hippos in the current generation}}{\text{maximum number possible } (= 1ØØ)}$$

and the reproductive rate is 1/25Ø of the number k that you input. If this is less than 1 (k < 25Ø) the number of hippos tends to zero: the hippo population slowly dies out. If it is between 1 and 3 (k between 25Ø and 75Ø) the number settles to a single steady value. Above that, it starts to oscillate more and more wildly.

For example, the period-2 sequence for k = 766 corresponds to having 58 hippos one year, 75 the next, then 58 again, then 75 again, and so on. What happens is that the population at 58 is less than the lake can sustain, so the number of hippos increases; but it *overshoots* the happy medium to give 78, which is too many. So the next year it drops—too far again—to 58; then repeats.

The random behaviour seen at k = 985 (and partly at 897) is very interesting mathematically, because we know it isn't *really* random at all. It's produced by the very simple formula on line 360 of the program. But it certainly *looks* random. It's called *deterministic chaos*, and it's a two-edged sword. On the one hand, it shows that apparently random events can have a simple underlying structure; on the other, it casts doubt on the ability of apparently well-behaved theories to make useful predictions. This isn't the place to teach you about chaos; but if you want to know more, I can recommend pages 307–318 of *Concepts of Modern Mathematics* by Ian Stewart (no, it's *not* Shiva actually: it's Penguin Books).

PEEK lets you see what information an address holds, and POKE lets you change it. The question is, where to PEEK and what to POKE it with!

6 System Variables

Spectrum memory, as you are no doubt well aware, comes in two kinds: Read Only Memory (ROM) and Random Access Memory (RAM). Only in RAM can you change the contents of a memory location (*address*). Most of the machine's operating system lives permanently in ROM; but a certain amount is set up in RAM so that it can be changed as necessary. This is the *system variable* area of RAM, and it runs from address 23552 to 23733. The *Manual* gives a comprehensive (but not always comprehensible) list on pages 173–176.

Most of these variables are not especially useful, as far as incorporating them into programs is concerned; but some are. Although they are given names in the *Manual*, these are for reference purposes only and are not directly accessible from BASIC.

To find out the value of a systems variable, you use PEEK: to change it, you use POKE (see *Easy Programming*, page 93). For more detail, see below.

The aim of this chapter is to describe those system variables that you are likely to find useful, and to give you a few ideas for using them. The important thing is to realize that the system variables are *there*, and that you are free to bend them to your will should the occasion arise.

KEYBOARD RESPONSE

Table 6.1

Mnemonic	Address	Standard Value
REPDEL	23561	35
REPPER	23562	5
PIP	23609	0

These control the waiting time for a key to auto-repeat; the speed with which it repeats; and the length of keyboard BLEEP on entering a character. For a faster-responding keyboard with audible BLEEPs, enter (in command mode or from a program):

```
POKE 23561, 10
POKE 23562, 1
POKE 23609, 50
```

You can vary the numbers to suit your tastes: sensible ranges seem to be (respectively) 10–20; 1–3; 40–100.

To avoid having to type these in every time you switch on, you can make a tape with them on, and LOAD it first. This of course takes even more time than typing them out—but if you also include on the tape your favourite utilities (Line-renumbering, see Chapter 12; Attribute-changing, see Chapter 7; some common user-defined graphics) it can make a helpful package to have sitting up in the BASIC area.

ORGANIZATION OF MEMORY

The computer's memory is divided up into blocks that do different jobs. The boundaries between these blocks can wander around: the addresses of the boundaries are held in the systems variables as given in Table 6.2.

Table 6.2

Mnemonic	Addresses	Value after NEW (16K machine)
(16384: start of display file)		16384
(22528: start of attributes file)		22528
(23296: start of printer buffer)		23296
(23552: system variables area)		23552
(23734: used for microdrive)		23734
CHANS	23631–2	
PROG	23635–6	
VARS	23627–8	
E-LINE	23641–2	
WORKSP	23649–50	
STKBOT	23651–2	
STKEND	23653–4	
RAMTOP	23730–1	32599
UDG	23675–6	32600
P-RAMT	23732–3	32767

All of these are 2-byte variables. This means that to find, e.g. where the program is, you must use

PRINT 23635 + 256 * PEEK 23636

and in general (first byte)+256*(second byte). Conversely, to change PROG to, say, 13244 (I don't recommend this! It's just chosen to give the general idea) you have to work out what 13244 is in the form (byte) + 256 * (byte). In fact the second (or *senior*) byte is given by

INT (13244/256), which is 51

and the first (*junior*) byte by

13244 − 256 * INT (13244/256), which is 188.

So the command would be

POKE 23635, 188: POKE 23636, 51

Similarly to change RAMTOP to the value n (see below for reasons for wanting to do this) you use

POKE 23730, n − 256 * INT (n/256): POKE 23731, INT (n/256)

Only RAMTOP and UDG can usefully be POKEd, among this set of system variables; but each can be PEEKed to find out whereabouts each section of memory lies.

CHANS is the area for the microdrive communications system, and you won't find much use for that. PROG is the start of the BASIC program area: you need to know it for (e.g.) the line-renumbering routine in Chapter 12. VARS is the place where the

variables are stored: if you want a fancy line-renumbering routine you might find a use for it. Or, given enough persistence, you could write a routine to delete from memory specific variables (a kind of local CLEAR); though Machine Code would be better here. E-LINE up to STKEND are of more interest to the Spectrum than to its user.

The space between STKEND and RAMTOP, however, is important. It's the amount of free memory available. (Actually, the machine puts two stacks (see *Machine Code and Better Basic*) in there: one for the Z80A chip, one for the GOSUBs; but these are normally quite small.) So to estimate (to within a few dozen bytes) the memory that's spare, you use:

```
PRINT PEEK 23730 + 256 * PEEK 23731 - PEEK 23653 - 256 * PEEK 23654
```

or incorporate such a command into a program. In fact 256 * (PEEK 23731 − PEEK 23654) is simpler and close enough.

RAMTOP normally sits at the start of the user-defined graphics; but it can be lowered to make room for things that you don't want the BASIC system to clobber. For example, Machine Code routines (see *Machine Code and Better Basic*) or extra user-defined characters (see below). Stuff above RAMTOP is not affected by NEW. (It's also not saved; but you can use byte storage, see the *Manual* page 142, to get round that.)

UDG is where the user-defined graphics area starts. The main reason for wanting to change this is if you're short of memory and don't want all 23 user-defined characters. Then you *raise* the value to release the extra space.

BUILT-IN CLOCK

The systems variable FRAMES counts the number of TV screen frames that have been scanned since the computer was last switched on. It scans 50 frames per second, so effectively you have a built-in clock. The *Manual*, pages 129–131, goes into this quite thoroughly, so I won't repeat the description here; but the main point to notice is that the variable has *three* bytes: its numerical value is

```
PEEK 23672 + 256 * PEEK 23673 + 65536 * PEEK 23674
```

and the number of seconds elapsed is this, divided by 50. Notice that address 23674 only changes every 65536/50 = 1310.72 seconds, or about twenty minutes. So for a lot of applications (such as the next) the first two bytes are all that matter.

Here's a program to test your reaction-time.

```
10 PRINT "Reaction time test"
20 RANDOMIZE
30 PAUSE (100 + 300 * RND)
40 LET t0 = PEEK 23672 + 256 * PEEK 23673
50 PRINT AT 10, 15, "GO!"
60 IF INKEY$ = "" THEN GO TO 60
70 LET t1 = PEEK 23672 + 256 * PEEK 23673
80 LET t = t1 - t0
90 IF t < 0 THEN LET t = t + 65536
100 PRINT AT 15, 2; "Your reaction time is □"; t/50; "□ seconds"
```

Line 90 takes care of the (unlikely but possible) event that the third byte of FRAMES has clicked over by 1 during the reaction period.

To use the program, RUN it: as soon as "GO!" appears on the screen, press any key. (Don't cheat by holding it down!)

THE CHARACTER SET

The systems variable CHARS holds the address of the *character set*—a table of 0s and 1s that defines the shape of the screen characters. More precisely, it holds 256 less than the address; and the table starts with character 32, a SPACE.

Ordinarily, CHARS takes the value 15360. The instructions for printing out character number n are contained in bytes 15360 + 8 * n to 15360 + 8 * n + 7, and give the eight rows of the 8 × 8 square of pixels for that character, in binary, just as for user-defined characters (*Easy Programming*, page 49).

You can PEEK these, to see how a given character is stored in ROM and built up on screen, using this program.

```
10  INPUT "Character ?"; c$
20  IF c$ = " " OR LEN c$ > 1 OR CODE c$ < 32 THEN GO TO 10
30  LET n = CODE c$
40  FOR i = 0 TO 7
50  LET x = PEEK (15360 + 8 * n + i)
60  LET p$ = " "
70  FOR t = 1 TO 8
80  LET xs = INT (x/2): LET xj = x - 2 * xs
90  LET p$ = ("*" AND xj) + ("." AND NOT xj) + p$
100 LET x = xs
110 NEXT t
120 PRINT p$
130 NEXT i
```

This takes the binary numbers in ROM and turns 0 into a dot and 1 into an asterisk. So the letter "a" gives this:

Address in ROM	Contents in binary	Graphic effect
16136	0 0 0 0 0 0 0 0	. . . . .
16137	0 0 0 0 0 0 0 0	. . . . .
16138	0 0 1 1 1 0 0 0	. . * * * . . .
16139	0 0 0 0 0 1 0 0	. . . . . * . .
16140	0 0 1 1 1 1 0 0	. . * * * * . .
16141	0 1 0 0 0 1 0 0	. * . . . * . .
16142	0 0 1 1 1 1 0 0	. . * * * * . .
16143	0 0 0 0 0 0 0 0	. . . . .

See the "a" shape among the stars?

The dots and stars are for clarity: to get the true effect, change "." to "□" and "*" to "■".

You can use this technique in programs to produce letters eight times the usual size. By replacing each square by a 2 × 2 square, you can get 16 times original size: a bit chunky, but dramatic. By manipulating the eight binary numbers in 2 × 2 blocks, you can bring the graphics characters into play and produce letters 4 times the usual size (see over). And, by inventing suitable user-defined graphics, you can produce characters double the usual size.

More striking is the effect of POKEing CHARS.

Get a program into RAM which has about a page of listing, for better effect. Enter by direct command

POKE 23606, 2

Hmmm . . . that's a bit weird: the characters have shifted a couple of lines and got a bit broken up. Now try

POKE 23606, 8

(and ignore the screen display messages, which also look weird). Now you've got nice letters again, but the listing seems to be in some sort of code . . . In fact each letter has got changed to the one next to it in the character table, because we've fooled the computer into looking eight bytes ahead of where it thinks it is.

For *real* fun, enter

POKE 23607, 50

Judicious use of this command, in a program, would make listings of it incomprehensible. However, a bit of detective work by the would-be-pirate would edit them out again . . . Life is Sad . . .

To get back to a comprehensible display, enter

POKE 23606, 0: POKE 23607, 60

but be careful since you won't be able to see what you're doing on the screen until it's all over. Pulling the plug for a second will reset if all else fails.

This is all very fascinating, but what's the point? The main point is that you can provide the computer with entire new character sets—as many as you have room for in memory—rather than just the 23 user-defined characters. The way to do this is explained in Chapter 15, because it would make this chapter too long at this point; the idea is to build up the new character set in a part of memory that the BASIC system doesn't use, by lowering RAMTOP and POKEing in the codes for the rows. Then the odd POKE to CHARS and all your new characters become accessible. POKE it back, and the old ones show up. Two tiny subroutines to do the POKEing, plus 2048 bytes of data, and you've got 256 new characters—graphics symbols, mathematical symbols, or whatever. Of course you don't have to go the whole hog: fewer than 256 characters can be set up by the same method, if you prefer.

By using the graphics characters (top row of keys) you can draw symbols at four times their usual size, which is quite nice for dramatic captions. Here's a program which accepts any string of length 8 or less, and expands it fourfold. (The limit on length is just so that the result fits the screen: you can easily modify it if you want to write several lines.) It's based on the clever way that the Sinclair people have (this time!) coded the graphics symbols. To get the CODE for a graphics symbol, add together the numbers in the grid

2	1
8	4

corresponding to its *black* squares; then add 128. For example ▚ has code 2 + 4 + 128 = 134. You can check this from page 92 of the *Manual*.

```
10 DIM q (8, 8)
20 LET i0 = PEEK 23606 + 256 * PEEK 23607
30 INPUT c$
40 LET s = LEN c$: IF s > 8 THEN LET s = 8
50 FOR r = 1 TO s
60 LET i = i0 + 8 * CODE c$ (r)
100 FOR a = 0 TO 7
```

```
110  LET m = PEEK (i + a)
120  FOR b = 0 TO 7
130  LET q(a + 1, 8 - b) = m - 2 * INT (m/2)
140  LET m = INT (m/2)
150  NEXT b
160  NEXT a
200  FOR u = 1 TO 4
210  LET t$ = " "
220  FOR v = 1 TO 4
230  LET k = q (2 * u - 1, 2 * v) + 2 * q (2 * u - 1, 2 * v - 1)
       + 4 * q (2 * u, 2 * v) + 8 * q (2 * u, 2 * v - 1)
240  LET t$ = t$ + CHR$ (128 + k)
250  NEXT v
260  PRINT AT u, 4 * r - 4; t$
270  NEXT u
280  NEXT r
```

Figure 6.1 shows the result of inputting "Testing?".

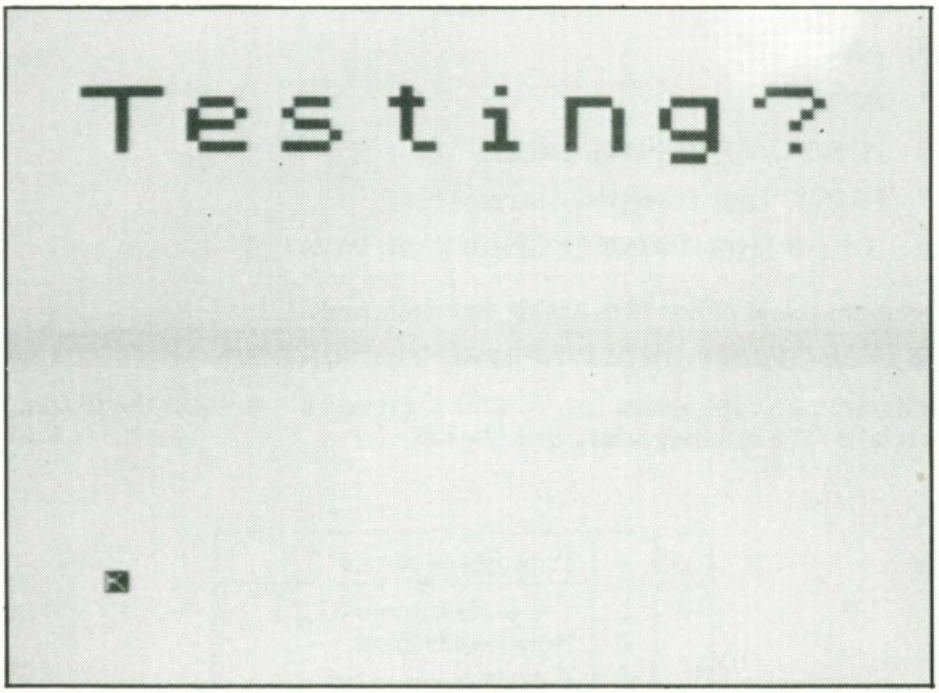


Figure 6.1 Testing quadrupled characters.

IMPOSSIBLE JUMPS

The Spectrum lets you leave out lots of line numbers (a major objection to BASIC—why number lines you don't want to refer to?) by writing several statements on one line:

```
10  LET a = 1: LET b = 49: PRINT a + b: GO TO 10
```

(say). All well and good, but you can only jump (with GO TO) to the *first* statement in such a line.

Unless you POKE the system variables NEWPPC and NSPCC, in addresses 23618–9 (2 bytes for NEWPPC) and 23620 (1 byte for NSPPC). This forces a jump to the line held in NEWPPC, and the statement number in NSPPC.

Rather than elaborate the theory, here's a test program that makes all clear.

```
3  INPUT a
5  POKE 23618, 10: POKE 23620, a
7  PRINT "I wasn't supposed to come here"
10 PRINT "1": PRINT "2": PRINT "3"
```

RUN this, and INPUT 1 for a; try again with a = 2 and a = 3.

Line 7 is never reached at all. The first command in line 5 forces a jump to line 10. If a is 1, the second command in line 5 forces a jump to the first part of line 10; if a is 2 it causes a jump to the second part; if a = 3, to the third.

In general the jump is forced by the command

POKE 23618, nj: POKE 23619, ns: POKE 23620, a

which has the same effect as

GO TO nj + 256 * ns, a-th statement

(were such a command possible in BASIC, which it isn't).

One oddity: IF/THEN statements. The THEN part is treated by the computer as a separate statement in the line, which has to be included in the count. And if you jump to the THEN part, using NSPPC, you'll get an error message "Nonsense in Basic". To see what happens, try this:

```
2  INPUT x
3  INPUT a
5  POKE 23618, 10: POKE 23620, a
7  PRINT "I wasn't supposed to come here"
10 IF x = 0 THEN PRINT "1": PRINT "2": PRINT "3"
```

The computer thinks of line 10 as having *four* commands:

(IF x = 0) (THEN PRINT "1") (PRINT "2") (PRINT "3")

so you'll have to try the values 1, 2, 3, 4 for a. Putting x = 0 makes the IF true; x = 1 makes it false. The resulting print-out is like this:

x	a	Print-out on screen
0	1	1 2 3 (in columns)
	2	Nonsense in Basic
	3	2 3
	4	3
1	1	(blank—condition not true)
	2	Nonsense in Basic
	3	2 3
	4	3

Notice that even with x = 1, jumps to the third or fourth statement produce a print-out: the IF/THEN is ignored (as it would be if the statements had been on separate lines, and you'd used a suitable GO TO).

THE BOTTOM OF THE SCREEN

The attributes for the lower part of the screen (where the error messages come) are set by the machine; and occasionally you can end up with a block of colour at the bottom which doesn't match the border, if you change the border in a program run. The system variable BORDCR at address 23624 contains 8 * c where c is the colour number. POKEing it doesn't produce an immediate effect, however; but if you don't mind clearing out your variables, you can reset the colour using

```
POKE 23624, 8 * c: CLEAR
```

You may find a use for this. See also Chapter 7 on the display and attribute files.

While on the topic of the "Message" section of the screen, there's another way to get at it by using a command designed for the microdrive. Try this program (# is on key 3, symbol shift).

```
10 PRINT #0; "Hello there!"
20 GO TO 20
```

(The last command is there just to stop the "OK" message appearing and obliterating everything.) Using PRINT #0 you can print to the "Message" section. There are some quirks; for example, you tend to lose line 21 of the main section, or get scrolls when you don't expect them. You can PRINT #0 a message of more than 64 characters: the lower section expands and scrolls up. Using colour commands and/or control characters you can get colour into the act as well.

If you've got a printer attached, try using PRINT #3 instead. This time, the message goes to the printer—just like LPRINT.

PRINT #1 also produces messages in the lower section of screen; and PRINT #2 produces them in the usual place—like plain PRINT. The other PRINT #n commands, for n > 3, refer to the microdrive system.

This is useful to know if you want to print out a series of messages (say) with the options of using the screen *or* the printer. Use

```
PRINT #n; "Whatever the message is"
```

Then setting n = 2 gives screen, n = 3 printer. This is a lot better than the alternative

```
IF n = 2 THEN PRINT "Whatever the message is"
IF n = 3 THEN LPRINT "Whatever the message is"
```

SCROLLING

To my mind, the most infuriating feature of the Spectrum is the way it asks for scrolls. You have to keep hitting the keyboard to make it scroll, or stop it scrolling; and mine usually ends up scrolling when I want it to stop, and stopping when I want it to scroll. The whole system is distinctly perverse and I wish Sinclair had thought a bit harder about it.

If you want to force a scroll from a program, you can use the systems variable SCR-CT, address 23692. This holds the number of lines left before a scroll request is printed out. So the command

```
POKE 23692, 2: PRINT AT 21, 31: PRINT
```

causes it to scroll up one line, *without* asking for a keyboard input. (I mentioned this in *Easy Programming*, but it's worth mentioning again for completeness.)

PLOT COORDINATES

I mentioned these too: same reasoning applies. The two-byte systems variable COORDS holds the pixel coordinates of the last point PLOTted. The addresses are:

23677 row coordinate,

23678 column coordinate.

So, if you've just used

PLOT 47, 124

then address 23677 holds 47, and 23678 holds 124. Check this:

```
10 INPUT row, column
20 PLOT row, column
30 PRINT PEEK 23677, PEEK 23678
```

You use COORDS, of course, to find out where you are at the moment before doing a DRAW (which moves you from the current PLOT position by the offsets specified in the DRAW command, as you well know). If you want to draw to a specified point a, b—and have forgotten, or not stored in the machine, the current PLOT position—you can use

DRAW a − PEEK 23677, b − PEEK 23678

The main use for this would be in curve-plotting, but you might also find a use for it if you use the keyboard to move a pixel around (as in Chapter 1, the Sketchpad program) and want to know where you are.

Ever got a really nice diagram on the screen, but in a terrible colour combination? And the only way to change it is to change the program, which wipes the screen, so you have to start again . . .

There's a better way.

7 Attribute and Display Files

If you've owned a ZX81 you'll know that it runs at two speeds, sensibly called FAST and SLOW, and that the screen display blanks out in FAST. You'll also be aware (see *Machine Code and Better Basic*) that the screen display is held in a part of RAM called the display file, which moves around and is held by a system variable called D-FILE. (If you haven't owned or used a ZX81, skip this paragraph. Oh dear, too late.)

The Spectrum isn't like this: it is pure and un-D-FILEd.

The hardware for the screen display works all the time, so there is no SLOW speed (and hence no need for speed commands). The information for the screen display is held by two chunks of memory, the *attributes file* and the *display file*. These live in fixed positions in RAM, and work in different ways.

ATTRIBUTES

The attributes file is easier to understand (and, unless you're a Machine Code buff, more useful anyway).

The screen display consists of 22 lines (24 including the bottom bit for messages) of 32 characters each, making a total of 22 * 32 = 704 (or 24 * 32 = 768) positions. PRINT AT r, c produces a character in the c-th position along row r (counting from 0 upwards). The character printed there is held in display file; but its attributes (colour, flash, etc.) are held in the attributes file. The order in which the attributes are held is straightforward: start at top left and read along rows from left to right, just like reading a book. The attributes file starts at address 22528; so the attribute for row r, column c is held at address:

```
22528 + 32 * r + c
```

You can POKE this to change the attribute without clearing the screen. This is useful, because the PAPER and INK commands only affect newly printed stuff.

Each attribute is a single byte, whose eight bits are divided up like this:

FLASH on/off	BRIGHT on/off	three bits	for	paper colour	three bits	for	ink colour

where "on" is 1, "off" is 0 as usual.

In other words, for FLASH value f, BRIGHT value b, PAPER value p, and INK value i, the attribute value is:

```
128 * f + 64 * b + 8 * p + i
```

You can find out the attribute of row r, column c, by using

```
LET att = ATTR (r, c)
```

and you can convert this to a list of values f, b, p, i using:

```
LET f = INT (att/2)
LET b = INT (att/4) − 2 * f
LET p = INT (att/32) − 16 * f − 8 * b
LET i = att − 8 * INT (att/8)
```

The following program converts the screen display so that every cell has the same (chosen) attribute. For example, if you've laboriously fed in a world map in black ink on white paper, and now you want red ink on yellow paper, you can make the change without losing the map or starting again. Use direct entry, and type:

```
LET f = 0: LET b = 0: LET p = 6: LET i = 2:
LET att = 128 * f + 64 * b + 8 * p + i: FOR t = 0
TO 703: POKE 22528 + t, att: NEXT t
```

Better still, have this in memory already, as part of a general utility program: then you can INPUT f, b, p, i. Of course, you could also work out in your head that att = 8 * 6 + 2 = 50 here; so the command becomes

```
FOR t = 0 TO 703: POKE 22528 + t, 50: NEXT t
```

and you're in business!

If you change the loop to

```
FOR t = 0 TO 767 etc.
```

then you'll also change the lower part of the screen; and

```
FOR t = 704 TO 767
```

changes *just* that bit. The main time you need this is when you've loaded in a *picture* (LOAD "Mona Lisa" SCREEN$) and it ends up with the wrong colour for the lower part of the screen, thanks to changes in BORDER since the picture was SAVEd. Unlike BORDCR, Chapter 6, you don't have to CLEAR variables. You won't need this often, but it might just be useful.

For rapid attribute-changes, this routine is too slow. The remedy is to go to Machine Code, see *Spectrum Machine Code*, by Ian Stewart and Robin Jones, Shiva, Chapter 14.

DISPLAY

This is much more complicated, because each character in the display is stored as a list of *eight* bytes (its rows in 8 × 8 pixel form; see Chapter 15). And *these are not stored in the obvious order*.

If you've ever loaded a picture using SCREEN$ you'll have noticed the curious order in which the computer "paints in" the picture. It does it in exactly the order that the bytes are stored in the display file.

To see this clearly, RUN this program:

```
10 FOR r = 0 TO 21
20 PRINT "[32 inverse spaces, or graphics character g8c's]"
30 NEXT r
```

Then save it: SAVE "blank" SCREEN$. Finally, CLS and LOAD "blank" SCREEN$ and watch what happens. (If you miss out the CLS you'll not get much joy . . .)

It's easier to describe the process in terms of the coordinates used for hi-res graphics, a 256 × 176 grid. Number the rows from 175 at the top to 0 at the bottom, and the columns from 0 at left to 255 at right.

The display file starts at address 16384.

The first 32 bytes hold the instructions for line 175. Each byte governs an 8-column segment of that line. The first byte deals with columns Ø–7, the next with 8–15, and so on. If you CLS and then input

```
POKE 16384, BIN ØlØlØlØl
```

or the decimal equivalent

```
POKE 16384, 85
```

you'll see a row of four dots

. . . .

at top left. Change it to POKE 16385, 85 and so forth, and see how the row of dots moves. The dots, of course, are the binary number ØlØlØlØl converted to graphics: 1 being "plot a pixel" and Ø "plot a blank" on the hi-res grid.

Each horizontal line of display goes into the display file as a sequence of 32 bytes, in order.

Unfortunately, the lines themselves don't go in numerical order at all, as we've just seen.

First these lines go in:

175 167 159 151 143 135 127 119

(think of them as the top line of the first eight rows of characters). Then the computer goes on to the second lines:

174 166 158 15Ø 142 134 126 118

Then

173 165 157 149 141 133 125 117

and so on until it reaches line 112. By now the top third of the screen (rows Ø–7 in lo-res) have been dealt with.

Then the next third is dealt with in a similar order; and finally the last third, including the "messages" section of screen.

I've described this mostly to satisfy your curiosity: you'll only need to know it if you want to write moving graphics in Machine Code, or that sort of thing. If you *do* want to do that, it would take too much space here to describe how: *Spectrum Machine Code* takes the topic further.

Projects

1. Modify the attribute-changing program so that different sections of screen get different attributes, by using suitable DATA statements. Use it to produce a "solid" map of the world, like the one in Chapter 2 but with the continents (so far as is practicable) shaded in different colours, and cyan oceans.

2. If you're mathematically minded, show that the following sequence of instructions calculates the position in display file corresponding to a hi-res pixel in row r, column c.

```
10 INPUT r, c
20 LET r1 = 175 - r
30 LET b = INT (r1/64)
40 LET c1 = INT (c/8)
50 LET c2 = c - 8 * c1
60 LET r2 = INT (r1/8)
70 LET r3 = r1 - 8 * r2
80 LET a = 2048 * b + 256 * r3 + 32 * r2 + c1 + 16384
90 LET bitpos = c2 + 1
100 PRINT "The pixel with hi-res coordinates □"; r; "□"; c;
    "□ is stored in the display file at address □"; a; " □ at bit number □"; bitpos
```

(Here the bit-position bitpos runs from 0 to 7 along the byte from most significant figure to least, like this: 01234567.)

Sinclair presumably have a good reason for using this particular order . . .

And now the Spectrum gets its own back by taking a look at your *system variables . . .*

8 Psychospectrology

. . . The art of psychological experiment with a Spectrum.

The idea of this chapter is to use the computer to investigate certain curious phenomena in the psychology of visual perception: how we see things. The computer is ideal for this: it can set up carefully controlled pictures ("stimuli" as the psychologists say) to bring out specific features of the perceptual mechanism. (See, I'm getting the hang of the jargon already, and we've barely got started.)

AFTERIMAGES

If you stare at a bright object for long enough, the cells in the eye that receive the light become "tired" and cease to respond so strongly to it. This results in the formation of "afterimages": dark spots the same shape as the original bright object. The effect wears off after a few seconds.

Suppose the original object is not only bright, but *coloured*. How do the afterimages behave? The following program lets you find out, using the Spectrum to generate and maintain the original stimulus.

```
10  PRINT AT 19, 0; "Afterimages"
20  PRINT "Stare at the square"
30  INPUT "Colour?", p
40  FOR i = 1 TO 6
50  PRINT BRIGHT 1; PAPER p; AT 8 + i, 13; "□□□□□□"
60  NEXT i
70  PAUSE 500
80  CLS
90  GO TO 10
```

When you RUN this, input the colour by pressing the corresponding number in the top row of keys. A bright square will appear: stare at it, trying not to move your eyes off it. When it disappears, you should see a fuzzy, phantom square, which fades in a few seconds. (If you don't, or it's very pale, try increasing the PAUSE to about 1000.)

If your eyes are working like mine, the colours you see should be roughly these:

Original	*Afterimage*
black	white
blue	yellow
red	cyan
magenta	green
green	magenta
cyan	red
yellow	blue
white	black

Well, OK, maybe the "black" looks grey, and the red is a bit brownish; but it should be close.

Notice that the codes for the original and the afterimage add up to 7 (e.g. red + cyan = 2 + 5 = 7). This means that the image colour is *complementary* to the original: it contains precisely those colour signals that are *not* present in the original.

Why? Presumably because overexposure to these has reduced the eye cells' capacity to respond to them, which shows up as a kind of exaggerated response to the complementary colours.

In ordinary circumstances, you won't notice afterimages much (unless you look too close to the Sun, which is dangerous if done for more than a split second, so *don't*). This is because the eye is constantly hopping from one point to another. (It's called *saccadic* movement.) But this Spectrum experiment shows them up quite vividly. (Don't stare *too* long at the Spectrum screen, either!)

THE HERING ILLUSION

This dates from 1861, and is named after its discoverer Ewald Hering.

```
10  FOR i = 5 TO 255 STEP 12
20  PLOT i, 0: DRAW 255 - 2 * i, 175
30  NEXT i
50  FOR i = 5 TO 175 STEP 12
60  PLOT 0, i: DRAW 255, 175 - 2 * i
70  NEXT i
100 INPUT "Spacing?", q
110 PLOT 0, 87 - q: DRAW 255, 0
120 PLOT 0, 87 + q: DRAW 255, 0
```

It draws a bunch of rays through the centre of the screen. Then it asks for an input. Try something between about 10 and 20 first. Two lines appear. They look *curved* (although the curvature of the TV screen can spoil the illusion a bit).

However, it's clear from lines 110 and 120 that they must be *straight*. The other lines fool the eye.

Try this with different values for the "Spacing" INPUT, and different INK/PAPER combinations. What values give the best illusion?

THE WUNDT ILLUSION

It took until 1896 for someone to try the obvious, and make the lines curve the other way. The genius responsible was Wilhelm Wundt, the first man to suggest that psychologists might care to carry out *experiments*. Big Oak Trees and all that . . .

```
10  FOR i = -125 TO 125 STEP 10
20  PLOT 127, 0: DRAW i, 87: DRAW -i, 88
30  NEXT i
40  FOR i = 2 TO 87 STEP 10
50  PLOT 0, i: DRAW 127, -i: DRAW 128, i
60  PLOT 0, 175 - i: DRAW 127, i: DRAW 128, -i
70  NEXT i
100 INPUT "Spacing?", q
210 PLOT 0, 87 - q: DRAW 255, 0
220 PLOT 0, 87 + q: DRAW 255, 0
```

It works very much the same way. I reckon the illusion is more effective on the TV screen than Hering's version: what do you think?

THE POGGENDORF ILLUSION

Have you noticed how they're all German? Johann Poggendorf proposed this one in 1860.

```
10  FOR i = 0 TO 20
20  PLOT 20, 70 + i: DRAW 200, 0
30  NEXT i
40  PAUSE 50
50  PLOT 20, 10: DRAW 200, 130
60  IF INKEY$ = "" THEN GO TO 60
70  OVER 1: PLOT 113, 70: DRAW 30, 20
80  OVER 0
```

This draws a rectangular block, and two lines radiating out from it. They look as if they're offset.

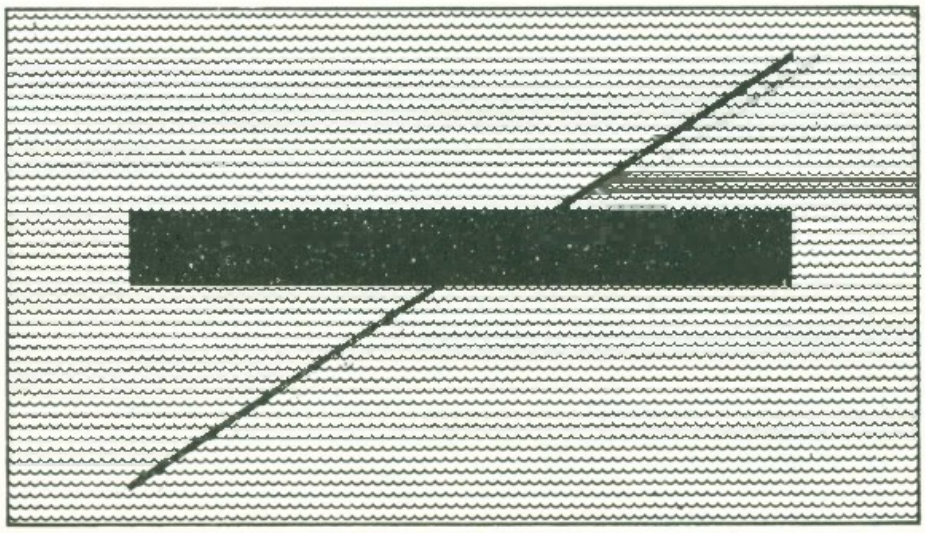


Figure 8.1 The Poggendorf Illusion: is the whole line straight?

However, if you press a key, a white line is drawn between them, to show that they are aligned with each other: the whole line now looks straight.

Hmmm . . .

THE MÜLLER-LYER ILLUSION

At first I thought it was *two* Germans . . . but no: Franz Müller-Lyer had a double-barrelled name (Doppelname, as they say).

```
10  PLOT 40, 130: DRAW 180, 0
20  PLOT 40, 60: DRAW 180, 0
30  PLOT 25, 45: DRAW 15, 15: DRAW -15, 15
40  PLOT 235, 45: DRAW -15, 15: DRAW 15, 15
50  PLOT 55, 115: DRAW -15, 15: DRAW 15, 15
60  PLOT 205, 115: DRAW 15, 15: DRAW -15, 15
70  IF INKEY$ = " " THEN GO TO 70
80  FOR t = 1 TO 7
90  PLOT 40, 140 - 10 * t: DRAW 0, -7
100 PLOT 220, 140 - 10 * t: DRAW 0, -7
110 NEXT t
```

You've seen this one. There are two arrows, with heads pointing different ways. The bottom one is clearly longer. Oh yes? Hit any key; watch the dotted lines . . .

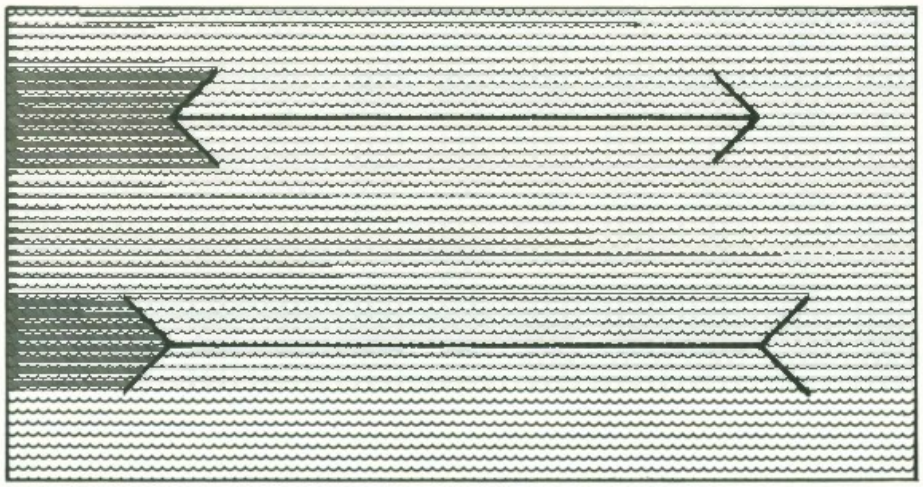


Figure 8.2 The Müller-Lyer Illusion: are the lengths the same?

THE WERTHEIMER ILLUSION

Due to Max Wertheimer in 1912 . . .

```
10  PAPER 0: INK 6: BORDER 0: CLS
20  INPUT "Pause?", p
25  INPUT "Number?", n
30  FOR t = 1 TO 20
```

```
35 FOR j = 1 TO n
40 PRINT AT 8, 15 + 2 * (j - 5); "□"
45 NEXT j
50 PAUSE p
55 FOR j = 1 TO n
60 PRINT AT 8, 15 + 2 * (j - 5); "■"
70 PRINT AT 14, 15 + 2 * (j - 5); "□"
75 NEXT j
80 PAUSE p
85 FOR j = 1 TO n
90 PRINT AT 14, 15 + 2 * (j - 5); "■"
95 NEXT j
100 NEXT t
```

When you RUN this, you'll be asked for a PAUSE—anything between 1 and 100 is fine—and a NUMBER. I suggest you input 1 for the first few tries.

The screen displays a yellow square (or two). If PAUSE is small, you'll see two (possibly flashing) lights. If PAUSE is large, you'll see one light for a while, then the other. But, if PAUSE is in between, what you see is *one* light, bouncing up and down. (Try PAUSE = 35.) You can see this effect with the fog-warning lights on British motorways.

If you try NUMBER at values bigger than 1 (and not more than 10 or 12) you'll see a whole array of lights. What you perceive depends a bit on the fact that they aren't flashing quite in synchrony: the BASIC program takes time to go round a loop. Experiment.

This illusion is important for computer-games: moving graphics wouldn't work without it. For that matter, neither would TV or motion pictures. No Logie Baird, and no Yogi Bear.

THE SCHWINDEL ILLUSION

. . . Due to Hans-Wilhelm Schwindel in 1872. Well, actually, no: I invented it myself, but you'd never have guessed—unless you knew that "Schwindel" is German for "fake" (hence "swindle"). No doubt half a dozen Germans invented it half a century ago, but not on a Spectrum.

```
10 OVER 1
20 POKE 23607, 50
30 FOR i = 1 TO 704: PRINT CHR$ (35 + 10 * RND);: NEXT i
40 POKE 23607, 60
50 LET i = 125 * RND: LET j = 85 * RND
60 LET i1 = 125 * RND: LET j1 = 85 * RND
70 PLOT i, j: DRAW i1, j1
80 PAUSE 20
90 GO TO 50
```

This draws a random screenful of hi-res dots. (Don't BREAK in the middle of this bit, or you'll get weird messages. If you *do* BREAK, enter POKE 23607, 60.)

Then it starts drawing lines. You'll see the lines go in; but once they're in, you'll totally lose track of where they were. The eye is detecting *changes* in the random screen, but it can't hold the resulting image in any detail. Something to do with the way we perceive *textures*.

For a variant, try circles. Change lines 50–90 so that they read:

```
50  LET i = 50 + 150 * RND: LET j = 40 + 90 * RND
60  LET r = 20 * RND + 10
70  CIRCLE i, j, r
80  PAUSE 20
90  GO TO 50
```

Again, you'll see them going in; but the image won't last. If you're not even convinced the circles are going in at all, add a colour-change:

```
55  INK 3
```

Now you'll see that something's changing; but you won't start to see the circles until a goodly part of the screen has gone magenta. The colour change seems to impress the eye (and brain) even more than the change in the individual pixels, and it obscures the latter almost entirely.

Most commercial uses of computers require large quantities of data. These are stored on disc or tape in the form of . . .

9 Files

What is a file, in the computing context? Scientists have a disconcerting habit of taking a word in common use and giving it a subtly altered meaning to suit their purposes. Computer scientists have done just that with the word file. To them, the word implies a (usually) large collection of data items related to some specific topic.

So for instance, an estate agent would hold a file consisting of all the properties which he has available for sale. He should beware of saying to his secretary, "Bring me the file on 36 Acacia Avenue, please," within earshot of a computer man because he will be smugly told that he is misusing the word, since he only really wants one item of the file, which the computer man calls a *record*. Our estate agent may retort, with considerable justification, that he was using the word "file" before the computer man was thought of. The computer man will, wisely, ignore this line of argument, and camouflage the fact that he hasn't responded to it by remarking that a single feature of a record, such as, in this case, the current owner of 36 Acacia Avenue, or the price being asked for the property, is called a *field*. At this point the estate agent will almost certainly disengage himself from the argument, since he doesn't see much percentage in selling fields, particularly without outline planning permission.

Let's look at another example of a file of a more personal nature. Suppose you wanted the details of your record collection kept. The file is the sum total of all the information about this collection. A record of the file is, coincidentally, information about one record (thing that whirls round at 33⅓ r.p.m.) in the collection. We'll assume for simplicity that all the (gramophone) records are of the pop variety, and therefore have twelve separately titled tracks. We'll worry about what to do with the Brandenburg Concertos later. So each file record might consist of the fields given in Table 9.1.

Table 9.1

Field No.	Description	No. of characters (max)
1	Artist	30
2	Date purchased	6
3	Track 1 title	20
4	Track 1 length (minutes)	5
5	Track 2 title	20
6	Track 2 length (minutes)	5
7	:	
8	:	
.	and so on	
.	:	
.	:	
.	:	
25	Track 12 title	20
26	Track 12 length (minutes)	5
	Total no. of characters/record:	336

Each of the fields described in the table has a fixed length. So, for instance, if the artist is "Pink Floyd", which only occupies 10 characters (including the space between the words) a further 20 spaces will have to be added to make up the standard 30 symbols. Thus "30" is just a reasonable guess as to the maximum number of characters which the artist's name might take up. It should be adequate for most purposes; even "Bonzo Dog Doo-Dah Band" fits OK. But not "Dave Dee, Dozy, Beaky, Mick and Titch".

Now, you may object that a lot of memory is going to be wasted this way, and that it would be better to allow fields to have variable lengths, and to delimit each of them with some special character. To that, I can only answer that your argument is sound but that to do things like that would make the programming much more difficult than I am prepared to make it, so there. Anyway, in at least some cases, there is no question about the field length. For example, the "Date purchased" field always contains exactly 6 characters: 2 for the day, 2 for the month and 2 for the year, as in 031178 for "3rd November 1978". Similarly, the duration of a track is *almost* fixed. I've allowed for 2 decimal places, so that the time can be quoted as 3.16 minutes, for instance. That only takes up four characters, however (including the decimal point), and I've allowed five. This is just to guard against the possibility that a track is ten minutes (or more) long.

As I've shown, the whole record occupies 336 characters using these assumptions. So if you've got 200 records in your collection, the file is going to occupy 67200 bytes, which is a good deal more than the Spectrum has space in memory for at one time!

This is a typical feature of computer files; by and large we expect them to occupy more space than there is available in main memory. That means they have to be held on some backing store, and for our purposes, that means cassette tape.

FILES ON TAPE

What form should the file on the tape take?

Before answering that question, it will be helpful to think a little about how we're going to use it when we've got it. One thing that's going to be necessary fairly frequently is to revise the file to include newly acquired records, and to delete discarded ones. This is known as *file maintenance*.

Now, we can't *actually* delete something from a small segment of cassette tape. The only way to do the job is to copy the entire file, except for the bit we want to delete, from one tape to another. The original tape is unaffected, but the new one no longer contains the deleted record. Figure 9.1 indicates how this can be arranged, at least in principle.

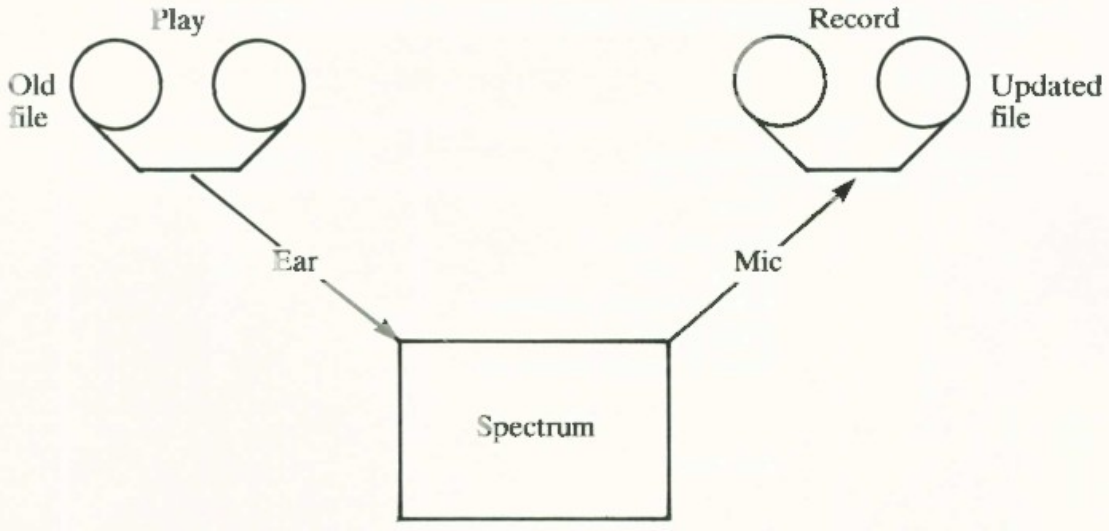


Figure 9.1 Use of two cassette-recorders to handle taped files.

So we need two tape recorders, one in "play" mode (connected to the "ear" sockets), the other in "record" mode (connected to the "mic" sockets).

What are the programming considerations here? In outline, the code is:

1. Read a record.
2. If it's the terminating record then end.
3. If it's the one to be deleted then go to 1.
4. Write a record.
5. Go to 1.

Now, if the tape is left running, there's no guarantee that by the time steps 2, 3 and 4 have been executed the next record hasn't passed through, so obviously, every time a record is read, the tape will have to be stopped. Similarly, there's no point in leaving the tape being written to running when there's nothing being written to it. Since it's going to take less than 2 seconds to read or write a record, the starting and stopping of tapes is going to get pretty tedious pretty quickly.

BLOCKS OF DATA

So what we need is a compromise. We can't hold the whole file in memory and we don't want to deal with it record by record, so why not split it into blocks of so many records each, so that a block can be comfortably held in memory at one time? For a 16K Spectrum, about 9K is available to the user. If the program occupies 1K, we can afford blocks of about 4K in size so that, at any given time, there is a 4K block which has been read in (to an *input buffer*) and whose contents are being transferred to an *output buffer* for subsequent writing out to tape. So now our organization looks like Figure 9.2.

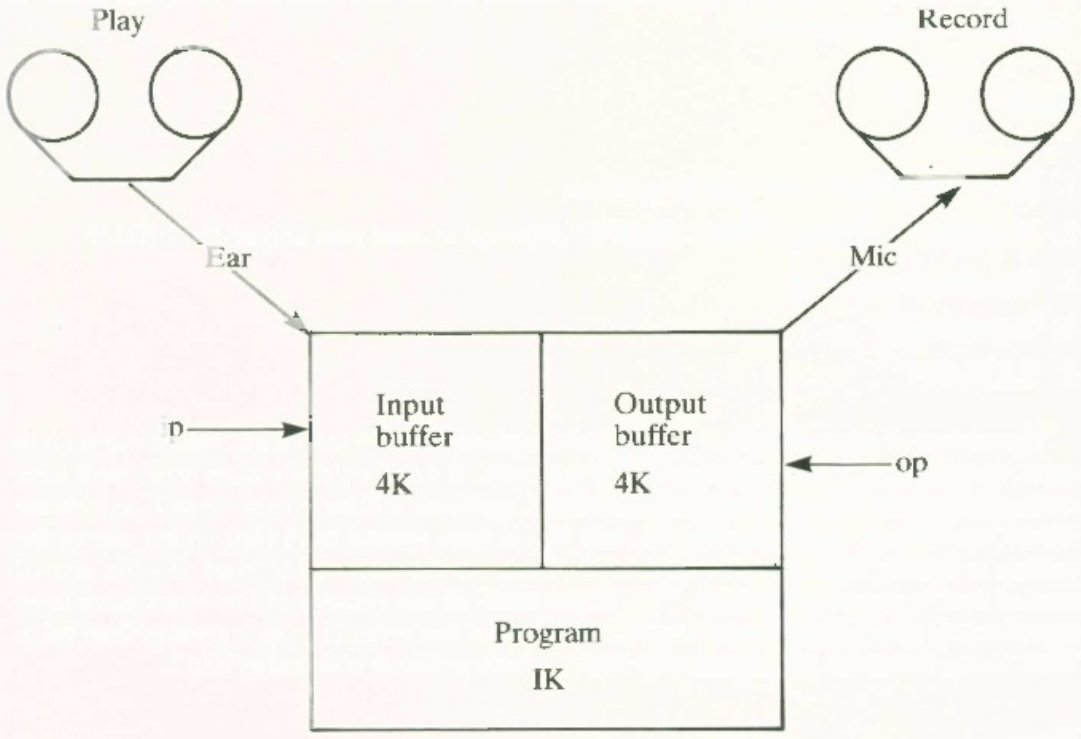


Figure 9.2 Arrangement of memory for file-handling.

For the record collection example, we'll be able to get 12 records per block.

So far as the user is concerned, it'll be convenient if he doesn't have to think about the housekeeping aspects of this arrangement. In fact, it'll be simplest if it *appears* as though

single records are being read and written. We'll need two subroutines for this, "read a record" (*read*) and "write a record" (*write*). Most of the time, these routines won't actually be doing any reading or writing at all, but simply transferring data from the input buffer or to the output buffer. However, when the input buffer is empty it'll be necessary to read the next block, and conversely, when the output buffer is full, we'll have to write a block. We'll call these routines *getblock* and *putblock*.

There are a couple of other considerations: first, we need two *pointers*, ip and op, to show how full each buffer is at any time. These will have to be set up to zero to start with, so we'll have a routine called *initcfs* (for "initialize cassette file system") to do this, and any other initializations which turn out to be necessary. Second, we haven't thought about how the file should be terminated. Obviously, there's no guarantee that the file is exactly so many blocks long, so that we have to have some way of forcing the activation of *putblock* when a file delimiter of some kind is recognized. We'll adopt the convention that field 1 of the terminating record is "cfsend" (on the grounds that it's very unlikely that these letters are going to mean anything within a file), and that the other fields of this record are insignificant, so that, in practice, they may contain anything.

One more thing; we haven't said exactly how the input and output buffers are to be set up. Clearly, they are both string arrays. We'll allow the user the facility of deciding how big the buffers should be for a particular application. The simplest way of doing this is to prompt him for the number of records per block (nrb) and the number of bytes per record (bpr). Then we dimension two arrays:

```
DIM i$ (nrb, bpr)      [input buffer]
DIM o$ (nrb, bpr)      [output buffer]
```

This can be done in *initcfs*.

Armed with these ideas, we can start to look at the way *read* and *write* will work. In outline they are as follows.

Read

1. IF ip = 0 OR ip > nrb THEN getblock
2. IF i$ (ip) = "cfsend" THEN PRINT "Attempt to read past end of file": STOP
3. Transfer i$ (ip) to r$
4. Increment ip
5. RETURN

The action of the pointer ip needs a little explanation. To start with, *initcfs* will set it to zero, so of course that means the buffer is empty. If a call to *read* occurs now, we'll have to invoke *getblock*. This is catered for in line 1. *getblock* will have to set ip to 1, to indicate where the first record to be read is, and this is then passed to r$ (line 3) for use by the calling program. Now we increment ip so that, on the next call to *read*, it's the second record that is transferred to r$. This is fine until all the records of the block have been transferred, when ip will point beyond the end of the array (i.e. ip is greater than nrb). That's why there are two conditions under which *getblock* is called in line 1.

Write

1. Increment op
2. Transfer r$ to o$ (op)
3. IF op = nrb OR r$ = "cfsend" THEN putblock
4. RETURN

Line 1 increments the pointer op straight away, because *initcfs* sets it to zero, and that's before the beginning of the output array o$. So, on the first call, what's in r$ will be transferred to o$ (1) which is what we want. We know the buffer is full if op = nrb but, also, we want to force the execution of *putblock* if our end of file marker has been passed. That's why both conditions are tested for in line 3. Incidentally, *putblock* has to reset op to zero, so that it gets reset to 1 at the beginning of the first call to *write* after a call to *putblock*.

getblock and *putblock* are pretty straightforward too, but we need to overcome one more problem before outlining them: we haven't thought about how to name the data which are going to be saved. We could give each block the same name, but this could easily get confusing and is asking for trouble. A better technique is to allow the user to provide names for his files (within *initcfs* again) and then alter the name automatically within *putblock*, so that every block has a different number. For instance, if the user calls a file "fred", the blocks will actually be saved as fred0, fred1, fred2, etc. Similarly, *getblock* will have to keep a record of the block count on input. Again, the blockcounts will be initialized in *initcfs*.

So the outlines are:

getblock

```
PRINT "turn on PLAY recorder"
LOAD input file + inblockcount DATA i$ ( )
PRINT "turn off PLAY recorder"
LET ip = 1
LET inblockcount = inblockcount + 1
RETURN
```

putblock

```
PRINT "turn on RECORD recorder"
SAVE output file + outblockcount DATA o$ ( )
PRINT "turn off RECORD recorder"
LET op = 0
LET outblockcount = outblockcount - 1
RETURN
```

Note that inblockcount and outblockcount are going to be slightly trickier to handle than it appears, because part of the time they're used as strings to be added to the filenames, and part of the time they are numbers to be incremented. Also we've now got quite a few strings lying about which won't be easily identifiable because they can only have single character names, so before we write the actual code here's a list of string names and functions.

Stringname	*Function*
i$	input buffer
o$	output buffer
r$	record which user apparently writes to or reads from
f$	input file name
g$	output file name
n$	block number to be appended to input or output file name

THE CODE

We'll start the cassette file system from 9500 onwards, allowing 100 for each routine, so *initcfs* is at 9500, *read* is at 9600 etc. So we'll need lines 1 to 3:

```
1 LET initcfs = 9500: LET read = 9600
2 LET write = 9700: LET getblock = 9800
3 LET putblock = 9900
```

in any program which uses cfs.

Let's write *initcfs*:

```
9500 LET ip = 0: LET op = 0
9510 LET inbc = 0: LET outbc = 0          [block counts]
9520 INPUT "input file name"; f$
9525 INPUT "output file name"; g$
9530 INPUT "no. of bytes per record"; bpr
9540 INPUT "no. of records per block"; nrb
9550 DIM i$ (nrb, bpr): DIM o$ (nrb, bpr)
9560 RETURN
```

No problems so far. *read* should be pretty straightforward:

```
9600 IF ip = 0 OR ip > nrb THEN GO SUB getblock
9610 IF i$ (ip) (TO 6) = "cfsend" THEN PRINT "attempt to read past end
     of file": STOP
9620 LET r$ = i$ (ip)
9630 LET ip = ip + 1
9640 RETURN
```

and so should *write*:

```
9700 LET op = op + 1
9710 LET o$ (op) = r$
9720 IF op = nrb OR r$ = "cfsend" THEN GO SUB putblock
9730 RETURN
```

They're pretty well identical to the outline programs, aren't they?

getblock and *putblock* require a little jiggery-pokery to handle the string-to-numeric conversions:

```
9800 LET m$ = STR$ inbc
9810 PRINT "Turn on PLAY recorder"
9820 LOAD f$ + m$ DATA i$ ( )
9830 PRINT "Turn off PLAY recorder"
9840 LET ip = 1
9850 LET inbc = inbc + 1
9860 RETURN
```

```
9900 LET m$ = STR$ outbc
9910 PRINT "Turn on RECORD recorder"
9920 SAVE g$ + m$ DATA o$ ( )
9930 PRINT "Turn off RECORD recorder"
9940 LET op = 0
9950 LET outbc = outbc + 1
9960 RETURN
```

TESTING

Now what's needed is a little program to test that it all works. We don't want to have to key in huge piles of stuff, so we'll set up small buffers and get the program to generate its own file. The simplest thing is just to generate a sequence of ascending numbers.

This should do:

```
10 GO SUB initcfs
20 FOR n = 1 TO 100
30 LET r$ = STR$ n
40 GO SUB write
50 NEXT n
60 LET r$ = "cfsend"
70 GO SUB write
80 STOP
```

Run this. The first thing that happens is that *initcfs* asks you for an input filename. Of course, there isn't one because we haven't created a file yet—that's what we're about to do. So just give it some arbitrary name, "null" for instance. Now you're asked for the output filename, which you might call "test" or "fred" or whatever. Then, *initcfs* asks for the number of bytes per record. In this case, it's never more than 3 (when n = 100) but beware! "cfsend" occupies 6 bytes, so all records must be at least 6 bytes long. (If you don't like that, you can always use some special character. Watch it, though—control characters may have odd effects!) Finally, *initcfs* wants to know the number of records per block. Choose 20, for reasons which will become clear in a minute.

Now the beast does a bit of processing until it's filled the output buffer, when, of course, *putblock* will be called. So you see displayed on the screen the message:

"Turn on RECORD recorder"

Since the next thing is a SAVE command, you then get the usual prompt to switch on the recorder and then hit any key. So if you wished, you could dispense with line 9910. The only disadvantage with this is that the standard Sinclair prompt does not point out that the tape must be in RECORD mode. Incidentally, when you're handling one input and one output file you seem to require more than a natural number of hands. Things can be made slightly simpler by leaving the two recorders in PLAY and RECORD modes respectively, and controlling their movements using the PAUSE button (provided you have one, of course).

Meanwhile, back at the program, as soon as the block has been saved, you get a prompt to turn off the recorder. In this case, because the buffer is very rapidly filled, you almost immediately get another "Turn on RECORD recorder" message, so it's hardly worth the bother. All that happens if you don't turn off between block SAVEs is that you get slightly longer gaps on the tape between blocks.

Altogether, six blocks will be saved by the program, since exactly five are needed for the numbers 1 to 100, which means that a sixth is required just to hold "cfsend".

Now we need to check that the file has indeed been saved correctly. The simplest thing to do is to change lines 20, 30 and 40 like this:

```
20  GO SUB read
30  PRINT r$
40  GO TO 20
```

and RUN again, with the tape in the PLAY recorder. What should happen is that after *initcfs* has asked for the file details (this time it's the output file which is null, remember) the system prompts for the PLAY recorder to be turned on, *getblock* then reads 20 numbers which it passes to *read*, which in turn passes them to r$ one at a time, after which they're printed, and a prompt is issued for the recorder to be turned off. And that's what does happen, with a couple of additions. Firstly, of course, the Spectrum displays the filename of the file it's reading, so we should see:

```
character array: fred0
```

first time round.

SCROLLING PROBLEMS

This is indeed what happens, but there is a small fly in the ointment: the scrolling mechanism now gets in the way. The system prompts you with "scroll?" and when you respond affirmatively it immediately comes up with:

```
"Turn off PLAY recorder"
```

So if you're slow to re-enable scrolling, you could find the tape halfway through the next block before the program has started to read it. (This problem is particularly obvious with a block size of 20, which is why I chose it.) This isn't a total disaster because the Spectrum will only try to read the block it's *supposed* to be reading next, so you can always rewind a bit; but that *is* rather tedious. A better alternative is to disable paging altogether during calls to *getblock*.

Recall from Chapter 6 that there's a system variable called SCR-CT at 23692 which can be made to do this. It holds the number of lines (plus 1) that will be printed before the next "scroll?" prompt is issued. So we could set this to 255 (the largest possible) whenever *getblock* is called by adding the line:

```
9825  POKE 23692, 255
```

It must go *after* the LOAD, because that resets SCR-CT to 1.

So now every call to *getblock* allows 256 lines of output before a "scroll?" prompt will appear. Whether this is adequate will depend on the application. In this case it's more than enough, but it might be safer to include the statement in *read* as well as *getblock* since this routine is going to be called more often. Even then, the effect isn't absolutely guaranteed since it depends how much the user program is writing to the screen between reads, but under normal circumstances there should be no problem.

SO FAR SO GOOD . . .

We should pause for breath here, and review what's been going on. First note that *initcfs* has been used in two distinct ways:

1. to *create* a file to be written to;
2. to *open* a file which already exists, to be subsequently read.

So we could have written two separate routines instead of *initcfs*, called *create* and *open*, and there is indeed something to be said for this approach, since as we've seen, if we don't happen to want an input and an output file we have to allocate dummies to keep

initcfs happy. Secondly, the user is being asked to handle the termination of files himself. In other words he has to know that the file delimiter is "cfsend". We could have written another subroutine called *close* which would do the job automatically, so that lines 60 and 70 of our test program could be replaced by:

```
60  GO SUB close
```

and *close* would just be:

```
LET r$ = "cfsend"
GO SUB write
RETURN
```

MICRODRIVE

Now, if you look at the Spectrum keyboard, you'll find the keywords CLOSE # and OPEN #. These are the equivalent commands to those I've been discussing for files held on the microdrives. (There's no CREATE #, so OPEN # must do both jobs, just as *initcfs* does.) The equivalents for *read* and *write* are INPUT # and PRINT #. Everything else (i.e. the organization of file blocks and so on) is handled by the Spectrum operating system and so the file structure on the microdrives is just as transparent to the user as cfs is. Actually, the microdrive handling routines have to do a lot more than *read*, *write*, *getblock*, *putblock* and *initcfs* do, but the principles are similar.

AUTOMATIC CONTROL

There's a question which has probably been niggling you for some time: "Could we get the Spectrum to control the cassette motors automatically?".

The answer is "yes", and it isn't very difficult. You need recorders which have remote start jacks (most do). You also need a parallel I/O port and a couple of 5 volt low current relays. The relay contacts are used to complete the remote motor start circuits and their coils are driven from any convenient two bits of the port. Then, instead of printing messages, we simply POKE the port with a bit pattern (using BIN) which turns the appropriate line on or off. Appendix B gives hardware details. Unfortunately SAVE still automatically sends its prompt to switch on the recorder and waits for a key to be hit. So total automation is still tantalizingly just over the horizon.

USING FILES

Now, you may recall that the impetus for all this effort was the idea of writing a program to handle our record collection details, and that seems to have got lost in the welter of details about utility subroutines for handling the file system. But, of course, now that we've got them, it's going to make the main program a lot easier to write.

There are three basic functions of any file system:

1. Create the file from scratch.
2. Maintain it by doing necessary additions and deletions.
3. Search the file for some desired entry.

For simplicity we'll write these as separate programs, although it would be a simple matter to link them together via a menu (see our book *Machine Code and Better Basic*).

The create routine starts pretty straightforwardly. There'll be a call to *initcfs* in which the input file will be set to "null", the output file to "reccol" (say), the number of bytes per record to 336, and the number of records per block to something like 5 to allow a comfortable space for the program.

Now we hit the only serious problem in this routine: it is to set up each record in a convenient way for the user. For instance, we know that the "artist" field is 30 bytes long, but we don't want the user to have to key in "ABBA" followed by 26 spaces. So we'll

have a subroutine called *inrec* which handles the input of a single record in a user-friendly way.

The *create* program then looks like this:

```
100  GO SUB initcfs
110  GO SUB inrec
120  GO SUB write
130  INPUT "Any more? (y/n)"; q$
140  IF q$ = "y" THEN GO TO 110
150  GO SUB close        [assuming you've implemented close]
160  STOP
```

Now we can worry about *inrec*. Let's set up a string array, a$, which is to hold the record as it builds up. So if *inrec* is at 8000:

```
8000  DIM a$ (336)
```

Now prompt the user for the first required piece of data, and put it in the right place:

```
8010  INPUT "Artist"; a$ (TO 30)
```

Then

```
8020  INPUT "Date of purchase"; a$ (31 TO 36)
```

After that we want to handle 12 tracks in the same way:

```
8030  FOR t = 1 TO 12
8040  PRINT "track"; t
```

Now we want to write something like:

```
8060  INPUT "title"; a$ (begin TO end)
```

having worked out what "begin" and "end" are in line 8050.

Obviously "begin" and "end" change depending on what track we're on at the moment. Let's write down a short table of their values to give us an idea about the relationships involved.

Track (t)	Begin	End
1	37	56
2	62	81
3	87	106

So begin $= 37 + 25 * (t - 1)$

and end $= 56 + 25 * (t - 1)$

So:

```
8050  LET begin = 37 + 25 * (t - 1): LET end = 56 + 25 * (t - 1)
```

The track length goes from end + 1 to end + 5:

```
8070  INPUT "length"; a$ (end + 1 TO end + 5)
```

And then:

```
8080  NEXT t
```

Pass the result to r$:

```
8090  LET r$ = a$
```

and that's it:

```
8100  RETURN
```

MAINTENANCE

What about the maintenance program? Again, it will start with a call to *initcfs*, and, for the first time, we want to define both input and output files. The input file is "reccol", and we need to identify the output file as being an update on this; for instance, "reccola" would do. Subsequent updates might be called "reccolb", "reccolc" and so on, in the manner of car registrations. Or, you might prefer to provide date information, and call the file, say, rc982, for "record collection as at September '82". In any event, you need *some* formal system. Otherwise it's too easy to pick up the wrong file and modify the wrong information.

We'll keep life pretty simple to start with, and allow the user to make just one addition or one deletion from the file on one run of the program.

So we have:

```
100  GO SUB initcfs
110  INPUT "add or delete (a/d)?"; q$
120  IF q$ = "a" THEN GO SUB add: STOP
130  IF q$ = "d" THEN GO SUB delete: STOP
140  PRINT "Enter a or d"
150  GO TO 110
```

DELETE

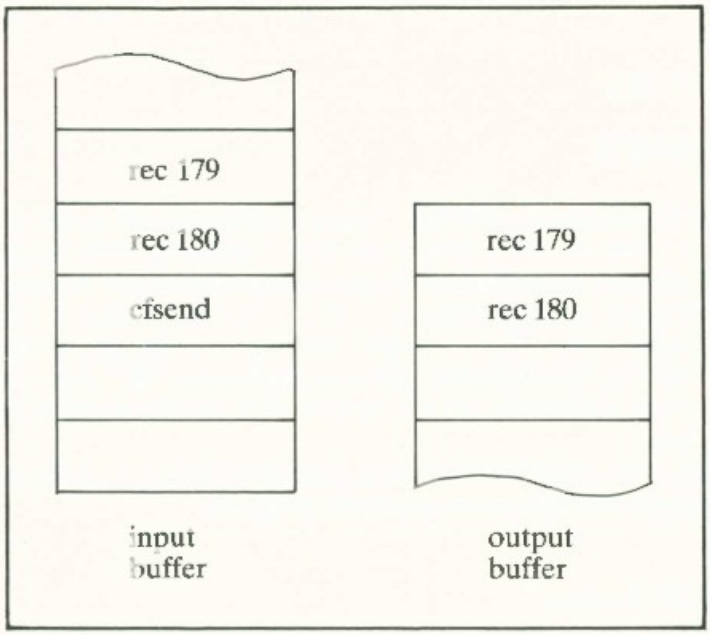
Now for the *delete* routine (because it's easiest). Suppose it's to run from 6000 onwards. We need to dimension a string array to hold the artist name and date of purchase to identify the record uniquely. Of course, this assumes you haven't bought two LPs by the same artist on the same day. This kind of ambiguity often causes problems in file design, and the usual way out is to add an extra field called a *key*, at the beginning of each record, which contains a number used only for this one record. A bank account number is an example of this. Anyway, assuming that we're OK, we'll need:

```
6000  DIM a$ (336)
6010  INPUT "Artist"; a$ (TO 30)
6020  INPUT "date purchased"; a$ (31 TO 36)
```

which sets up the required string in just the same way as the *inrec* routine we wrote just now. Fine; now all we have to do is to pull records in from the input file, see if they match the one we want to delete and, if not, shovel them out to the output file:

```
6030  GO SUB read
6040  IF r$ (TO 36) = a$ THEN GO TO 6030
6050  GO SUB write
6060  GO TO 6030
```

Simple enough? Unfortunately it's too simple by half. Let's think about what happens close to the end of the file.



We'll suppose we've just read record 180, found that it's not a candidate for deletion, and so it's been transferred to the output buffer with the *write* routine. Now *read* is called again and "cfsend" is found. *read* says that's beyond the end of the file, and it halts with an error message. Now you may say "That's no real problem, because we've finished at that stage anyway". But we haven't quite, because the output buffer still has records 179 and 180 in it which haven't been output, and there's no end-of-file marker on the output file, so mysterious things are guaranteed to happen if we try to read our newly created file. Incidentally, when this kind of thing occurs, the output buffer is said not to have been *flushed*, a picturesque piece of jargon.

There are a number of ways out of this hole that we—all right I—have just fallen into. Perhaps the simplest is to have *two* end-of-file markers, one for the benefit of cfs (cfsend) and one for the user's benefit. Let's use "}}".

So we need to rewrite the *close* routine to generate both end-of-file markers:

```
9740  LET r$ = "}}"
9750  GO SUB write
9760  LET r$ = "cfsend"
9770  GO SUB write
9780  RETURN
```

(Of course, we have to identify *close* for BASIC's benefit; we could edit line 3:

```
3  LET putblock = 9900: LET close = 9740        )
```

Now we can modify the *delete* routine to test for the user's end-of-file marker, and close the output file when it's found:

```
6000  DIM a$ (336)
6010  INPUT "Artist"; a$ (1 TO 30)
6020  INPUT "date purchased"; a$ (31 TO 36)
```

```
6030  GO SUB read
6040  IF r$ (TO 2) = "}}" THEN GO SUB close: RETURN
6050  IF r$ (TO 36) = a$ THEN GO TO 6030
6060  GO SUB write
6070  GO TO 6030
```

Notice that, in lines 6040 and 6050, only the first 2 and first 36 bytes of r$ respectively are used for the comparisons. This is important, and easy to forget if you're not careful. The point is that r$ is 336 bytes long and, even if it's empty, "}}" isn't the same thing as "}} + 334 spaces" as far as BASIC is concerned.

ADD

Now for the addition routine. We haven't said anything yet about the order in which the records are stored on the tape. Let's assume for the minute that they are alphabetical by artist name, but where there is more than one record by the same artist, the order is undefined. So our problem can be stated broadly like this:

1. Input addition.
2. Read a record.
3. If it's end of file then close file: RETURN.
4. If artist name (record) is before artist name (addition) then write record: go to 2.
5. Write addition.
6. Write record.
7. Go to 2.

In other words we do exactly what we would do with a card index: make out a new card, go through the file until we find the right alphabetical niche, pop it in. The only difference is that, in the computer case, we're *physically* moving records from one place to another (input to output) every time we read them. It's as if we had two card index trays, and the rules were that, as soon as we've read one in the "input" tray it has to be transferred to the "output" tray.

There is, however, a subtle bug in our algorithm. Suppose that the last record in our collection is by Suzi Quatro, and we wish to add a Yardbirds album. The procedure works through the file, transferring records as it goes, finally dealing with the Suzi Quatro LP. Then it sees an end of file marker, so it closes the file, leaving the addition details still in memory! So we need to modify step 3:

3. If it's end of file then check: close file: end.

check will be a routine which checks to see if the additional record has been written out yet, and, if it hasn't, *check* will write it.

Let's write the *add* routine from 5000 on. First we have to input the additional record. But we've already got a routine which accepts a full set of record details: *inrec*. So call that. No point in reinventing the wheel . . .

```
5000  GO SUB inrec: LET transfer = 0
5010  GO SUB read
5020  IF r$ (TO 2) = "}}" THEN GO SUB check: GO SUB close: RETURN
5030  IF r$ (TO 30) < a$ (TO 30) THEN GO SUB write: GO TO 5010
```

A couple of things need some explanation before we get too far. First, the "LET transfer = 0" in line 5000. *check* is going to need some way of knowing whether or not the additional record has been transferred to the output file. As soon as the addition has been accepted, we set "transfer" to zero. When the addition is transferred to the output file, we'll set transfer to 1. So *check* only has to test "transfer" to see if it's zero. If it is, the addition has yet to be output.

Second, line 5030 compares two strings; r$, which has just been returned by *read*, and a$ which was set up by *inrec*. (r$ was also set by *inrec*, but this was immediately overwritten by *read*.) The use of the "less then" symbol, here, has the meaning "alphabetically precedes", so dealing with strings in alphabetical order presents no problem.

To continue:

```
5040  LET b$ = r$                [save last record read]
5050  LET r$ = a$                [transfer addition to r$ for writing . . .
5060  GO SUB write              and write it]
5070  LET transfer = 1           [signal that it's been written]
5080  LET r$ = b$               [put last record read back in r$ . . .
5090  GO SUB write              and write it]
5100  GO TO 5010
```

Now, finally, to write *check* (from 5200 on):

```
5200  IF transfer = 1 THEN RETURN   [no action necessary because addition
                                    has been output]
5210  LET r$ = a$                   [transfer addition to r$
5220  GO SUB write                  and write it]
5230  RETURN
```

Project

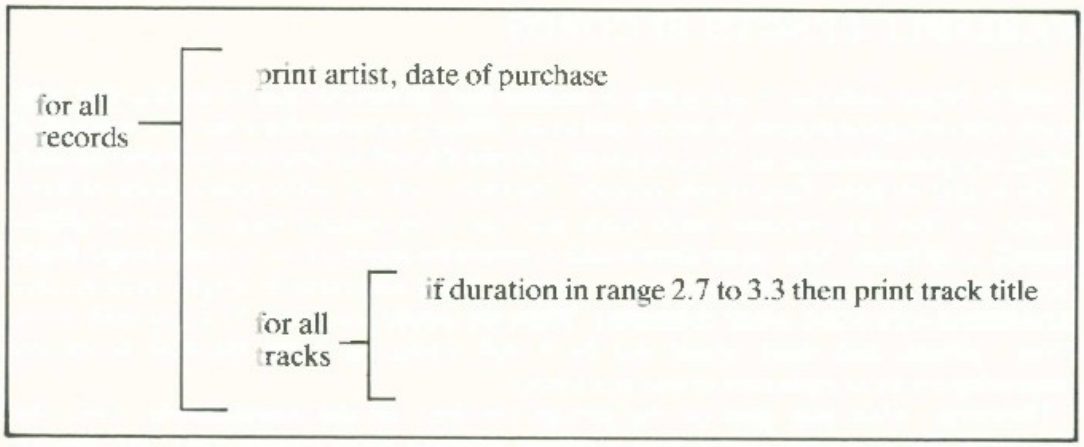
All the foregoing assumes that only one alteration to the file is to be made. If you go out on a mad record-buying spree and get 15 new records, and then donate 4 of your old ones to Oxfam, you'll have to run the file maintenance program 19 times to get the file up to date!

Try to write a maintenance program which will accept all alterations at the beginning. You'll have to read all the additions into an array and all the deletions into another. Then you'll have to compare each record read in with each of the entries in the two arrays to decide whether to delete or copy the record, or make an insertion. Note that each addition will need its own transfer flag (i.e. there'll have to be a transfer *array* rather than a single variable). Note also that the order in which the additions are entered is unimportant, because the program will search for any addition in the array which should be inserted. Don't forget to allow for the fact that more than one insertion may be necessary between two existing records. It's an interesting thought that if you create the file with just a couple of records, and then use this maintenance program to add the rest, the file will automatically be generated in alphabetical order!

SEARCHING THE FILE

Having spent a considerable time creating this file, we ought to find some use for it. Suppose that we want to make up a tape of background music for a party. We want a range of tracks all about 3 minutes long. So what we would like is a list of tracks which last between 2.7 and 3.3 minutes, for instance.

So we want a program which does this:



This is pretty simple:

```
100  GO SUB initcfs
110  GO SUB read
120  IF r$ (TO 2) = "}" THEN STOP
130  PRINT r$ (TO 30)
140  PRINT r$ (31 TO 36)
150  FOR t = 1 TO 12
160  LET begin = 57 + (t - 1) * 25
170  LET d = VAL r$ (begin TO begin + 4)
180  IF d < 2.7 OR d > 3.3 THEN GO TO 200
190  PRINT r$ (begin -20 TO begin -1)
200  NEXT t
210  GO TO 110
```

Notice that, since we're only *reading* a file, there's no need to do a *close* operation when the end of it is detected at line 120.

The only tricky bit here is evaluating the duration of each track as a number. First we have to find the relevant portion of r$ (line 160). Then it's necessary to create a numeric value from this string (line 170) so that comparisons between numbers can be done in line 180. Of course, the 2.7 and 3.3 could just as easily be variables entered at the beginning of the program. That would allow other questions to be asked, like:

"List all tracks at least 4.5 minutes long" (by making the low boundary 4.5 and the high one something unreasonably large: 9999, for instance)

Or:

"List all tracks exactly 3 minutes long" (by making both boundaries 3)

Identifying other specific features of the file is just as easy. Try the following:

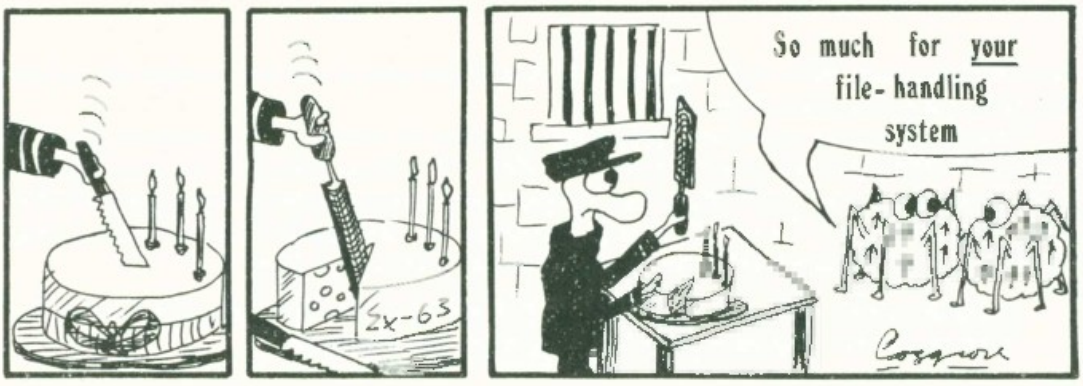
1. List all tracks by a given artist.
2. As for (1), but purchased between two given dates. (This is slightly more difficult than it looks, because of the date comparisons.)
3. List all tracks whose titles include the word "rock", or any other word entered in an INPUT statement.

VARIABLE LENGTH RECORDS

I said to begin with that I was going to assume that all records had 12 tracks, that each track title occupied exactly 20 bytes, and so on. What I've created is a file of records all of which are guaranteed to be 336 bytes long. I doubt if it will surprise you to learn that such a file is said to have *fixed length* records. Further, each record contains fields of fixed length, so that, for instance, each track title has to be padded with spaces (or abbreviated) to 20 bytes. Now, in an ideal world, it would be nice to allow variable length fields (terminating each with a field delimiter of some kind) and variable length records (also terminated with some other delimiter). Then we would not waste 27 bytes with every "Yes" album, and there would not be 8 null tracks for every classical symphony (assuming we treat each movement as a track).

However, what you gain on the swings you lose on the roundabouts. First, the programs for extracting fields from a record become more complicated. We have to search, byte by byte, looking for field delimiters, and we can no longer talk about "the fifth field"—there may not be one. Second, the utility routines for reading and writing, and handling file blocks become rather hairy. After all, in our fixed system it was easy to define a block as (effectively) a two-dimensional array, which was number of bytes per record × number of records per block. If the number of bytes per record changes, we can no longer think in that comfortable way. On the other hand, if we keep the buffers a fixed size then the number of records per block changes, depending on the record sizes!

I don't want to give the impression that handling variable length records is beyond the wit of man; only that it may not be as sensible as it appears at first sight. The point is that as we increase the complexity of the utility programs, we also increase their size, which correspondingly decreases the quantity of main memory left for the user program and the file buffers. Also, unless a lot of space is being wasted in the fixed length format, the introduction of extra delimiters may use up most of the file space being saved. Often, careful choice of field lengths in a fixed length system provides a perfectly adequate answer. We should also think carefully about the file contents. Do we *really* want classical and pop records on the same file? Wouldn't it be better to have two files, so that each can be defined in a sensible way, without having to cater for the features of the other?



In any event, we should never forget that we are trading cassette backing store, which is cheap, against RAM, which isn't. In fact, 5 Kilobytes (KB) of data will cost you around 1p on cassette tape. (It's easy to calculate—just remember that, at a transfer rate of around 200 bytes/second it will take 25 seconds to save 5 KB, and then work out 25 ÷ (60 × 60) of whatever you paid for your last C60. You'll probably find my figure is pessimistic.) 5 KB in main memory is, even today, as memory prices fall like lead balloons, still going to cost you around £10. No contest, I think.

If the above treatise sounds like the plausible ramblings of somebody who doesn't want to have to think too hard unless he's actually forced into it, then you've caught my mood exactly. Computers are supposed to make life easier, not more difficult.

A modest proposal

There's one niggling feature of cfs as it stands: *initcfs* asks the user for details of the record and block sizes, even for files which already exist.

Modify cfs so that a "header" block is written first to any output file, containing the record and block size details. Then, if an input file name is specified to *initcfs*, this routine no longer asks for the buffer size details but reads them from the header block of the input file.

(For a *less* modest proposal, see Chapter 17.)

There are lies, damned lies, and computer print-outs.

10 Statistics made Simple

National Wealth Servers Ltd. (NWS) employed 24 staff at £5Ø per week. The Managing Director got £1Ø4,ØØØ per year. When the Regional Organization of Wealth Serving Employees (ROWSE) went on strike for more pay, the management took out a full-page ad in the *Gardener* pointing out to the public that the *average* wage was £128 per week. The Managing Director was quoted as saying "These layabouts should get back to work at once and stop whining: £128 is a perfectly fair weekly wage."

Statistics can be used, and misused. Averages *can* be a fair measure of the "norm", the "typical value"—sometimes. They can also be distorted by the odd exception that is wildly out of line, as here. The *calculation* is correct:

$$\text{Average} = \frac{\text{Total weekly wage}}{\text{Number employed}}$$

$$= \frac{24 \times 5Ø + 2ØØØ}{25}$$

$$= 24 \times 2 + 8Ø = 128$$

But the *interpretation*—"Most workers get about £128"—is not.

Which goes to show that an understanding of basic statistics is well worth having. In this case ROWSE published its own ad, pointing out that a more appropriate statistic is the *mode*, the commonest wage, here £5Ø. Recognizing the impeccable logic of this argument, NWS sacked its Managing Director, handing over control to a committee of employees, and upped their wages to £128 per week, leaving a saving of £4Ø56 per annum—which, over the next ten years, almost defrayed the cost of the original advertising campaign.

Of course, it doesn't always happen that way . . .

PRESENTATION OF DATA: HISTOGRAMS

This isn't the place to teach you statistical theory. What I'm going to do is write out some programs that let you explore statistical ideas without having to go into their inner workings. That way you can get some feel for what they mean in practice. And an important part of statistics involves the way that data are presented.

The Spectrum being a visual beast, I'll concentrate on two standard types of graphic display: the *histogram* and the *pie chart*.

A histogram displays how often a given "event" has occurred. For example, suppose I throw a die twenty-eight times, with the results:

1 is thrown 3 times
2 is thrown 6 times
3 is thrown 5 times
4 is thrown 5 times
5 is thrown 4 times
6 is thrown 5 times

Then a histogram display will have six vertical bars, labelled 1–6, of the corresponding heights. So bar 1 has height 3, bar 2 has height 6, and so on; See Figure 10.1.

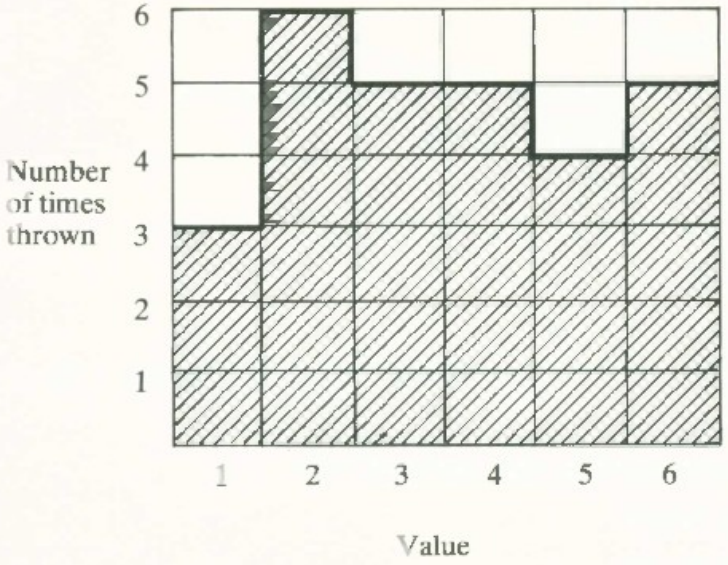


Figure 10.1 Histogram showing number of times a given value occurred when throwing a die.

The average, or *mean* value of the throw is given by

$$\frac{1\times 3+2\times 6+3\times 5+4\times 5+5\times 4+6\times 5}{28}=3.57$$

Notice how each score is multiplied by the size of the bar above it. Now a fair die, in the long run, should give roughly equal numbers of occurrences for each of the values 1–6, with a mean value

$$\frac{1+2+3+4+5+6}{6}=3.5$$

So this particular statistic agrees pretty well with theory. Note, however, that the commonest value in the experiment—the *mode*—is the number 2, which occurs 6 times.

The mode tells you where the highest point of the histogram is. The mean tells you whereabouts to draw a vertical line so that the total area would balance there. These need not be the same; and the mean only gives an idea of the "typical" value if the numbers aren't too spread out. Here they are *very* spread out.

I'll talk about how to measure the degree of spreadoutness later on: for the moment all I really want is the idea of a histogram. To get you used to that, here's a program that displays a histogram for throws of a die, which you input as you throw. It also says what the mean value thrown is at any stage.

```
10  LET init = 500
20  LET valin = 1000
30  LET hist = 1500
40  LET mean = 2000
50  LET wtot = 0: LET num = 0
100 DIM d (6)
```

```
200  GO SUB init
210  GO SUB valin
220  GO SUB hist
230  GO SUB mean
240  GO TO 210
500  REM init
510  PLOT 103, 175: DRAW 0, -160: DRAW 48, 0
520  PRINT AT 21, 13; "123456"
530  FOR i = 1 TO 6
540  PLOT 99 + 8 * i, 15: DRAW 0, -5
550  NEXT i
560  FOR i = 0 TO 20
570  PLOT 103, 15 + 8 * i: DRAW -5 -5 * (i = 10 OR i = 20), 0
580  NEXT i
590  PRINT AT 0, 9; "20"; AT 10, 9; "10"
600  RETURN
1000 REM valin
1010 IF INKEY$ < > " " THEN GO TO 1010
1020 IF INKEY$ = " " THEN GO TO 1020
1030 LET c = CODE INKEY$ - 48
1040 IF c < 0 OR c > 6 THEN GO TO valin
1050 IF c = 0 THEN STOP
1060 LET d (c) = d (c) + 1
1070 RETURN
1500 REM hist
1510 IF d (c) = 21 THEN INPUT "No room for display"; x$: STOP
1520 PRINT AT 20 - d (c), c + 12; PAPER c; "□"
1530 RETURN
2000 REM mean
2010 LET num = num + 1: LET wtot = wtot + c
2020 LET a = .01 * INT (100 * wtot/num)
2030 PRINT AT 2, 22; "Mean ="
2040 PRINT AT 4, 23; a; "□ □ □"
2050 RETURN
```

Type this in and RUN. Take a die, and throw it: hit the corresponding key. (That is, if you throw a "4", hit key 4, and so on. No need to press ENTER.) A coloured histogram builds up: for each number 1–6 it shows how many times that number has been thrown. The mean is printed out too, as you go. If any number gets thrown more than 20 times the program halts with a message (actually you have to hit a key to get the halt). To terminate before that, hit key 0.

You'll easily see how this works. The main program is in lines 200–240: *init* sets up the axes and scales: *valin* reads the key; *hist* plots the chart; *mean* works out the mean and prints it.

The INT business in line 2020 is just a way to ensure that only two decimal places (or fewer) are printed. It's a useful trick. (For 3 decimal places, use LET a = .001 * INT (1000 * wtot/num) and so on, with an extra zero in both slots for each extra decimal place required.)

PRESENTATION OF DATA: PIE CHARTS

Aptly named, these show how the cake is divided between different recipients . . . well, fairly aptly named. A circle is sliced into pieces, with bigger shares represented by a bigger slice. You know the kind of thing.

The next program is an automatic pie-chart slicer. It accepts as input a series of named items (of expenditure, say) and produces a pie chart. It works best with 10 items or fewer, and preferably no item should amount to less than 3% of the total. It *works* even if these criteria are not met, but pie charts themselves aren't very useful if they have too many slices, or slices so thin you can't see them.

```
200  REM data in
210  INPUT "Number of items?"; n
220  DIM i$ (n, 9): DIM v (n)
230  DIM a (n + 1)
240  FOR i = 1 TO n
250  INPUT "Name of item?"; i$ (i)
260  INPUT "Value of item?"; v (i)
270  PRINT AT i, 5; i$ (i), v (i); "□ □ □"
280  INPUT "Is this correct? y/n"; q$
290  IF q$ = "n" THEN GO TO 250
300  LPRINT i$ (i), v (i): REM only type this line if you have a printer
310  NEXT i
320  LET tot = 0
330  FOR i = 1 TO n
340  LET tot = tot + v (i)
350  NEXT i
500  REM piechart
510  CLS: CIRCLE 84, 84, 75
520  LET ang = 0
530  FOR i = 1 TO n
540  LET ang = ang + v (i) * 2 * PI / tot
550  LET a (i + 1) = ang
560  PLOT 84, 84
570  DRAW 75 * COS ang, 75 * SIN ang
600  REM label
610  LET u = .5 * (a (i) + a (i + 1) )
```

```
620  PRINT AT 11 - 7 * SIN u, 10 + 7 * COS u; i
630  NEXT i
700  REM table
710  FOR i = 1 TO n
720  PRINT AT i, 21; i; ":"; i$ (i)
730  NEXT i
740  COPY: REM if you have a printer
```

Lines 300 and 740 should be left out unless you have a printer and actually want a record of the results. Lines 280 and 290 provide a way of correcting mistakes if you catch them straight away: if this feature annoys you, delete them. (Incidentally, you don't need to hit "y" for yes: any key except "n" will work. ENTER is the obvious one.)

For example, key in the following figures, which give the Gross National Product (per head) of the EEC countries in 1978.

Number of items: 9 Name of item	Value of item	y/n
Belgium	129	n to correct
Denmark	144	"
France	116	"
Germany	137	"
Holland	123	"
Ireland	50	"
Italy	60	"
Luxembourg	126	"
UK	73	"

Figure 10.2 shows the resulting pie chart. Experiment using other sets of figures, real or imaginary.

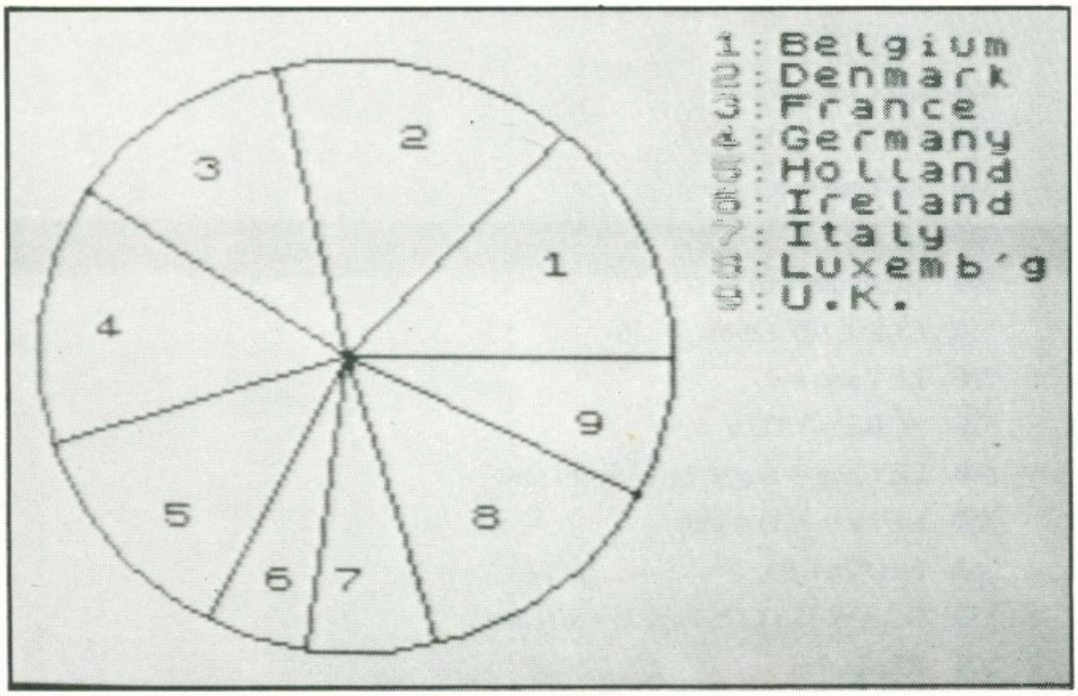


Figure 10.2 How the EEC pie is divided . . .

MEANS, MODES AND ALL THAT

I've already explained means and modes. (There's another creature called a median, but let's not get confusing.) The mean is an "average" value, the mode a (there may be more than one) "commonest" value.

I also mentioned that the mean is reasonably "typical" provided the data aren't too spread out. To measure the spread, statisticians use a gadget called the *standard deviation*. It's the root mean square deviation from the mean, if you must know.

They also have a favourite curve, called a *normal curve*, used to approximate histograms: it is chosen to have the same mean and standard deviation, and it's shaped like a camel's hump.

Rather than present you with a mass of mathematics (you can always look the stuff up in a statistics text—any will do) I've written a program which lets you produce your own histogram, and then tells you the mean, the (smallest mode, the standard deviation, and plots out the normal curve approximation as a bonus.

If you (or your child) are studying statistics at school, this program will help you (her, him) get a good feel for what these things represent.

```
10  LET init = 500
20  LET draw = 1000
30  LET stats = 2000
40  LET normal = 3000
50  DIM a (24)
60  LET step = 3
100 GO SUB init
110 GO SUB draw
120 GO SUB stats
130 GO SUB normal
140 STOP
500 REM init
510 CLS
520 PLOT 15, 165
530 DRAW 0, -150
540 DRAW 240, 0
550 FOR t = 1 TO 24
560 PLOT 15 + 10 * t, 15: DRAW 0, -3 -3 * (t = 10 OR t = 20)
570 NEXT t
580 FOR t = 1 TO 15
590 PLOT 15, 15 + 10 * t: DRAW -3 -3 * (t = 10), 0
600 NEXT t
610 PRINT OVER 1; AT 21, 0; "[13 spaces] 10 [11 spaces] 20"
620 PRINT AT 7, 0; "1"
630 PRINT AT 8, 0; "0"
640 RETURN
1000 REM draw
1005 LET h = 15
```

```
1010  FOR i = 1 TO 24
1020  IF INKEY$ < > " " THEN GO TO 1020
1030  IF INKEY$ = " " THEN GO TO 1030
1040  LET k$ = INKEY$
1045  LET h = h + 10 * (CODE k$ - 53)
1050  IF h < 15 THEN LET h = 15
1055  IF h > 165 THEN LET h = 165
1060  LET a (i) = (h - 15) / 10
1070  PLOT 15 + 10 * i - 10, 15: DRAW 0, h - 15:
      DRAW 10, 0: DRAW 0, 15 - h
1080  NEXT i
1090  RETURN
2000  REM stats
2005  LET tot = 0: LET norm = 0: LET m = 0: LET mv = 0
2010  FOR i = 1 TO 24
2015  IF a (i) > mv THEN LET mv = a (i): LET m = i
2020  LET tot = tot + i * a (i)
2025  LET norm = norm + a (i)
2030  NEXT i
2040  LET av = tot/norm
2050  PRINT AT 0, 3; "Mean = □"; av
2055  PRINT AT 1, 3; "Mode = □"; m
2060  LET std = 0
2070  FOR i = 1 TO 24
2080  LET std = std + a (i) * (i -av) * (i - av)
2090  NEXT i
2100  LET std = SQR (std/norm)
2105  LET std = .01 * INT (100 * std)
2110  PRINT AT 2, 3; "Standard deviation = □"; std
2120  RETURN
3000  REM normal
3010  FOR i = 0 TO 240 STEP step
3020  LET y = EXP (- (.5 + i / 10 - av) * (.5 + i / 10 - av) / (2 * std * std) )
      / (std * SQR (2 * PI) )
3030  LET y = y * tot
3035  IF y > = 150 THEN GO TO 3060
3040  IF i = 0 THEN PLOT i + 15, y + 15
3050  IF i > 0 THEN DRAW step, y + 15 - PEEK 23678
3060  NEXT i
3070  RETURN
```

USING THE PROGRAM

The program is designed to make it very easy to set up trial histograms; and in consequence the way these are entered is fiendishly unorthodox and infuriating until you get used to it. It goes like this. The bars at positions 1–24 are entered in turn. Initially the height is 0 (and is always between 0 and 16). The *next* bar-height is 5 − k *larger* if the number key k in the top row is hit. That is, the number-keys control the *change* in height of the bar from one column to the next, like this:

Key	*Effect on height of bar*
1	4 smaller
2	3 smaller
3	2 smaller
4	1 smaller
5	same
6	1 higher
7	2 higher
8	3 higher
9	4 higher

For instance, RUN and then hit (no need for ENTER) the keys

5 5 6 6 6 7 8 5 2 3 4 4 4 5 5 5 5 5 5 5 5 5 5 5

to get the result of Figure 10.3.

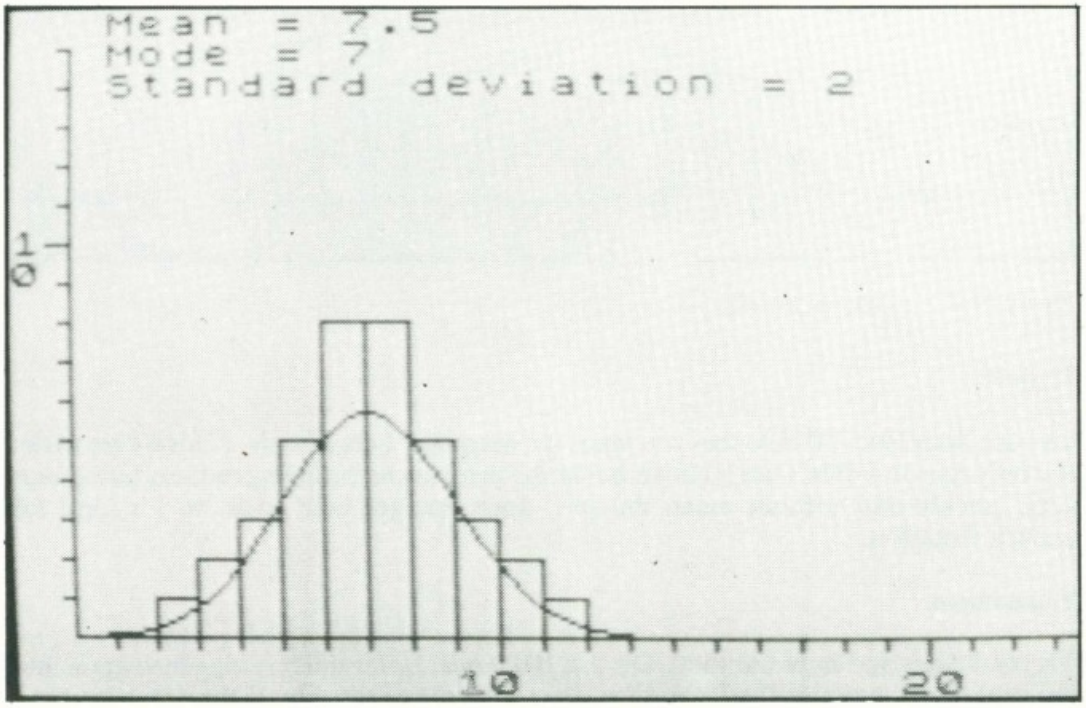


Figure 10.3 Normal curve approximating a histogram fairly well . . .

Invent your own histograms, and check that:

1. The mean is a reasonable "average" value. If you cut the histogram from sheet metal it would balance at the mean.
2. The mode is the first "peak" value. (The other places at this height are modes too, but the program doesn't notice other peaks. It would be easy to change this.)

3. Histograms that are more "spread out", like
   5 5 6 6 6 7 8 5 5 5 5 5 2 3 4 4 4 5 5 5 5 5 5 5
   (inputs) have larger standard deviation; histograms that are "bunched" like
   5 5 5 5 5 5 9 9 1 1 5 5 5 5 5 5 5 5 5 5 5 5 5 5
   have smaller standard deviation.
4. The normal curve is a reasonable approximation to histograms that have a single peak, are not too spread out, and are roughly symmetric . . .

But it can be quite bad for other histograms: for example when there are two humps like
5 5 5 7 7 7 9 1 1 4 4 5 5 5 8 8 3 3 3 5 5 5 5 5
which gives the result of Figure 10.4.

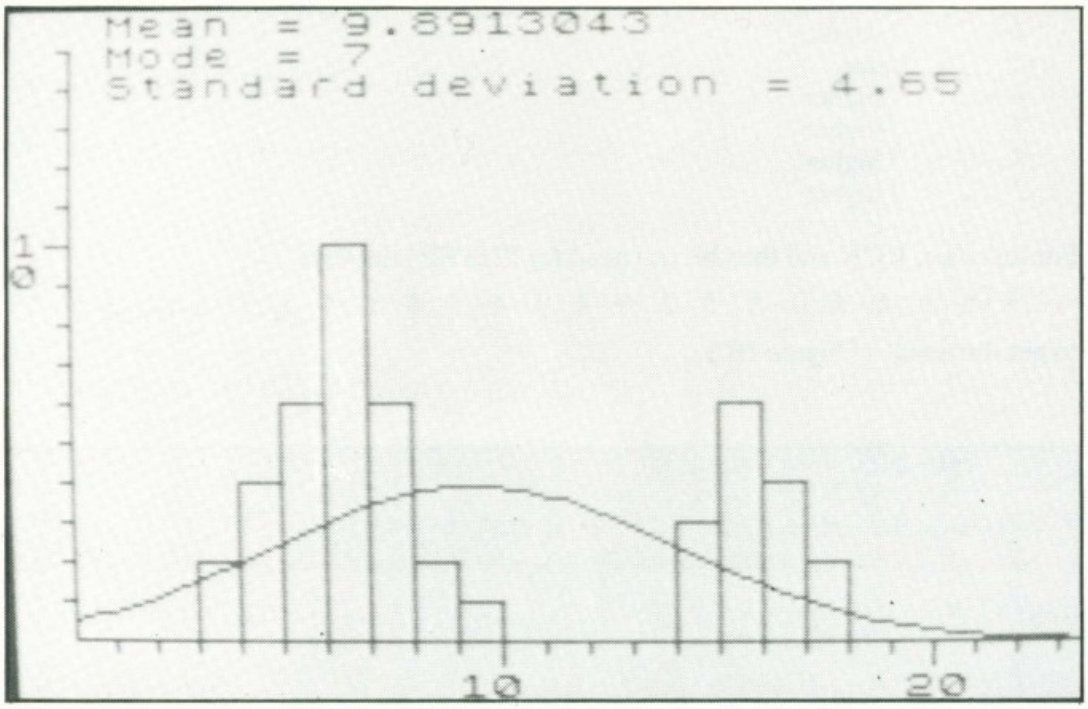


Figure 10.4 . . . and very badly.

Project

Change lines 1045–1060 so that you input the heights of bars *directly*. (This is easy to do: the only reason I didn't use it above is that the program as it stands produces histograms very quickly and without much thought, once you get used to it, so it's ideal for experimentation.)

Experiment

Throw 4 dice, and note the total. Do this 100 times. Enter the resulting histogram into the program above (via the Project) and see what the results are. Is the normal curve a good fit, or not?

Simulation

Use the Spectrum's random numbers to simulate throwing 4 dice: repeat the analysis.

A little attention to detail can work wonders—such as translating from Spectrumese into algebra . . .

11 Improving the Display

Many programs can be made much more attractive by setting up user-defined characters to get a more accurate representation of the desired effect. Here I'll consider one project along these lines, which takes *polynomials* in several variables x, y and z, in the form that the Spectrum uses, and makes them look like ordinary algebra. (If you don't like algebra, bear with me: the computing is the main thing.)

Ordinary algebra writes polynomials like this:

$ax^2 + bx + c$

$2x^3 + 5y^7$

$a^3 + b^3 + c^3 - 3abc$

But the Spectrum uses "*" for multiplication, instead of just writing the symbols next to each other; and it uses "↑" rather than raising symbols off the line, like this:

a * x ↑ 2 + b * x + c

2 * x ↑ 3 + 5 * y ↑ 7

a ↑ 3 + b ↑ 3 + c ↑ 3 − 3 * a * b * c

The first step is to develop user-defined characters for the raised exponents 1, 2, . . ., 9, 0. Figure 11.1 shows a possible layout. Enter these as graphics characters "a"–"j" in the usual way (*Easy Programming*, p. 49).

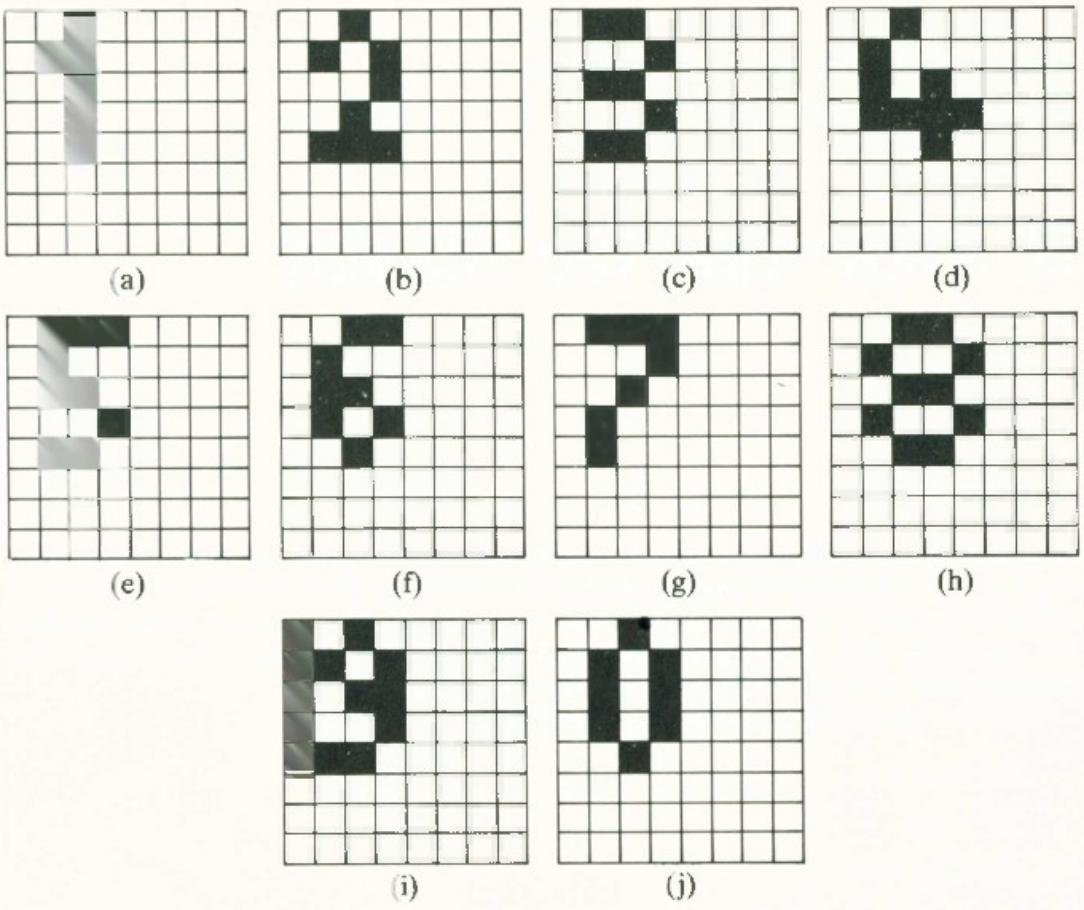


Figure 11.1 Graphics data for exponent characters.

Next, type in the following program. It accepts an expression in Spectrumese, and runs through it performing three tasks:

1. Strip out all *'s.
2. Turn all *numbers* after ↑ into user-defined graphic exponents.
3. Strip out the ↑'s once (2) has been done.

```
  5  LET pow = 200
  6  LET prod = 600
  7  LET print = 1000
 10  INPUT "Expression to be tidied?"; e$
 20  LET l = LEN e$
 30  LET f$ = " "
 40  LET i = 1
 50  IF e$ (i) = "↑" THEN GO TO pow
 60  IF e$ (i) = "*" THEN GO TO prod
 70  LET f$ = f$ + e$ (i)
 80  LET i = i + 1
 90  IF i > l THEN GO TO print
100  GO TO 50
```

```
200  REM powers
210  LET j = i + 1
212  IF j > l THEN GO TO print
215  LET c = CODE e$ (j)
220  IF c < 48 OR c > 57 THEN GO TO 500
230  LET f$ = f$ + CHR$ (c + 95 + 10 * (c = 48) )
240  LET j = j + 1
245  IF j > l THEN GO TO print
250  GO TO 212
500  LET i = j
510  GO TO 50
600  REM products
610  LET i = i + 1
620  GO TO 50
1000 REM print result
1010 PRINT "Old: □"; e$ ' ' "New: □"; f$
```

TESTING AND SAVING

Try the expressions listed above, Spectrum version, as the inputs e$; check that the right results appear. Now try some other expressions like:

```
2 * x * y * z − 17 * a ↑ 552
m ↑ 77 + n ↑ 88
a* b * c * d * e * f * g + 45 * h ↑ 999 + 32
```

and so forth.

The program won't cope with *everything*: see what it does to 2 * 2, for example! But it illustrates the idea.

To SAVE this in useful form, we must do a little more work, because user-defined graphics don't save automatically. First we must work out where they are.

In the 16K Spectrum, they start at address 32600. But you may have changed this (via the system variable UDG, which lives in addresses 23675–6). So you may prefer to work the address out as

```
PEEK 23675 + 256 * PEEK 23676
```

However, there's an easier way, because the graphics character "a" has to start the UDG-area. So

```
USR "a"
```

will work. (PRINT these, and see.)

Wind your tape to the place you want to start, and add a further line in preparation for what comes next, namely

```
1  LOAD "exp" CODE USR "a", 80
```

Now save the entire program, using

```
SAVE "algebra" LINE 1
```

for an automatic start on line 1.

You haven't finished yet: now use byte storage to save the graphics:

SAVE "exp" CODE USR "a", 8Ø

This takes the 8Ø bytes starting at USR "a", the user-defined graphics area, and saves them on tape. The 8Ø is there because each character takes 8 bytes, and we have 1Ø characters, so the total is 8 * 1Ø = 8Ø. Note that the added line 1 reverses this procedure to LOAD these bytes back in again.

If you've got both safely on tape, rewind to the starting position, and type

LOAD "algebra"

You'll get the usual buzzes and beeps and red/blue/yellow stripes and so forth, and a message

Program: algebra

Leave the tape running. "Algebra" will automatically start at line 1; and this automatically loads the user-defined graphics back in. There will be more buzzes and beeps and stripes, and the message

Bytes: exp

after which "algebra" will go on to its next line, and continue to run as usual. Stop the tape, and away you go.

This is an example of *chaining* taped programs together, making use of the fact that LOAD commands can be written into a program. For another example, see Chapter 15: Changing the Character Set.

Projects

1. Modify "algebra" so that it works out any products of *numbers*, such as 23 * 45, rather than just sticking them together to give 2345 (which is wrong) as the current version will do.
2. Modify it to change all expressions like x * x into x^2, or x * x * x and x * x^2 into x^3, and so on.
3. Modify it to *sort* variables into alphabetical order, so that abcbab becomes aabbbc or (better) a^2b^3c.
4. Modify it to remove any term that gets multiplied by Ø.
5. Devise an expression which your best-modified version gets wrong; and modify it to deal with *that*.
6. GO TO 5.

There are programs that do things in their own right, and there are programs that help you write other programs. The latter are called utilities. *For example:*

12 Line Renumbering

Tidying up line numbers can be a terrible chore—so much so that on the whole, nobody does . . . *unless* they have a utility program to do it for them. You can buy very fancy line-renumbering programs; or you can write your own, thereby saving anything between £5 and £10, possibly at the price of having something less versatile.

The program below is a compromise. It's written on the premise that this isn't a Machine Code book, so the program has to be in BASIC; and since BASIC is slow, the program has to be quick enough for actual use. That, in turn, means it has to be fairly rudimentary. Specifically, it *only* renumbers the lines: it does *not* automatically renumber GO TO or GO SUB numbers. I'll say a bit more on this, after listing the program.

LINE NUMBERS IN PROGRAMS

In common with most utilities, this one requires some actual knowledge of what goes on inside the Spectrum when the buttons are pushed. You'll remember (*Easy Programming*, p. 93) that the program itself is stored in RAM as a series of character-bytes, beginning at the address held in the system variable PROG and ending immediately before the address held by VARS. Consulting the *Manual*, you'll discover that these addresses can be found using the commands:

PROG: PEEK 23635 + 256 * PEEK 23636

VARS: PEEK 23627 + 256 * PEEK 23628

You'll also find that each program line is stored in the format:

NS	NJ	LJ	LS	code for line	ENTER

where NS and NJ are the senior and junior bytes of the line-number, and LJ and LS are the junior and senior bytes of the number of characters in the line altogether.

Specifically, if the line-number is n, then we have

NS = INT (n/256)

NJ = n − 256 * NS

and similarly for LJ and LS. So for line 700, for example, you get:

NS = INT (700 / 256) = 2

NJ = 700 − 256 * 2 = 188

and these are the first two bytes in the section of RAM that stores line 700.

By changing NJ, say to 198, we can fool the Operating System into thinking that this line is actually line 710 (= 2 * 256 + 198). This suggests how to renumber the lines:

Run along the program area, PEEKing to find occurrences of the ENTER character (whose code is 13). Having found one, we know that the next two bytes are a line-number. POKE these to the desired new value.

FIRST ATTEMPT

If you write a routine to do just this, you'll find that there are snags. First, it fails to renumber the first line, because that does not follow an ENTER character. Second (and more obscurely) if either of bytes LS, LJ happens to be 13, you'll get trouble.

These faults are easily remedied: renumber the first line as a matter of course; and jump over LJ and LS.

Assuming for the moment that you want the new numbers to start at 10 and go up in 10s, this leads to a piece of code along these lines:

```
1000  LET prog = PEEK 23635 + 256 * PEEK 23636
1010  LET vars = PEEK 23627 + 256 * PEEK 23628
1020  LET ns = 0: LET nj = 10
1030  FOR i = prog TO vars - 1
1040  IF i = prog THEN POKE i, ns: POKE i + 1, nj:
      LET i = i + 4: LET nj = nj + 10: IF nj >= 256
      THEN LET nj = nj - 256: LET ns = ns + 1
1050  IF PEEK i = 13 THEN POKE i + 1, ns: POKE i + 2, nj:
      LET i = i + 5: LET nj = nj + 10: IF nj >= 256
      THEN LET nj = nj - 256: LET ns = ns + 1
1060  NEXT i
```

Type this in; precede it by some lines to renumber:

```
1  REM
2  REM xxxx
17 REM
```

and so forth; hit GO TO 1000.

Oh dear: it crashes with an error message:

```
N  Statement Lost, 1060: 1
```

Why?

LIST, and think hard.

SECOND ATTEMPT

Of course . . . the silly thing eventually starts renumbering *itself*. Once 1030 has been renumbered, the NEXT i in 1060 sends the machine back to 1060, which doesn't exist any more . . .

Well, that's easily cleared up: stop the renumbering *before* hitting the renumber routine itself. The easy way is to change line 1030, replacing vars-1 by vars-len, where len is the length of the renumber routine in bytes. (Find this using vars-prog.) With this particular routine, len = 385, so line 1030 should be

```
1030  FOR i = prog TO vars - 385
```

and now it works. And, for BASIC, relatively fast. It takes about 20 seconds to renumber a 50-line program, which while not instantaneous, is quick enough to save a lot of work. (For a simple and equally limited Machine Code routine, that *is* instantaneous, see *Machine Code and Better Basic* p. 159, or *Spectrum Machine Code* Chapter 16.)

THIRD ATTEMPT

Now, it may be occurring to you that we're possibly being a bit dumb. Those line-lengths bytes LS and LJ are exploitable: instead of PEEKing laboriously along looking for ENTER, we should be able to jump immediately to the next line-number bytes by adding on the line-length (give or take a byte or two).

It may also occur to you that this may not save very much time, because of the processing required for the addition and so forth. The only way to find out if it does is to try it.

Here's a program written along these lines: it clearly has at least one advantage—it's shorter.

```
1010 LET i = PEEK 23635 + 256 * PEEK 23636:
     LET ns = 0: LET nj = 10: LET fin =
     PEEK 23627 + 256 * PEEK 23628 - 285
1020 IF i >= fin THEN STOP
1030 POKE i, ns: POKE i + 1, nj: LET nj = nj + 10:
     IF nj >= 256 THEN LET nj = nj - 256:
     LET ns = ns + 1
1040 LET i = i + PEEK (i + 2) + 256 * PEEK (i + 3) + 4:
     GO TO 1020
```

The variable *fin* is of course the old *vars*, less the length (285) of this routine.

To see which method is faster, we load into each a reasonably long program using MERGE, and time the execution of the renumbering routine. (You'll need programs with line numbers less than 1000 to avoid overwriting part of the routine, though: see below.) Do this before reading on . . . or at least make an educated guess . . .

On my test run, with a 50-line program, the times were:

First routine: 20 seconds
Second routine: 1 second

This utterly dramatic improvement shows how important it is not to stop thinking about a program, just because it *works*.

FINAL (?) VERSION

We haven't finished yet, though. The final job is to refine the routine to maximize its usefulness. What criteria should be satisfied?

1. The routine should occupy as little memory as possible.
2. It should be written on as few lines as possible.
3. Those lines should be somewhere that is seldom, if ever, used in a normal program. Failing a Machine-Code cheat and a lowered RAMTOP, the place to put them is on lines 9995–9998. (Save 9999 just in case you want to tack on an extra piece of program, using 9999 GO TO wherever . . .)
4. The names chosen for variables should be things you don't normally use, so that the routine can be safely MERGED and used on a program with directly entered variables.

5. It would be nice to start and end on arbitrarily chosen lines, and to have arbitrary increments and start values for the new numbers. (The first requires extra checking which will cost time, and I won't include it. The second is essential: you may want to MERGE one of your favourite routines into something whose line-numbers overlap: so you'd first want to renumber the routine suitably.)

Taking these criteria into account (and noting that it's not possible to be all things to all men: some conflict with each other, requiring a compromise) I ended up with this:

```
9994 INPUT "Start, inc?"; o1, o2: LET o3 =
     INT (o1/256): LET o4 = o1 - 256 * o3
9995 LET o = PEEK 23635 + 256 * PEEK 23636:
     LET oo = PEEK 23627 + 256 * PEEK 23628 - 315
9996 IF o >= oo THEN STOP
9997 POKE o, o3: POKE o + 1, o4: LET o4 = o4 + o2:
     IF o4 >= 256 THEN LET o4 = o4 - 256: LET
     o3 = o3 + 1
9998 LET o = o + PEEK (o + 2) + 256 * PEEK (o + 3) + 4:
     GO TO 9996
```

This takes a little longer—about 3 seconds on a 50-line program. That's the price paid for a little more flexibility.

The reason for all those o's, of course, is that I hardly ever use 'o' for variables in normal programs, because of the danger of confusing it with 'zero'.

This is a genuinely useful routine. The idea is to save it under a name like "ren". Before writing a program, load it in up at the top end of the program area (thanks to its high line numbers). Then write your program. Tidy the lines by calling *ren*, using GO TO 9994; edit the GO TOs and GO SUBs to their correct values; and when you're happy, edit out lines 9994–9998 and save the final program.

If you've forgotten to load *ren* to begin with, you can always MERGE it later.

BLOCK RENUMBERING

This program is fine if you want to produce listings that go 10, 20, 30 . . . inexorably and without gaps—or 102, 104, 106, . . . for variety—but that's not always what you want. It tidies the listing; but it may make it less useful.

If you've read pages 87–92 of *Easy Programming*, on good style, you'll know that one way to produce civilized programs that work is to break them into blocks, each block being a subroutine; to use named line numbers for the blocks (such as LET block = 1000); and to start each block off at a nice round number (1000, 2000, etc. depending on how many blocks you've got to deal with).

Renumbering from start to finish will somewhat subvert this carefully produced structure. On the other hand, when you're developing block 1000 you'll rapidly find that debugging gives lines like 1035, then later on 1032 and 1033, and so on; and it either ends up all untidy or it may even leave you needing to insert a line between 1046 and 1047. The BASIC interpreter won't like the idea of line 1046½ or 1046.5.

So it would be nice to renumber *within a block* (where, incidentally, the GO SUB and GO TO problems are much less, and in a well-structured program, largely nonexistent). The following routine does this: you input the start and end of the block, the new starting number, and the new increment: it does the rest.

If you ask it to renumber a block starting from a lower line number than the current one, it BEEPs and asks for the input again. (This is done because the BASIC system is unhappy if lines get out of order in RAM.) Similarly if you finish renumbering the block with a line number that is larger than the following one, it BEEPs and keeps on renumbering to the end.

The routine as written lives on line 9000. If you input by direct command LET ren = 9000 you can access it by typing GO TO ren. Of course, a higher line number like 9990 might be preferable, and you can change the variables to less common ones as above: here I'll avoid such side-issues for clarity.

```
8999 STOP
9000 INPUT "Start/end of block"; stt, end:
     INPUT "Start/inc of new numbers"; nst, inc:
     IF nst < stt THEN BEEP .1, 0: GO TO 9000
9010 IF end > = 8999 THEN LET end = 8998
9020 LET ns = INT (nst/256): LET nj = nst - 256 * ns
9030 LET i = PEEK 23635 + 256 * PEEK 23636
9040 IF 256 * PEEK i + PEEK (i + 1) < stt THEN
     GO SUB 9080: GO TO 9040
9050 IF 256 * PEEK i + PEEK (i + 1) > end THEN
     GO TO 9090
9060 POKE i, ns: POKE i + 1, nj: LET nj = nj + inc:
     IF nj > = 256 THEN LET nj = nj - 256: LET ns = ns + 1
9070 GO SUB 9080: GO TO 9050
9080 LET i = i + PEEK (i + 2) + 256 * PEEK (i + 3) + 4: RETURN
9090 IF 256 * ns + nj - inc > 256 * PEEK i + PEEK (i + 1)
     THEN BEEP .1, 20: LET end = 8998: GO TO 9060
```

Again this is a genuinely practical utility, especially in conjunction with MERGE.

Project

Think about automatically taking care of GO TO and GO SUB renumbering. (I haven't talked about this here, partly out of laziness, partly because the result is much slower-running, and partly because the use of *named subroutines*, often good programming practice, requires even more thought. For instance, in SUPERFILLER the subroutine *test* is on line 2000, and this is set up by initializing the variable test = 2000 in line 20. When line 2000 gets renumbered, you don't need to change GO SUB test (line 250): what you *do* have to change is line 20, the value given to *test*. And even if you take care of this, there are tricks with GO SUB that renumbering messes up.)

The routines are not protected against line numbers going too high (above 9999). Do this. (It will slow the thing down a bit more, though—is it worth it?)

Not only is this simply beautiful—
it's beautifully simple too.

13 Polygons

This is basically a straightforward idea, but you can only do it on your own if you're happy about trigonometry (SIN, COS, TAN, and the like). The object is to develop the Spectrum's graphics to allow the construction of polygons—*and* fancier creatures of the same ilk.

A polygon, you'll no doubt recall, is a figure made up out of straight line segments. In a sense, that's all the Spectrum can draw anyway, but when the segments get short enough, the results approximate curves. Which is why a hi-res picture of Bo Derek doesn't *look* as if her figure is made up of straight lines . . .

Harrumph: back to the Spectrum. A polygon is *regular* if all of its sides and all of its angles are the same, so it's nice and symmetrical. To draw a regular n-sided polygon, in a circle of radius r, centred at x, y, use this:

```
10  INPUT "Number of sides?"; n
20  FOR i = 0 TO n
30  LET a = x + r * COS (2 * i * PI / n):
    LET b = y + r * SIN (2 * i * PI / n)
40  IF i = 0 THEN PLOT a, b
50  DRAW a - PEEK 23677, b - PEEK 23678
60  NEXT i
```

As it stands, there's a danger of going off-screen, so add:

```
5   IF r > x OR r > y OR r > 255 - x OR r > 175 - y
    THEN RETURN
```

The "RETURN" is because I'm thinking of using all this in a subroutine; so of course I'll also need

```
70  RETURN
```

to tidy it all up. Great, but so far there's no way to use it. So we add:

```
100 INPUT "Radius?"; r
110 INPUT "Centre?"; x, y
120 CLS
130 GO SUB 5
```

Try this out; but remember to start it with GO TO 100.

Fine, but it soon gets boring. However, we can make the results much prettier by using a loop. For example, delete line 10 and input

```
200  LET n = 5: LET x = 127: LET y = 87
202  CLS
210  FOR r = 5 TO 85 STEP 5
220  GO SUB 5
230  NEXT r
```

which gives the figure 13.1.

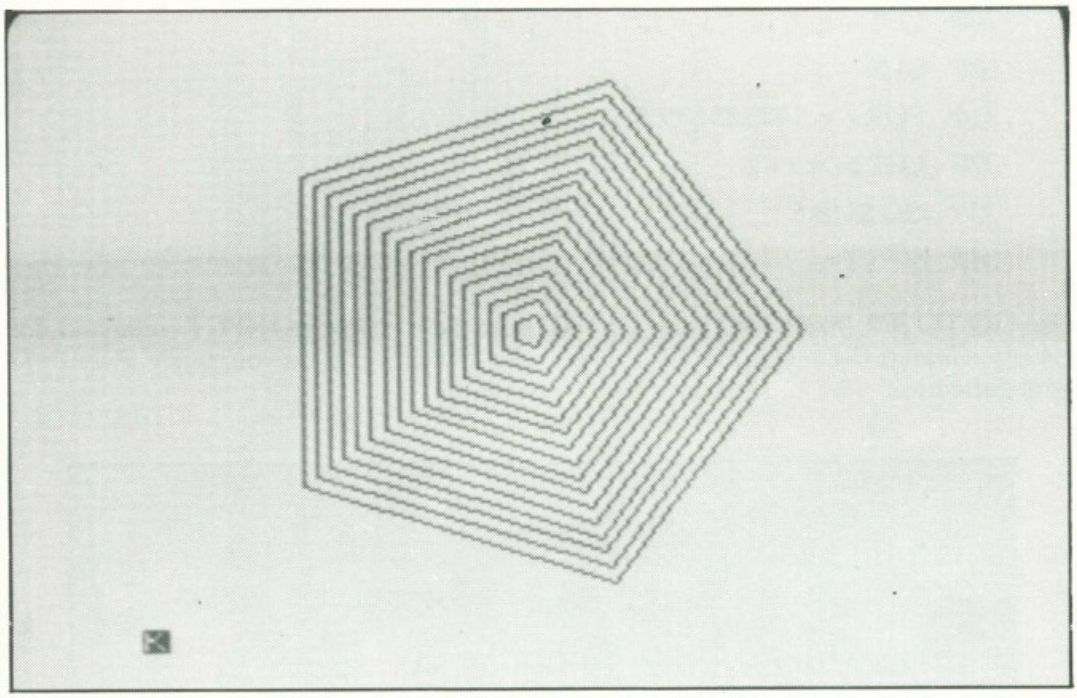


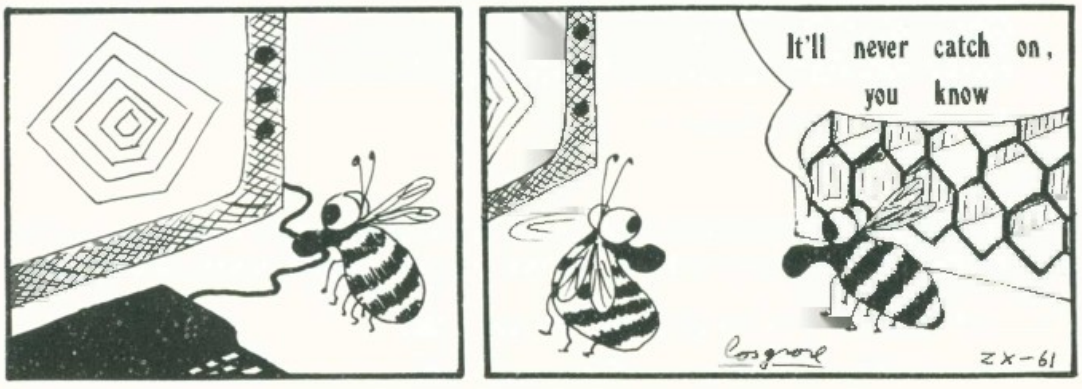
Figure 13.1 Concentric pentagons.

If you like that, get rid of the "LET n = 5" in line 200, and add:

```
205  INPUT "Number of sides?"; n
```

so that you can try different numbers. GO TO 200 now.

Try n = 50: pretty good circles (though slower than CIRCLE will get). Which is what I meant by the remark about Bo Derek . . .



ROTATIONS

After a time, even this palls. Those wretched polygons always point the same way. So let's add in a rotation:

```
206  INPUT "Rotate?"; ro
```

and now we rotate it all by ro degrees, provided we change line 30 to:

```
30  LET a = x + r * COS (2 * i * PI / n + ro * PI / 180):
    LET b = y + r * SIN (2 * i * PI / n + ro * PI / 180)
```

That lets us do even fancier things:

```
300  LET n = 7: LET x = 127: LET y = 87
305  CLS
310  FOR r = 5 TO 85 STEP 5
320  LET ro = r * 2
330  GO SUB 5
340  NEXT r
```

Use GO TO 300: you'll get Figure 13.2. Change the n = 7 to an INPUT command, for variety. Change the r * 2 in line 320 to r, or r * 3, or whatever takes your fancy. Even r * r / 50 is quite nice.

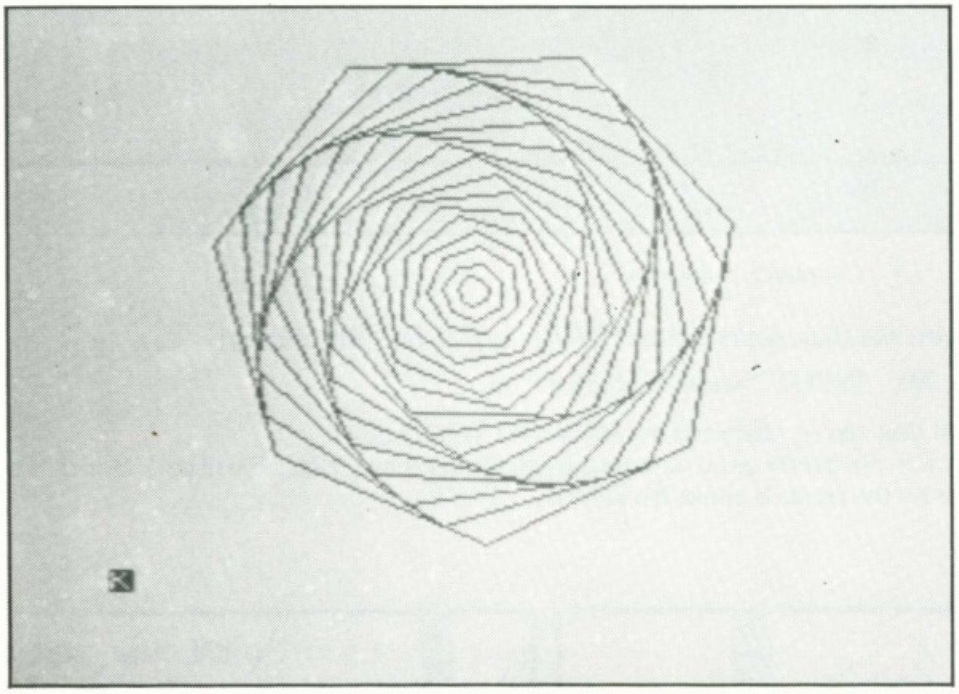


Figure 13.2 Rotating heptagons.

Looking at line 30 in its current form, I can't help wondering what would happen if the plot positions a and b were rotated by *different* amounts. In other words, change it to

```
30  LET a = r * COS (2 * i * PI / n + ro1 / 180):
    LET b = r * SIN (2 * i * PI / n + ro2 / 180)
```

where ro1 and ro2 are input using:

```
26  INPUT "Rotate 1 and 2?"; ro1, ro2
```

Try this on a GO TO 200 start, first. It kind of twists the polygon up. Now adapt line 320 of the routine starting at 300, say to

```
320  LET ro1 = r * 2: LET ro2 = r * 3
```

That's getting quite complicated now; and the designs are getting less predictable.

CURVED SIDES

What else? Well, the DRAW command can be used to draw curves as well as straight lines (*Easy Programming*, p. 34). So we can add that as an option; change line 50 to:

```
50  DRAW a - PEEK 23677, b - PEEK 23678, bend
```

and arrange to input the amount of bend somewhere, such as

```
207  INPUT "bend?", bend
```

You'll find that bends between about −4 and +4 work best; and I prefer the negative values myself.

Experiment for a while with GO TO 200. Now put the bends into the loop at 300. For example, add

```
315  LET bend = -r/25
```

and change 320 to

```
320  LET ro1 = r/2: LET ro2 = r/3
```

Getting quite complex, now . . .

STARS

If you change line 20 to (say)

```
20  FOR i = 0 TO 2 * n
```

then you can input values n = 5/2, 7/2, etc. to get a 5-pointed or 7-pointed *star*. With

```
20  FOR i = 0 TO 3 * n
```

you can try n = 5/3, 7/3, 8/3, and so on; and in general with

```
20  FOR i = 0 TO k * n
```

values of the form n = (whole number) / k will produce star-like creatures. (You'll need to input or assign k.)

Then you can add colour commands; plot using OVER 1 . . . The variety is endless. Here's quite a nice one to finish with: key it in and GO TO 400. Figure 13.3 shows the result.

First change line 20 of the subroutine to:

```
20  FOR i = 0 TO 3 * n
```

and then add:

```
400  LET n = 11/3: LET x = 127: LET y = 87
405  PAPER 0: BORDER 0: OVER 1: CLS
410  LET ro1 = 0: LET ro2 = 0
415  FOR r = 5 TO 75 STEP 10
418  LET bend = -r/25
```

```
420 INK 1 + r/16
430 GO SUB 5
440 NEXT r
```

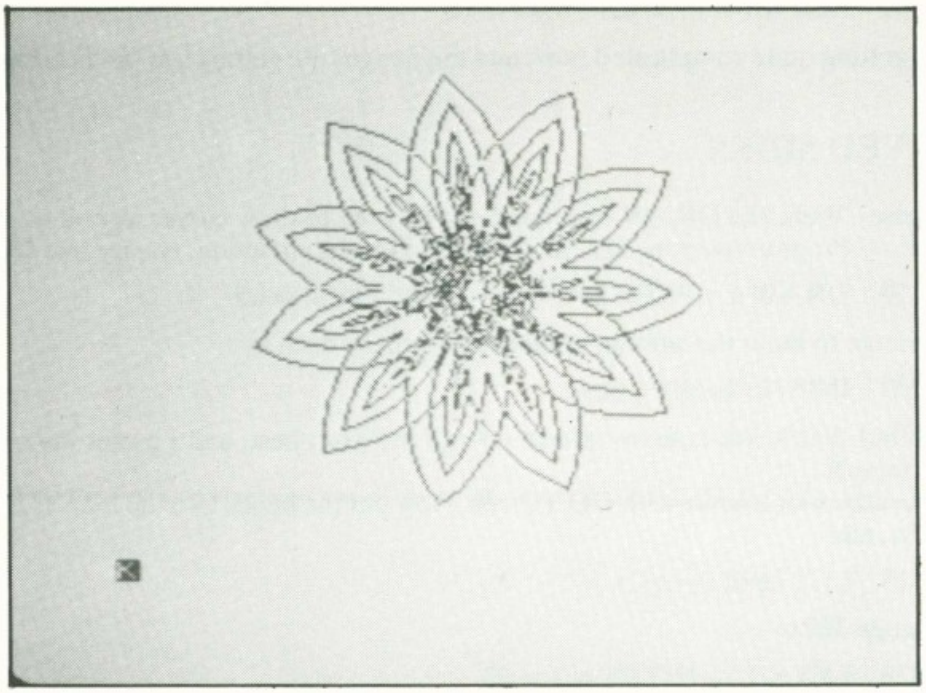


Figure 13.3 Curved-sided 5½-gons.

You can obviously use these routines inside other programs: for example, we haven't tried changing the centre x, y at all. See if you can draw a 4 × 4 array of pentagons; a line of overlapping heptagons; a curved line of hexagons growing larger and larger *and* rotating; a random arrangement of randomly coloured random-sided polygons and stars . . .

But if you've followed me this far, you shouldn't need further urging to have a go yourself.

Esnes erom sekam ti sdrawkcab
gnidaeh siht daer uoy fi . . .

14 Cryptography and Cryptanalysis

The art of putting things into code; and decoding the results without knowing what the code is.

It is a sad comment on the human condition that the earliest recorded use of coded messages, to avoid interception by the enemy, goes back to the Lacedaemaonians in 400 BC. The earliest book on the subject was *On the Defence of Fortifications* by Tacticus in the fourth century BC.

The converse problem—decoding a message without knowing the code—has more than just military significance. Historians need to know what was communicated in messages between commanders during the American Civil War; and linguists need to understand ancient scripts such as Egyptian hieroglyphs or Mycenaean Linear B. Some of these may not have been code *then*, but they sure are *now*.

The computer can be a powerful weapon for the cryptanalyst, because it can carry out "enlightened" trial-and-error at high speed. (Unenlightened trial-and-error takes far too long, even on a Cray-1. Your Spectrum wouldn't stand a chance.)

The simplest codes are *substitution codes*, in which each letter of the original message is encoded according to a fixed jumbled-up alphabet; to keep the chapter within bounds I'll concentrate on those.

SUBSTITUTION CODES

For example, suppose the message is

"My dog has four legs"

and the code is defined by

abcdefghijklmnopqrstuvwxyz
zvetrsnbjpkcuxdfgayohlqimw

The the coded message reads

"um tdn bzy sdha crny"

reading off from top row to bottom row.

The following program accepts a message and encodes it by a random substitution code. It will be developed into a routine to *decode* such messages, given an intelligent user.

```
100  LET a$ = "abcdefghijklmnopqrstuvwxyz"
105  PRINT a$
110  LET b$ = " "
120  FOR i = 1 TO 26
130  LET m = INT (1 + (27 - i) * RND)
```

```
140 LET b$ = b$ + a$ (m): LET a$ = a$ (TO m - 1) + a$ (m + 1 TO)
150 NEXT i
160 PRINT b$
```

So far this just randomizes the order of the alphabet by selecting the first letter at random, then the second at random from those left, and so on. Study it carefully: the magazines are full of "randomization" routines that pick two letters at random, swap them, and repeat a great many times: that's an incredibly slow way to achieve the result!

Now to input and encode the message:

```
200 INPUT m$
210 PRINT m$
215 LET c$ = " "
220 FOR i = 1 TO LEN m$
225 LET c = CODE m$ (i)
230 IF c = 32 THEN GO TO 250
235 IF c < 97 THEN LET c = c + 32
240 IF c >= 97 AND c < 128 THEN LET c$ = c$ + b$ (c - 96)
250 NEXT i
260 PRINT c$
```

The main points to note here are that line 235 converts capitals to lower case; 230 ignores spaces; and the c − 96 in line 240 occurs because the alphabet occurs for characters 97–123.

FREQUENCY ANALYSIS

How would a cryptanalyst approach such a code? (Of course, in practice this method of encoding messages is too easily broken to be of any real use: our immediate job is to see why.)

The essential thing to notice is that ordinary English does not use letters equally often. The letter "E", for example, occurs far more frequently than "Z". Here's a table of the average rate of occurrence (per 100 letters), determined by analysing government telegrams.

a	.074	n	.079
b	.010	o	.075
c	.031	p	.027
d	.042	q	.003
e	.130	r	.076
f	.028	s	.061
g	.016	t	.092
h	.034	u	.026
i	.074	v	.015
j	.002	w	.016
k	.003	x	.005
l	.036	y	.019
m	.025	z	.001

In other words, the most common letter is "e", occurring about 13% of the time; next "t" at 9.2%; then "n", "r", "o", "a", "i", "s", "d", after which the frequency of occurrence is less than 4%. So it's a good guess that the commonest letter in a coded version of a fairly long message should represent "e", and so on.

For the message above (which is rather short) the frequencies are:

a	1	out of 16
b	1	
c	1	
d	2	
h	1	
m	1	
n	2	
r	1	
s	1	
t	1	
u	1	
y	2	
z	1	

The commonest are "n" and "y", standing for "g" and "s" respectively; not much help, but then it's a very short message. If we started with a longer text, such as

"PEEK, POKE, BYTE and RAM is an excellent computer book"

then the coded version would be:

"frrk fdkr vmor zxt azu jy zx rierccrxo edufhora vddk"

(and in practice the spaces would be omitted). The frequencies are then:

a	2	out of 43
c	2	
d	4	
e	2	
f	3	
h	1	
i	1	
j	1	
k	3	
m	1	
o	3	
r	8	
t	1	
u	2	
v	2	
x	3	
y	1	
z	3	

The commonest letter is "r", which we assume stands for "e"; next come "d", "f", "k", "o", "x". If the frequencies hold good, these should be "t", "n", "r", "o", "a" respectively, in some order. In fact they are "o", "p", "k", "t", "n". We've got "n", "o", and "t" in there: a little trial-and-error will sort them out.

Indeed, just knowing the "e" tells us that the message is:

.ee. ...e ...e ... ... e..e..e.. ......e. ....

and we're well on the way. Some four-letter words that go ".ee." are beef, beer, been, beet, beep, bees, deed, deem, deep, deer, feed, feel, feet, heed, heel, jeep, jeer, keel, keen, keep, leek, leer, lees, leet, meed, meek, meet, need, neep, peek, peel, peep, peer, reed, reef, reek, reel, seed, seek, seem, seen, seep, seer, teed, teem, weed, week, weep. That's 48 to try: eventually you'd get the answer.

THE PROGRAM

So a cryptanalysis program for substitution codes should present you with an analysis of the relative frequencies of the various letters; then let you try guesses and see what the

result is. Which leads to the following program:

```
500  REM Frequency analysis
510  DIM n (26)
515  LET col = 0
520  LET tot = LEN c$
525  FOR i = 1 TO 26
530  FOR j = 1 TO tot
540  IF CODE c$ (j) - 96 = i THEN LET n (i) = n (i) + 1
550  NEXT j
554  LET s$ = STR$ (.01 * INT (100 * n (i) / tot) ):
     IF s$ (1) = "." THEN LET s$ = "0" + s$
555  PRINT TAB col; CHR$ (i + 96); "□"; s$;
560  LET col = col + 8
570  IF col = 32 THEN LET col = 0
580  NEXT i
```

This works out the frequencies. Lines 555–580 produce a 4-column print-out. Line 554 is an attempt (successful) to produce only two decimal places. If you omit the second bit about s$ (1) you'll find that some numbers print out like

```
.09
```

and others like

```
0.34
```

which is messy and untidy. Putting an initial zero clears this up. But it's a piece of pedantry, really.

Now for trial-and-error decode:

```
1000  REM decode
1010  LET p$ = " ": FOR i = 1 TO tot:
      LET p$ = p$ + ".": NEXT i
1020  PRINT AT 15, 0; p$
1100  REM trial
1110  INPUT "Code letter?"; k$: IF LEN k$ < > 1
      THEN GO TO 1110
1120  INPUT "Guess at decode?"; g$: IF LEN g$ < > 1
      THEN GO TO 1120
1130  FOR i = 1 TO tot: IF c$ (i) = k$ THEN
      LET p$ (i) = g$
1135  NEXT i
1140  PRINT AT 15, 0; p$
1150  GO TO 1110
```

Not bad. But there are snags. You can attempt to use the *same* letter to decode different letters in the coded message, and not notice: that way lies disaster! So it would be nice to check on this. To do so, we need to keep a record of the current state of knowledge.

```
10   DIM d (26)
1125 GO TO 1500
1500 REM record of choice
1510 LET k = CODE k$ - 96: LET g = CODE g$ - 96
1520 FOR a = 1 TO 26
1525 IF d (a) < > g OR d (a) = k THEN GO TO 1560
1530 INPUT "You have used □"; CHR$ (g + 96);
     "□ for code letter □"; CHR$ (a + 96);
     "□ do you want to leave it that way?"; y$
1540 IF y$ = "y" THEN GO TO 1110
1550 LET d (a) = 0
1555 GO SUB 2000
1560 NEXT a
1570 NEXT i
1600 LET d (k) = g
1610 GO TO 1130

2000 FOR b = 1 TO tot
2010 IF p$ (b) = g$ THEN LET p$ (b) = "."
2020 NEXT b
2030 RETURN
```

All we need now is a way to exit, properly informed, when we think we've cracked it:

```
1115 IF k$ = "0" THEN GO TO 2500
```

This lets us input "0" when asked "code letter?" to wrap it all up.

```
2500 REM wrap it all up
2510 CLS
2520 PRINT "Code message: " ' c$ ' '
     "Decoded message: " ' p$ ' ' '
2530 PRINT "Code known so far:"
2540 FOR i = 1 TO 26
2550 PRINT AT 12, i + 3; CHR$ (i + 96)
2560 IF d (i) < > 0 THEN PRINT AT 13, i + 3;
     CHR$ (d (i) + 96)
2570 NEXT i
```

To avoid cheating, you must now *delete* lines 105, 160 and 210. Get someone else to input the message. Or . . .

PROBLEM

Here are four messages for you to decode (answers at the back, page 139). They are all well-known quotations. Each is in a different substitution code.

1. jluruorufqfuofbdqcjavoadtegtfdjrmgjqfpjrqwonnoadofbuyuradqfpnutfutejluhqdr qplj
2. buxgbzzyliflqxnqxnzyngkrlyvrlbcxqxnfzefzflxhbnfxljylqxniyvxzgcyjlqxbaaxlflxrlqx jexxp
3. qbrxzuzxbzyuzaajrtzqjsrbrsjxtydhoqbrobrabrakxrryzxtrcvdqyzardxcmnchrtodnro bra
4. xjvwgzmcgzjvdjmvzgiozyzvlzznziljxvvlzuxnhljgdnzmmcgzjvvloxc

Projects

1. Modify the program to display the table of letter-frequencies if asked (say by input "1" when asked for "code letter?").
2. Add a bubble-sort (*Easy Programming*, p. 65) to list the letters used in order of frequency. This makes guessing easier.
3. Add an option (enter "2" at the "code letter?" stage) to print out the table of normal frequencies of letters, for reference.
4. The commonest two-letter combinations in English (*digraphs*) are:

en	.111	on	.Ø77
re	.Ø98	in	.Ø75
er	.Ø87	te	.Ø71
nt	.Ø82	an	.Ø64
th	.Ø78	or	.Ø64

   Write a digraph-counting routine to take advantage of this extra information.
5. Write programs to implement other codes (good references are the *Encyclopedia Brittannica* and *The Code Breakers* by D. Kahn) and to allow you to try to decode them.

Need more than 23 user-defined graphics characters? Now you can have 256 of them with a single POKE.

15 Changing the Character Set

I mentioned in Chapter 6 that you can set up new characters above RAMTOP and access them by POKEing the system variable CHARS. This chapter describes the process in detail.

For simplicity, assume we want 64 new characters. Then we'll need 64 * 8 = 512 bytes of clear space. RAMTOP normally lives at the value 32599 in a 16K Spectrum, so it has to be lowered to 32599 − 512 = 32087 to leave a 512-byte "attic". To do this, enter (directly)

```
CLEAR 32087
```

The cleared area starts at the next address, 32088. This is 125 * 256 + 88, so its junior byte is 88 and its senior is 125. We fool CHARS into pointing at this new area by deducting 1 more from the senior byte (remember, CHARS holds 256 *less* than the address of the character table). The actual command is thus

```
POKE 23606, 88: POKE 23607, 124
```

But don't input this yet.

This program lets you set up 64 new characters, and tests them to make sure all is well.

```
10   FOR i = 32 TO 96
20   GO SUB 400
30   PRINT i −31,
40   GO SUB 200
50   PRINT CHR$ i
60   PRINT '
70   NEXT i
80   GO SUB 400
90   STOP
200  REM new address for CHARS
210  POKE 23606, 88: POKE 23607, 124
220  RETURN
400  REM usual address for CHARS
410  POKE 23606, 0: POKE 23607, 60
420  RETURN
1000 REM input routine for new character set
1010 LET i = 32088
```

```
1020 INPUT j
1030 PRINT j; "□";
1040 POKE i, j
1050 LET i = i + 1: GO TO 1020
```

Here 200 resets CHARS to the new area; 400 sets it to its usual position; and 1000 is an input routine.

Start with GO TO 1000. For a test, we'll only use six characters. Exactly as in user-defined graphics, you need to draw a picture of the character on an 8 × 8 grid; convert the rows to binary 0s and 1s; then input this value (see *Easy Programming*, p. 49). Here I'll use the six characters shown in Figure 15.1: that means I must input, in order,

```
255 255 255 255 255 255 255 255    (for ■ )
255 129 129 129 129 129 129 255    (for □ )
  1   3   7  15  31  63 127 255    (for ◢ )
255 255 195 195 195 195 255 255    (for ◘ )
  1   2   4   8  16  32  64 128    (for ⁄ )
 24 126 126 255 255 126 126  24    (for ● )
```

That's enough: input STOP, then RUN. You should see the six characters listed as 1, 2, 3, 4, 5, 6 on the screen. If not, check your work carefully!

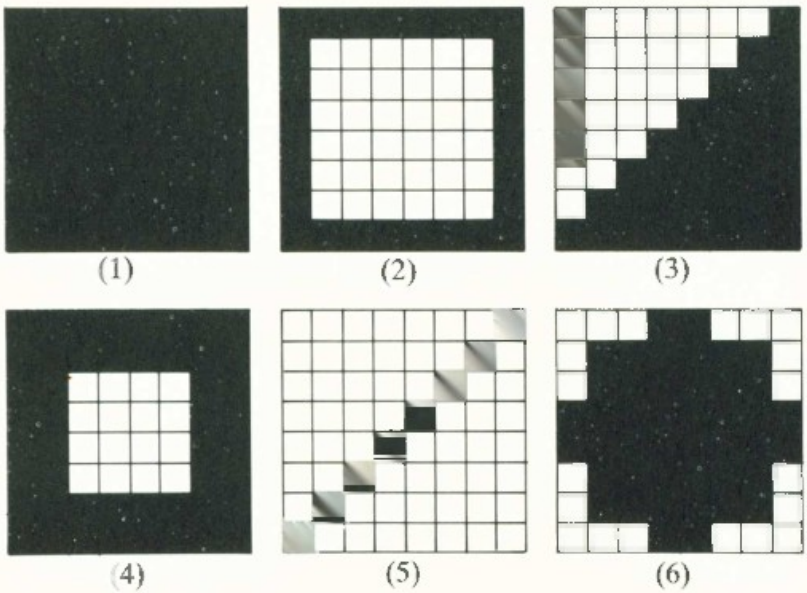


Figure 15.1 A set of six test characters.

Having seen that the method works, the final task is actually to input those 64 characters. That comes in stages:

1. *Design* them. You could use CHARACTER-BUILDER, a utility routine in *Easy Programming*.
2. Read off the data for the rows. (Ditto).
3. Input the data at line 1000 as above.

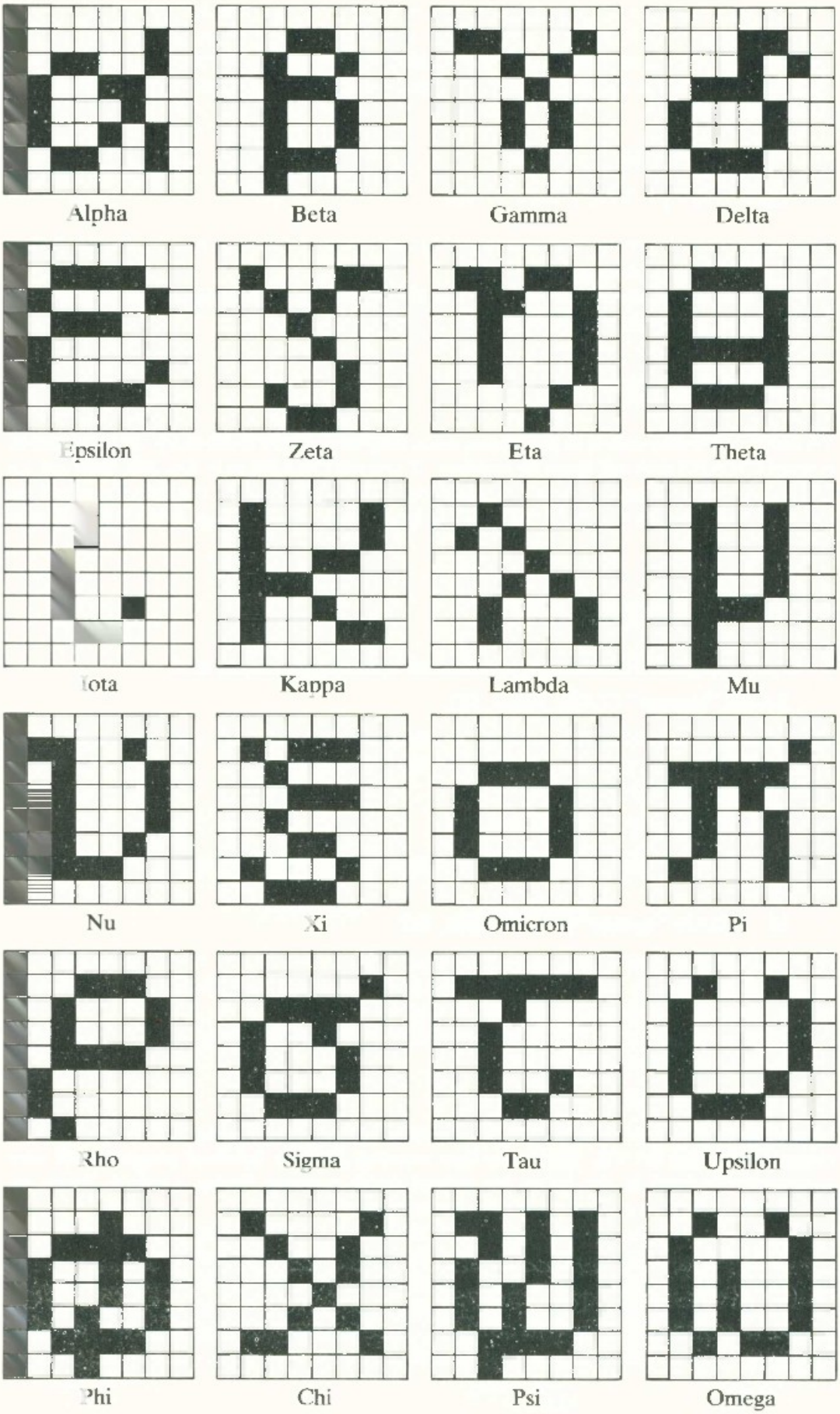


Figure 15.2 Design for a Greek alphabet.

I'm sure your mind is already thinking of lots of fancy tricks to make all this easier, such as *combining* CHARACTER BUILDER with the little program above. Let me just mention one. For direct input of the rows in *binary*, avoiding the tedious conversion to decimal (which is a nuisance since the Spectrum promptly converts back to binary) change line 1Ø2Ø to:

```
1Ø2Ø  INPUT b$
1Ø25  LET j = VAL ("BIN" + b$)
```

Now you input the rows as sequences of Øs and 1s, read off from the 8 × 8 grid: Ø for a blank, 1 for a blacked-in square.

What characters should you put in? The above suggest some possible graphics. Figure 15.2 shows a complete Greek alphabet.

USING THE NEW SET

To use the new characters in programs, once they're up there in memory, all you need do is reset CHARS and call them by number. If you set CHARS as in line 2ØØ, then ask for CHR$ 31 + n, or simply ask for the usual character with code 31 + n *directly*, you'll get the n-th character in your new list. For the usual characters, change CHARS back as in line 4ØØ.

SAVING THE NEW SET

To SAVE your hard work on tape, you need byte storage. If you enter

```
SAVE "newset" CODE 32Ø88, 512
```

then you get the 512 bytes starting at address 32Ø88 on tape, and they are named "newset". (Run the tape the usual way for SAVE, of course.)

To load them back in, you use

```
LOAD "newset" CODE 32Ø88, 512
```

Now there's an even better way. Write a program to do the loading. First enter the program:

```
1Ø  LOAD "newset" CODE 32Ø88, 512
```

Now SAVE this, under the name "newload" (say). Use SAVE "newload" LINE 1Ø. Immediately after it, on tape, SAVE the new characters using SAVE . . . CODE as above. So now you've got two chunks of stuff on the tape.

Rewind, enter LOAD "newload", push the buttons, and watch.

The first screen message will be:

```
Program: newload
```

Immediately this has loaded in, it will *run* starting from line 1Ø, because of the LINE 1Ø command in the SAVE. *Leave the tape running* and the newload program will now automatically load in the bytes for newset (with message Bytes: newset). Saves remembering all those address numbers and suchlike . . .

This method opens up new possibilities altogether. You can chain programs end-to-end, in such a way that each calls the next. Provided you're nippy with the tape controls, and don't want to go backwards, you can make use of this idea in all sorts of ways, to effectively increase the power of the machine, by making full use of the extra memory capacity *on the tape*. Chapters 9 and 17, on Cassette Files and Data Management Systems, explore this idea in one useful context.

One minor snag with the Spectrum's PLOT and DRAW commands is that they lead to error reports if the points being plotted go off the screen. The answer is to design a utility routine for . . .

16 Crashproof Curve-plotting

The easiest way to draw curves is to write a loop which, after PLOTting a starting point, DRAWs successively to other points, generated from either a list of data or a formula. However, this runs into trouble if points go off the screen. What follows is a blow-by-blow account of the development of one method of getting over this. It gets a little bit mathematical in places; but if maths isn't your strong point, ignore the algebra and concentrate on the overall structure.

Recall that for hi-res graphics the Spectrum employs a coordinate grid having 176 rows and 256 columns (numbered 0–175 and 0–255) starting from the bottom left corner of the screen. The edge of the screen therefore forms a rectangle 176 × 256 pixels in size. The key to the whole game is to devise a subroutine which, when fed the coordinates (x1, y1) and (x2, y2) of two points, *not necessarily on the screen*, works out where the line joining them hits the edge of this rectangle (Figure 16.1).

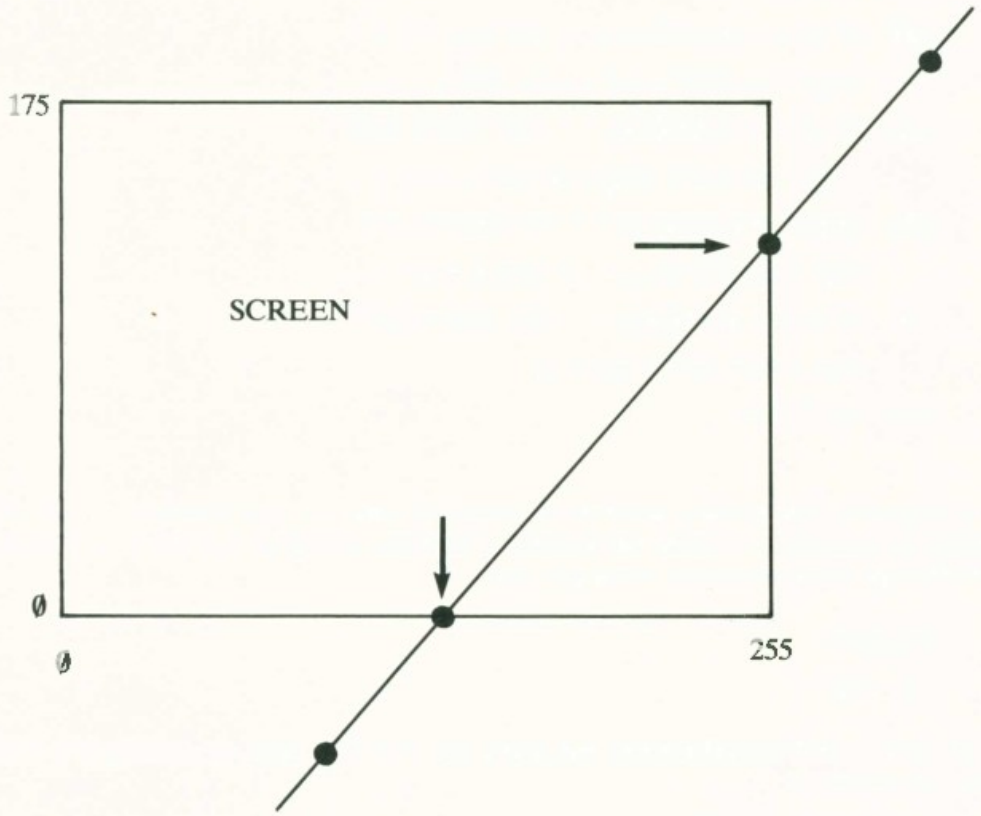


Figure 16.1 Where does a line between two points meet the edges of the screen?

A smidgin of the old coordinate geometry lets us compute these points. The algebraic expression

$$\frac{(a-b)*(d-c)}{(c-b)} + e$$

crops up repeatedly in various disguises, which is a sign that maybe a user-defined function is in order (see Chapter 3).

Using this, we get the following routine. (The line-numbers may look a bit irregular: the idea is that if you follow through this chapter and input all the program lines as you go, you'll end up with the complete program; but the description will go subroutine by subroutine. It's always easier to key in, and to design, long programs this way.)

```
    2  DEF FN a (a, b, c, d, e) = (a - b) * (d - e) / (c - b) + e
 1000  REM seg
 1010  DIM x (2): DIM y (2)
 1020  IF x1 = x2 THEN LET xt = x1:
       LET xb = x1: LET yl = -1: LET yr = -1
 1025  IF y1 = y2 THEN LET xt = -1: LET
       xb = -1: LET yl = y1: LET yr = y1
 1030  IF x1 < > x2 AND y1 < > y2 THEN LET
       xt = FN a (175, y1, y2, x2, x1): LET
       xb = FN a (0, y1, y2, x2, x1): LET
       yl = FN a (0, x1, x2, y2, y1): LET
       yr = FN a (255, x1, x2, y2, y1)
 1090  LET q = 1
 1100  IF xt > = 0 AND xt < = 255 THEN LET
       x (q) = xt: LET y (q) = 175: LET q = q + 1
 1110  IF xb > = 0 AND xb < = 255 THEN LET
       x (q) = xb: LET y (q) = 0: LET q = q + 1
 1120  IF yl > = 0 AND yl < = 175 THEN LET
       x(q) = 0: LET y (q) = yl: LET q = q + 1
 1130  IF yr > = 0 AND yr < = 175 THEN LET
       x(q) = 255: LET y (q) = yr
 1140  RETURN
```

In a program, this routine needs to be supplied with the four numbers x1, y1, x2, y2 (giving the coordinates of the two points): it puts the coordinates of the points where the line through these meets the rectangle into

```
x (1), y (1)
x (2), y (2)
```

To be able to call this subroutine we either use GO SUB 1000, or, writing in a civilized style, we initialize

```
22  LET seg = 1000
```

in which case we can use GO SUB seg. So add line 22.

It's wisest to *test* this beast before going any further: the algebra is messy enough for mistakes to go undetected at this stage, causing havoc later. Add *temporary* lines:

```
100  LET x1 = 127: LET y1 = 87
110  FOR t = 1 TO 50
120  LET x2 = 500 * SIN (t * PI / 50):
     LET y2 = 500 * COS (t * PI / 50)
130  GO SUB seg
140  PLOT x (1), y (1): DRAW x (2) - x (1), y (2) - y (1)
150  NEXT t
160  STOP
```

Now RUN: if you get a set of radial lines through the middle of the screen, stopping at the edges, all is well. If not, check the listing again. Delete 100–160 once the routine is OK.

That's done the brainwork. Most of the rest is now routine... or at least, subroutine...

HOW TO DRAW CURVES

First job is to decide the general structure. There are two main ways to draw a curve:

1. A *graph*. Plot FN (t) against t, where FN is some function. See *Easy Programming*, p. 79 for details.
2. A *parametrized curve*. Plot FNb (t) against FNa (t), where FNa and FNb are *two* functions, and t is called a *parameter*.

It would be nice if our routine allowed either option. (This isn't hard, because a graph is actually a special kind of parametrized curve, with FNa (t) = t. But one generally thinks of them in different ways.)

The user is going to have to *set up* these functions. We could have a line in the program like:

```
7777  DEF FNb (t) = some rubbish or other
```

and ask the user to *edit* this line to his desired function. But wouldn't it be nice to let him *input* his chosen function during the program run?

The VAL command (*Easy Programming*, p. 71) is tailor-made for this kind of application. If the user inputs FNb (t), say, as a *string* f$, then we can evaluate it by asking for VAL f$. For instance, if t = 71 and f$ = "t * t". then

```
VAL f$ = VAL "t * t" = 71 * 71 = 5041
```

and this *is* the value of the "square" function at t = 71.

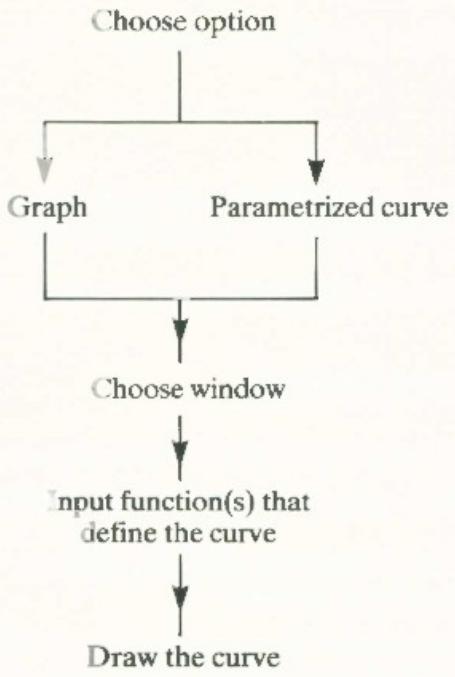
It's also going to be a nuisance if we can only plot curves where the coordinate values range from 0 to 155 and 0 to 175 (the usual problem of shifting and stretching axes, see *Easy Programming*, p. 81). A standard technique is to set up a *window* covering the desired area, and to transform the coordinates suitably during the drawing routine (Figure 16.2).

So the structure in outline is as shown over page.

Leaving the details to take care of themselves (*top-down* programming; see *Easy Programming*, p. 87) we write the main program:

```
100  PRINT "SPECTROGRAPH"
110  PRINT ' "OPTIONS:" ' ' "□ □ 1. Parametrized
     curve [11 spaces] 2. Graph"
```

```
120  INPUT "Select option number"; opt
130  GO SUB window
140  CLS
150  IF opt = 1 THEN GO SUB param
160  IF opt = 2 THEN GO SUB graph
170  STOP
```



Outline structure for setting up a window.

SUBROUTINES

That leaves us with three subroutines to write: *window*, *param* and *graph*. *window* is simplicity itself:

```
20   LET window = 500
500  REM window
510  INPUT "Window coordinates: left, right,
     bottom, top"; wl, wr, wb, wt
520  RETURN
```

I've already said that *graph* is a special case of *param*, so obviously the thing to do is write *param* first and then hope *graph* takes care of itself.

To plot a parametrized curve, with parameter t, we need to know two things: the range of values for t, and the size of steps along this range at which points are computed. So *param* has to ask for these, and then draw the curve.

```
26   LET param = 2000
2000 REM param
2010 INPUT "Parameter range: left, right"; tl, tr
2020 INPUT "Number of steps?"; ns
```

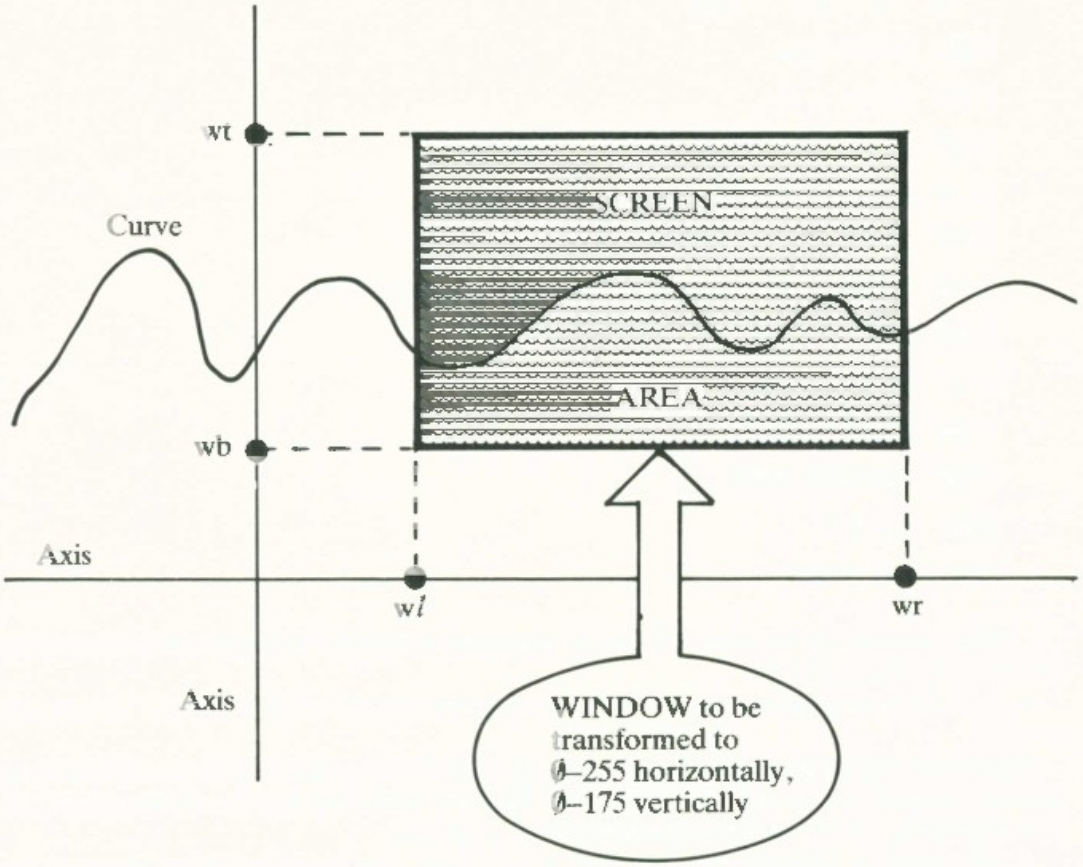


Figure 16.2 The screen area used as a window.

```
2030  INPUT "Specify x and y as functions of t"; x$, y$
2040  LET stepsize = (tr - tl) / ns
2050  GO SUB draw
2060  RETURN
```

That's managed to leave most of the work to the subroutine *draw*, not yet written. Great!

graph works much the same way, and jumps into *draw* at a suitable point:

```
  24  LET graph = 1500
1500  REM graph
1510  LET tl = 0: LET tr = 255
1520  LET ns = 255
1530  INPUT "Specify function of t"; y$:
      LET x$ = "t"
1540  LET stepsize = 1
1550  GO SUB draw
1560  RETURN
```

I've deliberately written this to bring out the analogy with *param*: *graph* automatically sets all of the variables in *param* except for y$, which is the function input, and then calls *draw* in exactly the same way. (You *could* replace the last two lines by GO TO 2050, but good style suggests otherwise provided it doesn't waste a lot of memory or time.)

We can't put off the evil moment any longer . . .

```
  28  LET draw = 2500
2500  REM draw
2510  LET t = tl
2520  LET u = VAL x$: LET v = VAL y$
2530  GO SUB transf
2540  IF FN o (u, v) THEN PLOT u, v
2550  FOR t = tl + stepsize TO tr STEP stepsize
2560  LET u = VAL x$: LET v = VAL y$
2570  GO SUB transf
2580  GO SUB flag
2590  GO SUB d (fl)
2600  NEXT t
2610  RETURN
```

. . . Well, maybe we can after all. We've managed to invent three new subroutines (one of which has four parts):

transf which transforms the variables so that the chosen window fits the screen area exactly,

flag which sets up a flag to tell us whether the points to be joined up in *draw* are both on screen, both off, or on and off respectively. This sets a variable fl to one of the values 1, 2, 3, 4 depending on the precise combination of positions.

d (fl) which is actually *four* routines *d (1)–d (4)*, for each flag value, because the actions needed are quite different from one case to the next.

We've also introduced a new user-defined function FN o (see Chapter 3). This is supposed to be an "on screen" flag. That is, if we define

```
1  DEF FN o (x, y) = x > = 0 AND x < = 255
   AND y > = 0 AND y < = 175
```

then FN o (x, y) = 1 if (x, y) is *on* the screen, and FN o (x, y) = 0 if (x, y) is *off* the screen. Now isn't that pretty? Of course you could do it other ways: my other half would probably define a flag called *onscreen* and write 2540 IF onscreen THEN . . . . In fact, this works with onscreen = FN o (x, y). Anything to make the listing look like the Queen's English rather than Einstein's Law of Gravity.

You may feel we're not getting anywhere yet, because we haven't approached the central problem of actually *drawing* stuff. Have faith: can't you *feel* the problem getting smaller as we knock bits off the corners and break it up into smaller pieces?

Transforming for a suitable window is easy if you've had six years of mathematical training like I have:

```
  30  LET transf = 3000
3000  REM transf
3010  LET u = (u − wl) / (wr − wl) * 255:
      LET v = (v − wb) / (wt − wb) * 175
3020  RETURN
```

And you can even check that I'm right. What we want is for the u-coordinates wl and wr to transform to 0 and 255; and wb, wt to transform to 0, 175. Now putting u = wl on the right-hand side gives u = 0 on the left; and putting u = wr gives u = (wr − wl) / (wr − wl) * 255 = 1 * 255 = 255 . . . gosh! And wt, wb work just the same way.

To pick up the thread:

```
32   LET flag = 3200
3200 REM flag
3210 LET fl = FN o (xo, yo) + 2 * FN o (u, v) + 1
3220 RETURN
```

That works like this: suppose we want to join the *old* point (xo, yo) to the *new* point (u, v). Then we have:

Old point	New point	Value of *fl*
on	on	4
on	off	2
off	on	3
off	off	1

(where on/off refer to whether the point is on or off the screen). So the value of *fl* distinguishes the cases.

Why distinguish? The required action is:

Value of *fl*	Action required
1	Draw that part of the line between the old and new points, that lies on-screen (if any)
2	Join the old point to the edge of the screen, along the line towards the new point
3	Join the edge of screen to the new point, along the line from the old one
4	Join the old point to the new

The reason for action (1) is that it *may* happen that, while both old and new lie off-screen, part of the line between them should fall *on* it, hence be drawn in (see Figure 16.1).

DRAWING ROUTINES

Here they all are, written in the easiest order for my poor little brain at the time.

```
34   DIM d (4)
37   LET d (1) = 3800
38   LET d (2) = 3600
39   LET d (3) = 3700
40   LET d (4) = 3500
3500 REM d (4)–both on
3510 DRAW u – PEEK 23677, v – PEEK 23678
3520 RETURN
3600 REM d (2)–old on, new off
3610 LET x1 = xo: LET y1 = yo:
     LET x2 = u: LET y2 = v
```

```
3620 GO SUB seg
3630 LET z = 1
3640 IF u < > xo AND SGN (u − x (1) ) < > SGN (x (1) − xo)
     THEN LET z = 2
3650 IF u = xo AND SGN (v − y (1) ) < > SGN (y (1) − yo)
     THEN LET z = 2
3660 DRAW x (z) − PEEK 23677, y (z) − PEEK 23678
3670 RETURN
3700 REM d (3)–old off, new on
3710 LET x1 = xo: LET y1 = yo:
     LET x2 = u: LET y2 = v
3720 GO SUB seg
3730 LET z = 1
3740 IF u < > xo AND SGN (u − x (1) ) < > SGN (x (1) − xo)
     THEN LET z = 2
3750 IF u = xo AND SGN (v − y (1) ) < > SGN (y (1) − yo)
     THEN LET z = 2
3760 PLOT x (z), y (z): DRAW u − x (z), v − y (z)
3770 RETURN
3800 REM d (1)–both off
3810 LET x1 = xo: LET y1 = yo: LET x2 = u: LET y2 = v
3820 GO SUB seg
3830 PLOT x (1), y (1): DRAW x (2) − x (1), y (2) − y (1)
3840 RETURN
```

All that stuff with SGN is to find out which of the points x (1), y (1) or x (2), y (2) is the right one to use. Ignore it if you hate maths.

That's almost it. The variables xo and yo for the old point's coordinates haven't been set up anywhere, though. The right place is at:

```
2535 LET xo = u: LET yo = v
```

and again at

```
2575 LET ur = u: LET vr = v
2595 LET xo = ur: LET yo = vr
```

TESTING

Now we're ready to test.

RUN, and type in:

Option	1			
Window	−5	5	−5	5
Parameter range	−5	5		
Number of steps	100			
Functions of t	t	t		

All is well: a (not entirely straight) line climbs from lower left to top right. (It didn't? You've boobed!)

Something a bit harder? Let's go for a parabola, part of which is off-screen. (The first test didn't use subroutines d (1)–d (3) at all . . . which is the main point of the whole exercise!)

Option	1			
Window	−5	5	−1	20
Parameter range	−5	5		
Number of steps	100			
Functions of t	t	t * t		

Figure 16.3 shows the result. It's pretty—but it's a funny parabola . . .

You may or may not be surprised how long it took me to find the bug. Too long—I must have been getting tired. A LPRINT trace showed that the culprit was subroutine d (1)—joining up two points, both off-screen. But what was wrong with it?

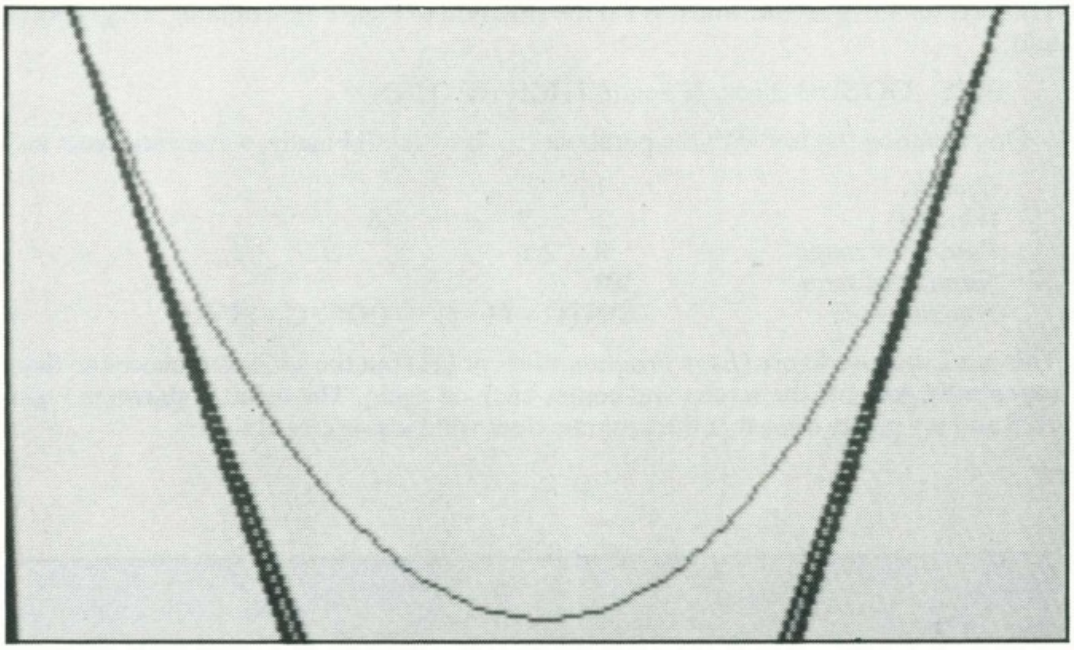


Figure 16.3 A parabola! Well, almost . . .

Eventually the obvious dawned. If the two points joined are both off-screen on the *same* side, the *line* through them can meet the screen even though the line *segment* between does not (see Figure 16.4).

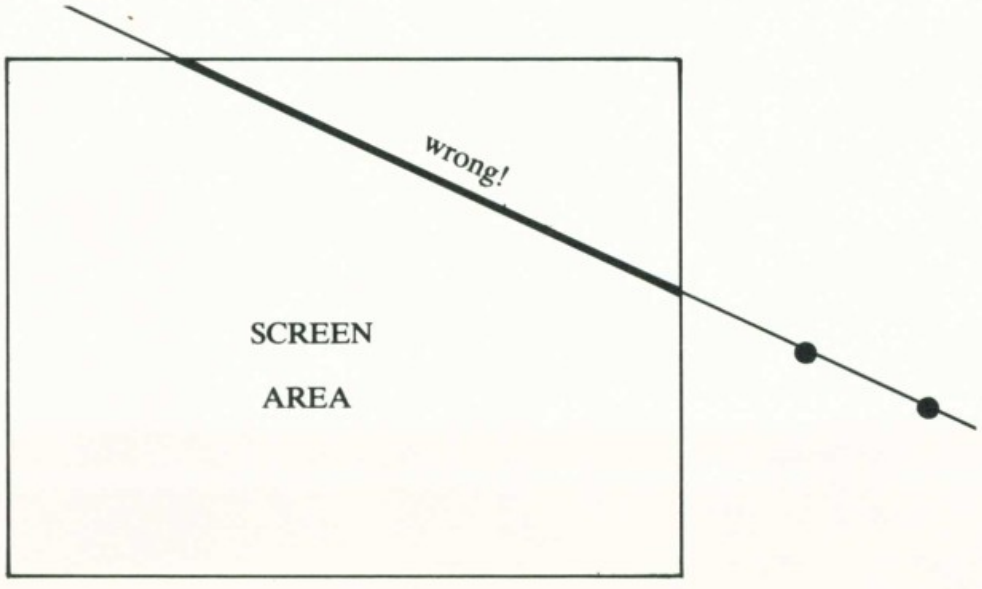


Figure 16.4 The source of the bug.

This is easily fixed, using yet another subroutine:

```
33   LET check = 4000
4000 REM check
4010 LET segon = 0
4020 IF x1 = x2 AND SGN (y1 - y (1) ) =
     SGN (y2 - y (1) ) THEN LET segon = 1
4030 IF x1 < > x2 AND SGN (x1 - x (1) ) =
     SGN (x2 - x (1) ) THEN LET segon = 1
4040 RETURN
```

This sets up a flag *segon*, which is 1 if the situation in Figure 16.4 obtains. To get into it, add:

```
3825 GO SUB check: IF segon THEN RETURN
```

On repeating the test with the parabola . . . it worked! Finally, a more stringent test:

Option	1
Window	−.7 .7 −.3 .6
Parameter range	0 2.1
Number of steps	300
Functions of t	SIN (11 * PI * t) COS (13 * PI * t)

This is a Lissajous figure (*Easy Programming*, p. 111) but the window is chosen so that it repeatedly goes off the screen and comes back on again. The result is shown in Figure 16.5 and it's pretty clear that the program does what it's supposed to do.

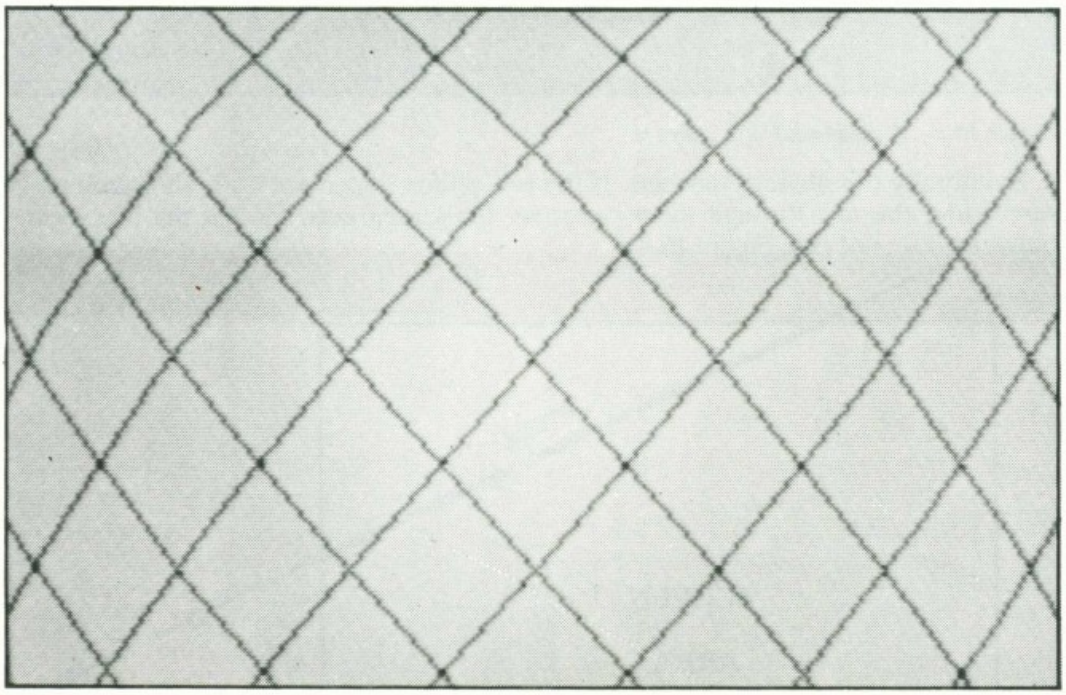


Figure 16.5 A Lissajous figure going on/off screen a great many times, making an excellent test.

This is a genuinely useful program. Try your own choices of option, windows, and functions, starting with the suggestions below. If in doubt, use the functions above and just vary the other numbers, one by one. A good source for ideas is *A Book of Curves* by E.H. Lockwood, Cambridge University Press.

It's clear that some improvements are still possible. In several places there are very similar chunks of code that are used more than once—no doubt a subroutine would shorten the listing. An option to re-run while changing only a chosen variable would be useful. You may even be able to devise a more clever approach to the whole problem: one trouble with top-downing is that if you select a bad strategy to start with, you tend to be stuck with it.

You can also add extra routines. Do you want axes drawn? Scales marked? Several curves superimposed? I'll leave these as projects for those so inclined.

SUGGESTIONS FOR CURVES

Option 2: Graph

(a)	Catenary: window	−5	5	−10	100
	EXP t + EXP (−t)				
(b)	Cissoid: window	−5	5	−2	2
	t / SQR (10 − t)				
(c)	Neile's parabola: window	−10	10	−1	10
	(t * t) ↑ (1 / 3)				
(d)	Serpentine: window	−10	10	−1	1
	2 * t / (4 + t * t)				
(e)	Strophoid: window	−1.5	2	−5	1
	t * SQR ((2 − t) / (2 + t))				

Option 1: Parametrized curve

(f)	Cochlioid: window	−20	20	−20	20
	parameter range	−30	30		
	number of steps	500			
	t * SIN t * COS t				
	t * SIN t * SIN t				

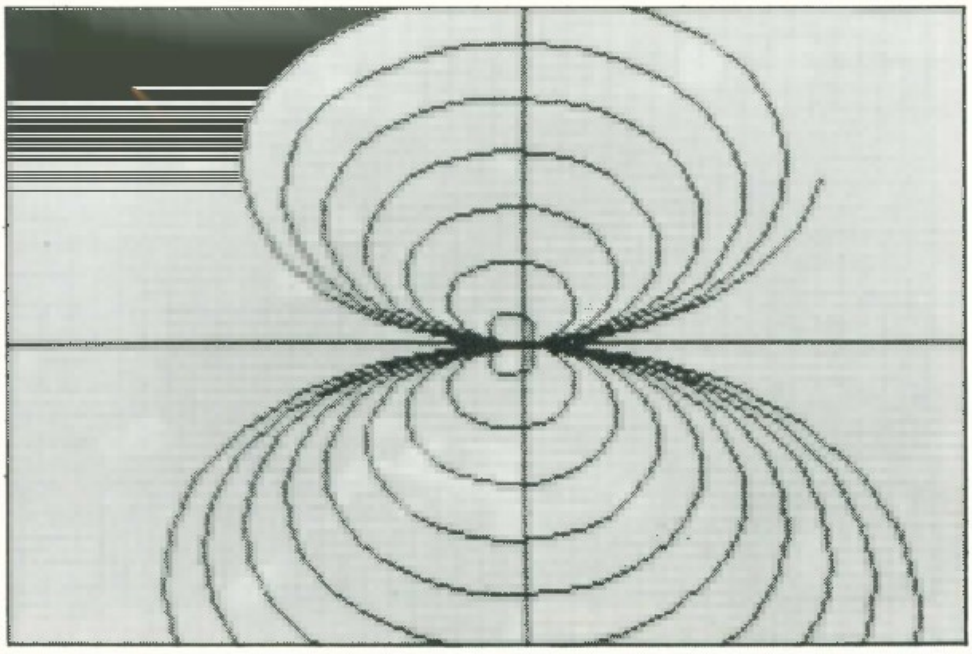


Figure 16.6 The cochlioid.

(g) Limaçon: window −10 10 −10 10
range −PI PI
steps 100
(3 + 5 * COS t) * COS t
(3 + 5 * COS t) * SIN t

(h) Rosace: window −1.2 1.2 −1.2 1.2
range 0 PI
steps 500
COS (11 * t) * COS t
COS (11 * t) * SIN t

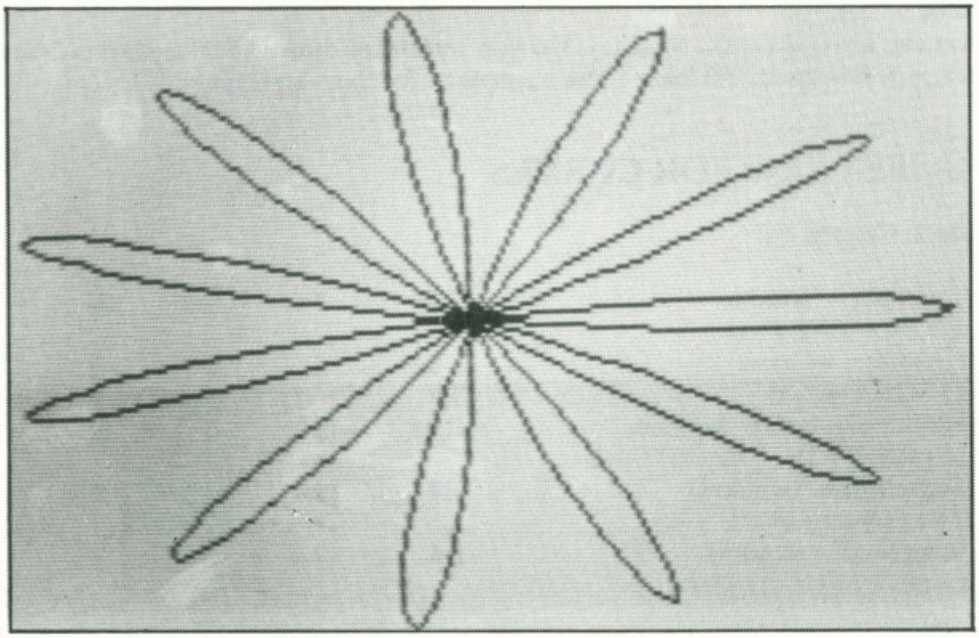


Figure 16.7 An 11-petalled rosace.

(i) Pseudo-Lissajous: window −10 10 −10 10
range −5 5
steps 500
10 / (1 + t * t) * SIN (3 * t)
10 / (1 + t * t) * SIN (5 * t)

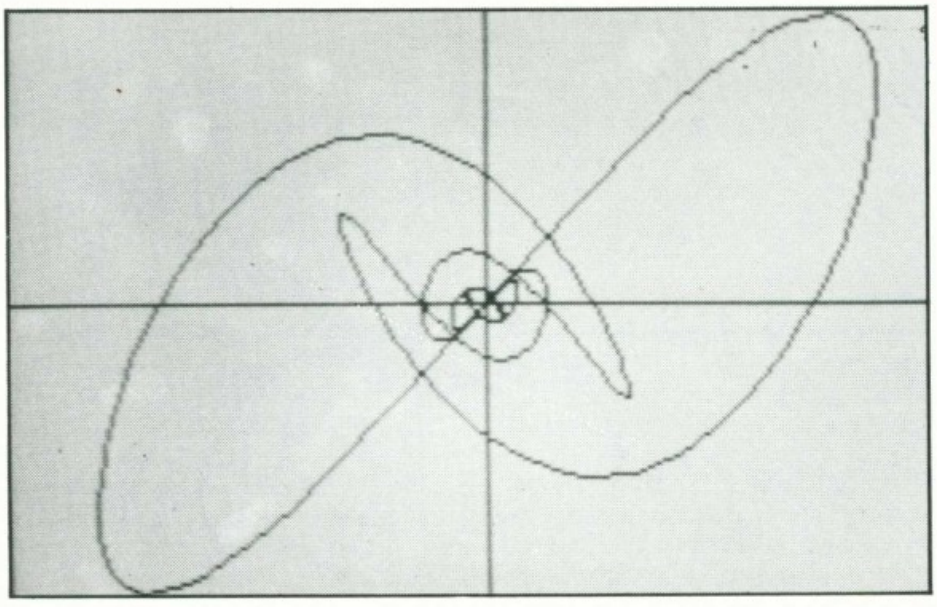


Figure 16.8 An elegant pseudo-Lissajous figure.

Most files have a lot in common.
Why not treat them all alike?

17 Data Management Systems

Let's develop the ideas on files (Chapter 9) further. There, it was assumed that before create, maintain and search programs can be written, we have to know the exact features of the file we're dealing with. And yet all files are going to look pretty much the same. Just the number of fields per record and their lengths will vary. So it wouldn't be difficult to modify *inrec*, for example, changing all those constants (30, 36, etc.) for variables. As an eminent Irish computer scientist has remarked: "All your constants should be variables".

Hang on, though. How will *inrec* know what the field lengths are in a particular case? Well, why not allow *initcfs* either to ask for them from the user when a file is being created, or to read them from the header block otherwise? The form of the header block is now getting a little more complicated, so let's look at it in some detail.

THE HEADER BLOCK

Basically, it's going to be two arrays. One contains a user-supplied name for each field (so it's a string array), and the other holds the number of bytes allocated for each field. We no longer need to know the number of bytes per record, because that's the sum of all the field lengths, but we still need the number of records per block, so let's build that into the numeric array. There's one other consideration. Fields can contain either character or numeric information, and will probably need to be handled differently in each case. So it would make sense to ensure that fields are distinguished in this way when they're set up. We could have another array with this information but I'm going to tag each field name with a "c" or "n" in the first byte to indicate character or numeric type. So our arrays might look like Figure 17.1.

Here, I'm assuming that the file is to hold a set of bank account transactions. We have a six-character date (held as characters because we aren't going to do arithmetic with it; if you want to, it would be best to have 3 separate numeric fields, nday, nmonth and nyear), a cheque number, and an amount of money (both numbers) and a description field in which up to 25 characters may be put to describe the transaction. Cheque numbers are always 6 digits long (at least, mine are), and 8 characters in the amount field allows you to enter up to £99999.99, which is somewhat on the optimistic side for *my* bank balance. (Notice that a character space is used up by the decimal point.) The number of records per block is 20 and, just so that field names match up with their lengths, n$ (1) is left blank. I'm always happier when there's the odd spare storage element anyway, because it allows some leeway if I've forgotten anything. Since both arrays are 11 elements long, there's room for 10 fields per record. Of course, it presents no problem to alter this if you feel it's a bit restrictive.

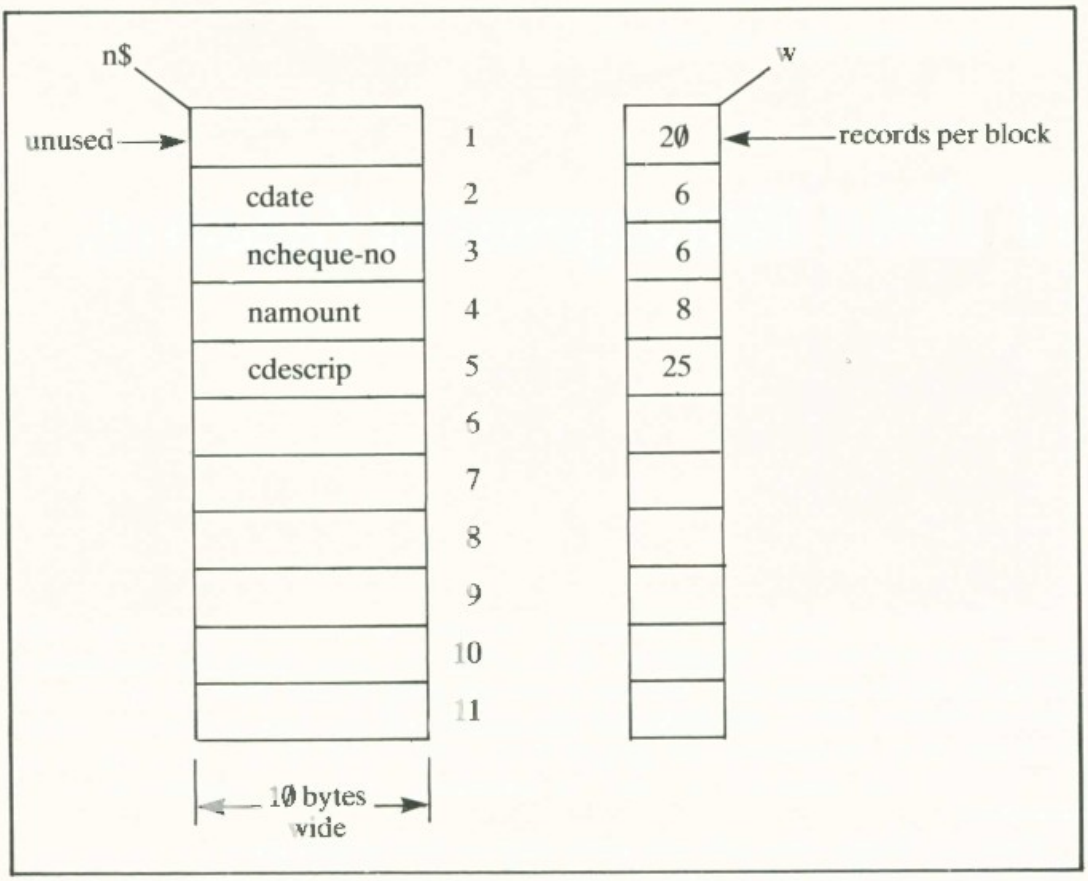


Figure 17.1 Header block layout.

SETTING UP A FILE

initcfs now looks like this:

```
9500  DIM n$ (11, 10): DIM w (11): LET ip = 0: LET op = 0:
      LET inbc = 0: LET outbc = 0: LET bpr = 0
9505  INPUT "Enter input filename:"; f$
9510  IF f$ = "null" THEN GO SUB set up: GO TO 9525
9514  PRINT "Start PLAY recorder"
9515  LOAD f$ + "h1" DATA n$ ( )
9520  LOAD f$ + "h2" DATA w ( )
9521  PRINT "Stop PLAY recorder"
9525  INPUT "Enter output filename:"; g$
9530  FOR p = 2 TO 11
9535  LET bpr = bpr + w (p)
9540  NEXT p
9545  DIM i$ (w (1), bpr): DIM o$ (w (1), bpr)
9547  IF g$ = "null" THEN RETURN
9550  SAVE g$ + "h1" DATA n$ ( )
```

```
9555  SAVE g$ + "h2" DATA w ( )
9557  PRINT "Stop recording": PAUSE 120
9560  RETURN
```

It's much the same as before, except that if we enter "null" as the input filename, it calls *setup* which will generate a new file description. Otherwise, it loads a file description from the first two blocks of the input file (which, for a file called "fred", will be "fredh1" and "fredh2"), works out the number of bytes per record (lines 9530 to 9540) and sets up the input and output buffers from this information. Finally, if the output file isn't "null", it writes the header blocks out.

We'll write *setup* from 9400 on:

```
9400  INPUT "Enter no. of fields:"; nf
9405  CLS
9410  FOR p = 2 TO nf + 1
9415  PRINT AT 10, 2; "field □"; p − 1
9420  INPUT "Name of field (1st char:c/n):"; n$ (p)
9425  INPUT "No. of bytes:"; w (p)
9430  NEXT p
9435  CLS
9440  INPUT "No. of records per block:"; w (1)
9445  RETURN
```

There's nothing much to comment on here. *setup* simply executes a FOR loop once for each field, entering the field name and length of the appropriate array elements. There's a slight fiddle necessary to account for the fact that field 1 actually occupies array element 2, of course. Finally, the number of records per block is specified and entered into w (1).

Now we're in the home straight. We can rewrite *inrec* using the fixed format version to give us clues as to how to tackle this, more general purpose, routine.

The old one started:

```
8000  DIM a$ (336)
```

The 336 was the length of a record. Now *initcfs* has evaluated this, and put it in bpr. So we have:

```
8000  DIM a$ (bpr)
```

The next line was:

```
8010  INPUT "Artist"; a$ (TO 30)
```

So, in more general terms, what we would like to do is to have a loop in which the equivalent line says something like:

```
INPUT "next field name"; a$ (beginning of field TO end of field)
```

and this gets repeated for every field. You can't write something like:

```
10  LET p$ = "next value"
20  INPUT p$; nv
```

because, although it's perfectly valid BASIC, its meaning is different from what we want. Line 20 does not say "prompt with whatever's in p$ and then accept a value into nv". It says "accept a string into p$ and then a value into nv". However, if you put *brackets* around p$, this produces the desired effect.

SCREEN DISPLAY

While this arrangement is conveniently simple, it can be a little confusing for the user. He only sees one field of a record at a time, and it would be helpful if he saw the *whole* record being built up. Also he wouldn't have any indication about the *length* of field he could use.

The simple solution is to build up a record on the screen by using PRINTs for prompts and copying each INPUT value on to the screen to match with its associated prompt. This will be a familiar technique to you if you cut your teeth on a ZX81 which didn't allow prompt strings in INPUTs!

So we would like the screen display to look something like this for the bank account example:

```
d:   date          060482
n:   cheque-no.    317462
n:   amount        -71.37
c:   description   Office armchair: tax ded.
```

where the underlined items are entered as INPUT values and immediately redisplayed as shown. The underlines will actually be printed, and will show the maximum width of each field.

Note several things: firstly, the field type has been separated out from the field name and is there just as a reminder to the user. Secondly, the amount is shown as negative, and you'll see why this convention is useful later. Thirdly, the user makes a note in the description field that the item is tax deductible. He's not using the system very sensibly, because he's bound to want a list of deductible items at the end of the year, and that means doing a search on part of a field. He should have thought of this before he started and used a fifth field to identify deductible items.

Anyway, back to the problem:

```
8010  CLS: LET begin = 1                    [clear the screen so the record
                                             display isn't cluttered, set
                                             pointer to start of a$ . . .
8020  FOR p = 2 TO 11                        and enter loop]
8025  IF n$ (p, 1) = "□" THEN GO TO 8120     [test for last field]
8030  PRINT AT p, 0; n$ (p, 1); ":";
      AT p, 4; n$ (p) (2 TO)                 print
8040  FOR c = 1 TO w (p)                     field
8050  PRINT AT p, 14 + c; "_"                template
8060  NEXT c
```

Now we need to know where the beginning and end are of the slice of a$ that this field occupies:

```
8070  LET end = begin + w (p) - 1
```

First time round, begin = 1, because it was set up in line 8010 and end =6, which means we can write:

```
8080  INPUT (n$ (p) (2 TO) ); a$ (begin TO end)
```

and the effect will be to prompt with the word "date" and transfer this to a$ (1 TO 6). Now we put this on the screen display:

```
8090  PRINT AT p, 15; a$ (begin TO end)
```

Finally we must set up the new "begin" position:

```
8100  LET begin = end + 1
```

and close the loop:

```
8110  NEXT p
```

Pass the result to r$ and exit from the subroutine:

```
8120  LET r$ = a$
8130  RETURN
```

TESTING

At this stage we can write a couple of test routines to see if it all works. First we need to create a file. The routine used to create the record collection file will do without modification. Just to remind you:

```
100  GO SUB initcfs
110  GO SUB inrec
120  GO SUB write
130  INPUT "Any more? (y/n)"; q$
140  IF q$ = "y" THEN GO TO 110
150  GO SUB close
160  STOP
```

When you run this, *initcfs* will ask for an input filename, which you'll enter as "null", of course, and then you'll be prompted for a record description. You could use the bank account one as a fairly simple example. Make the block size small, 5 say, so that you don't have to enter too many records before the blocking mechanism is activated. That way everything gets checked without too much keyboard pounding. Finally, enter 10 or 15 records, to make up the test file.

Now we need to know whether the data have been saved correctly. Replace lines 110–160 with:

```
110  GO SUB read
120  IF r$ (TO 2) = "}}" THEN STOP
130  PRINT r$
140  GO TO 110
```

When you run this you'll get records like:

```
12Ø88212345921.76 □ □ □ coconuts
```

printed out (if you've used the record format in the bank account example).

Now we know that the first 6 bytes represent a date, so that's 12/Ø8/82, and that that's followed by the cheque number (123459) and an amount (21.76—note the 3 spaces, because there's room for 8 bytes in the record definition) and finally, a description. But outputs like that are hardly user-friendly. So what we really need is a routine which unscrambles r$ into its separate fields. Let's call this *outrec* since it performs the opposite function to *inrec*, and we'll store it from 82ØØ on:

```
82ØØ  LET begin = 1
8210  FOR p = 2 TO 11
8220  IF n$ (p) = "□" THEN PRINT: RETURN
8230  PRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO);
8240  LET end = begin + w (p) − 1
8250  PRINT TAB 15; r$ (begin TO end)
8260  LET begin = end + 1
8270  NEXT p
8280  PRINT: RETURN
```

Unsurprisingly, this routine bears more than a passing resemblance to *inrec*, but there is one important difference. We want the records to scroll through continuously, rather than be displayed at the same position on the screen. If we arranged for the latter, you'd have to be a pretty fast reader to see anything at all! So "PRINT AT" is no good. To get the horizontal tabulation previously provided by a coordinate in a "PRINT AT" statement, I'm using the "TAB" function. If you haven't used this before, you can think about it as equivalent to "PRINT AT" without a row specification. In other words, if you say

```
PRINT AT 2, 15; . . .
```

you are specifying row 2.

```
PRINT TAB 15; . . .
```

gives the same column position, but on the next available print line, *wherever* that is.

Now all we have to do is change line 13Ø to:

```
13Ø  GO.SUB outrec
```

and, when the result is run, every record is displayed, in readable form, on the screen. Of course, you're unlikely to want a complete listing of the file on the screen. It would make more sense to send one to the printer. So we should have a routine called *loutrec* which outputs a record to the printer. It will look identical to *outrec*, except that all the PRINTs will be changed to LPRINTs.

THE GRAND DESIGN

Now we have all the tools we need to build our data management system proper. So we'll take a rest from coding, take a few steps back, and consider the grand design.

We implemented three routines in the original file-handling system: create, maintain and search. We need all these, plus a few extra ones; and this time, we'll link them via a menu from the same main program. So there isn't much point in having "maintain" as one of the options, and then immediately being asked whether it's "add" or "delete" that we want. We might as well have "add" and "delete" as major options.

Our main program looks like this, then:

```
10  CLS
20  PRINT AT 0, 5; "Spectrum Data Manager": GO SUB initcfs
26  CLS
30  PRINT AT 2, 0; "Options are:"
40  PRINT AT 3, 11; "1) create"
41  PRINT AT 4, 11; "2) add"
42  PRINT AT 5, 11; "3) delete"
43  PRINT AT 6, 11; "4) search"
100 INPUT "Enter option no.:"; opt
110 GO SUB 200 * opt
120 GO TO 26
```

Do I hear murmurs of discontent? Is someone out there muttering that one minute I'm saying that we'll need to implement some extra routines and the next I'm only allowing for the ones we've already thought of? It's a fair cop, guv. But notice how easy I've made it to tack on any new routines. We simply add another line to the option listing:

```
44  PRINT AT 7, 11; "5) whatever you fancy"
```

and then put the appropriate subroutine at line 1000 onwards, since the mechanism which steers the main program to the correct subroutine simply multiplies the option number by 200.

So create is at 200
add is at 400
delete is at 600
search is at 800
whatever you fancy is at 1000

and so on.

This is altogether a better approach than to build the system in a cut-and-dried way, not allowing for any expansion. It's only when you've used a system for a little while that you begin to realize its limitations, and wish you'd implemented a routine to rule the world, or whatever. Do things this way, and you can modify the program anytime inspiration (or, more likely, frustration) grips you.

INITIALIZATION

One more thing to note: *initcfs* is called before any of the options is invoked. That saves inserting it in every routine, and responding to it more than once if we want to do several things to the same files. Of course, it means the program must be re-run to establish different files. Unfortunately, it also means a little coding rethink, because *initcfs* does a few things other than just define filenames. It also sets pointers and block counters, and these things *do* have to be reinitialized at the beginning of a second file read. So we'll introduce a further subroutine into *cfs* called *reset* which just zeros these pointers:

```
9970 LET ip = 0: LET op = 0: LET inbc = 0: LET outbc = 0
9980 RETURN
```

Now the first line of *initcfs* can become

```
9500 DIM n$ (11, 10): DIM w (11): LET bpr = 0: GO SUB reset
```

and, of course, line 1 has added to it:

```
LET reset = 9970
```

Finally, we can call *reset* in the main program before looping round to the menu display:

```
120  GO SUB reset
130  GO TO 26
```

THE ROUTINES

create we already have, except it needs renumbering and a RETURN rather than STOP on the end:

```
200  GO SUB inrec
210  GO SUB write
220  INPUT "any more? (y/n)"; q$
230  IF q$ = "y" THEN GO TO 200
240  GO SUB close
250  RETURN
```

add is more of a problem. For one thing, it was previously not implemented separately, and for another, I've already commented on its rather primitive nature. So we'll take the opportunity of rethinking the problem. We need to load all the additions into an array to start with, and then, as the file is read, compare each of them with the current record to decide whether to make an insert or not. Also, we need to flag each potential insert to indicate whether it has been written out yet. For the time being we'll pigeon-hole that problem. So:

```
400  INPUT "how many records?"; nr
410  DIM b$ (nr, bpr)
420  FOR q = 1 TO nr
430  GO SUB inrec
440  LET b$ (q) = r$
450  NEXT q
```

set up array

load additional records into b$

Now, the next bit looks tricky. We have to identify the part of the record to be used as the key. Using the bank account file for example, we could be ordering records by date, cheque number or amount and, as usual, we want to allow the user as much flexibility as possible. So we'll call a subroutine, whose details we'll worry about later, called "extractkey". For the time being we'll simply say what we would like *extractkey* to do for us. It will ask the user for the name of the key field, and provide to the calling routine three values:

```
begin:   the byte of r$ or b$ (p) at the beginning of the key field
end:     "   "   "  "  "   "   "  "  "     end    "   "   "   "
ctype:   this will be 0 if the key is numeric and 1 if it's a string
```

Defining *extractkey* as a subroutine has the usual advantage that we seem to be doing very little work, since we've defined what it does, and can make use of it, before actually having to work out how it does it. Looking ahead a little, though, we can see that there's a second typical subroutine feature present: because *delete* and *search* are also going to need it!

Anyway, for the minute, we'll assume that *extractkey* is available, and work out how to use it:

```
460  GO SUB extractkey
```

SORTING

Next, we'll sort the additional records into order. We couldn't do so before because we didn't know what key to sort on until *extractkey* has been to work:

```
465  GO SUB sort
```

and now we can set a pointer to the first record in b$ to be inserted into the file:

```
470  LET ap = 1
```

Things are fairly straightforward from here in. All we need to do is compare b$ (ap) with the next record from the file (r$). If r$ has the smaller key, then output it, otherwise output b$ (ap). There's a point to watch, though. If r$ is output we need to get the next record from the file, but if an element of b$ is written we just have to bump the value of ap by 1. In either case, we must guard against reading past the end of file or past the end of the array. What happens when the input file or the addition array is exhausted? Different things, unfortunately. So we'll call a subroutine *loosends* which will tie them up.

```
480  GO SUB read
490  IF ap > nr OR r$ (TO 2) = "}" THEN GO SUB loosends: RETURN
```

Now we want to compare the keys of r$ and b$ (ap). Let's extract them into s$ and t$ first:

```
500  LET s$ = r$ (begin TO end): LET t$ = b$ (ap) (begin TO end)
```

Now to compare s$ and t$. But there's a problem. s$ and t$ may be numeric or character keys. If they're actually numeric, but are seen as strings, then 123 won't be seen as identical with 123.0. Worse still, 5 turns out to be greater than □□12!

So we need two more subroutines, *compn* and *compc*, which do numeric and character comparisons respectively. Both of them will return a value called comp which identifies the result of the comparison as follows:

Value of comp	Meaning
−1	s$ < t$
0	s$ = t$
1	s$ > t$

You may be wondering why I've specified "=" and ">" conditions when all we really need is "<". The reason is that I've got one eye on the *delete*, *search* and *sort* routines which will also use comparisons, probably in different ways from this one. So it's best to make the routines as general as possible.

Now we can write:

```
510  IF ctype THEN GO SUB compc
520  IF NOT ctype THEN GO SUB compn
```

You may not have seen this kind of construction before (unless you've read Chapter 3). There doesn't seem to be a test in the "IF". The point is that a condition in an "IF" statement is evaluated to 0 or 1 depending whether it is false or true. So if you write:

```
75  IF a = b THEN . . .
```

the "a = b" is replaced by 1 if their values are the same, and by zero otherwise. Since ctype is given the value 0 or 1, we're just saving one process in the evaluation, and the result reads more nicely than "IF ctype = 0 THEN GO SUB compn".

Now to test comp. If comp is less than zero we want to output r$ and get the next file record to replace it:

```
530  IF comp < 0 THEN GO SUB write: GO TO 480
```

Otherwise, we have to output the current additional record. But don't forget that we can

only output what's in r$, and its current contents will have to be saved and restored before and after this process:

```
540  LET t$ = r$: LET r$ = b$ (ap): GO SUB write: LET r$ = t$:
     LET ap = ap + 1: GO TO 490
```

Note the "GO TO 490" on the end. We don't go back to 480 because we haven't flushed the current file record out yet, so we don't need another one!

MORE SUBROUTINES

Now comes the reckoning. The subroutines called with reckless abandon from the *add* routine will have to be written. We'll give them the following line number starting points and, of course, these will have to be initialized at the beginning of the program:

extractkey	7800
loosends	7600
compc	7400
compn	7200
sort	7000

extractkey has first to ask the user which field is the key:

```
7800  DIM k$ (9): INPUT "Enter key field name"; k$
```

k$ will be a field name from the second byte on. In other words, it will *not* include the type character. It seems more natural to enter "amount" than "namount", and the "n" isn't essential, since we already have that information in n$.

Now we have to search through the array n$ looking for k$, but we also have to keep track of how many bytes along we've got. So let's adopt this procedure: assume it's the first field we're after, so begin = 1. If it's not, bump begin by w (2), so it's now pointing to the start of the second field. If that's not the one we want, bump begin by w (3) and so on.

```
7810  LET begin = 1                              (note: k$ must be 9
7820  FOR p = 2 TO 11                            bytes long because it's
7830  IF n$ (p) (2 TO) = k$ THEN GO TO 7860      being compared with the
7840  LET begin = begin + w (p)                  last 9 bytes of every
7850  NEXT p                                     element of n$)
7860  LET end = begin + w (p) - 1
```

Now we can identify the type of the field by looking in the first byte of the element of n$ being pointed at by p:

```
7870  IF n$ (p, 1) = "c" THEN LET ctype = 1
7880  IF n$ (p, 1) = "n" THEN LET ctype = 0
7890  RETURN
```

Actually quite painless, wasn't it!

Now for *loosends*. If we've reached the end of the file first, we have to flush the addition buffer. Otherwise, we have to copy the rest of the file. So it's pretty straightforward:

```
7600  IF ap > nr THEN GO SUB cpyfile: RETURN
7610  FOR p = ap TO nr
7620  LET r$ = b$ (p)
```

```
7630  GO SUB write
7640  NEXT p
7650  GO SUB close
7660  RETURN
```

and if *cpyfile* is at 7700:

```
7700  GO SUB write
7710  GO SUB read
7720  IF r$ (TO 2) = "}}" THEN GO SUB close: RETURN
7730  GO TO 7700
```

compc and *compn* are pretty easy:

```
7400  IF s$ < t$ THEN LET comp = -1
7410  IF s$ = t$ THEN LET comp = 0
7420  IF s$ > t$ THEN LET comp = 1
7430  RETURN

7200  IF VAL s$ < VAL t$ THEN LET comp = -1
7210  IF VAL s$ = VAL t$ THEN LET comp = 0
7220  IF VAL s$ > VAL t$ THEN LET comp = 1
7230  RETURN
```

THE SORTING ROUTINE

Finally, we need *sort*. If you've read *Easy Programming* you'll remember I introduced a bubble-sorting algorithm in the Debugging chapters. We'll use it again here. It's not the most efficient or elegant sorting procedure ever devised (quite the reverse, in fact) but it *is* simple and since b$ isn't a huge array, it won't take too long to execute.

```
7000  LET inc = 1
7005  LET flag = 0
7010  FOR p = 1 TO nr - inc
7020  LET s$ = b$ (p) (begin TO end): LET t$ = b$ (p + 1) (begin TO end)
7030  IF ctype THEN GO SUB compc
7040  IF NOT ctype THEN GO SUB compn
7050  IF comp > 0 THEN LET t$ = b$ (p): LET b$ (p) = b$ (p + 1):
      LET b$ (p + 1) = t$: LET flag = 1
7060  NEXT p
7070  IF flag > 0 THEN LET inc = inc + 1: GO TO 7005
7080  RETURN
```

So, in retrospect, *add* has required rather a lot of effort. But it's probably the trickiest of the menu routines.

DELETE

We'll freewheel downhill for a bit by writing *delete*.

```
600  INPUT "How many keys"; nr
610  GO SUB extractkey
620  DIM d$ (nr, end - begin + 1)
630  FOR p = 1 TO nr
640  INPUT "Enter deletion (key only):"; d$ (p)
650  NEXT p
660  GO SUB read
670  IF r$ (TO 2 ) = "}}" THEN GO SUB close:
     RETURN
680  FOR p = 1 TO nr
690  IF r$ (begin TO end) = d$ (p)
     THEN GO TO 660
700  NEXT p
710  GO SUB write
720  GO TO 660
```

(find key field limits and use this to dimension correct size array)

set up array of keys to delete in d$

search for a match between the key of r$ and an entry in d$. If one is found, get next record.

No match found, so write out the record . . . and get another one.

Onward, ever onward . . .

SEARCH

Before rushing in where angels fear to tread, we should give some serious thought to how the *search* routine ought to behave. The simplest thing to do, would be to allow the user to enter a single key, and then read through the file, displaying on the screen every entry with that key. To decide whether this will be adequate, try to put yourself in the position of the user and imagine the kinds of questions he might want to ask. Let's return to the bank account file as a convenient example. If there's a field which indicates whether an item is tax deductible, then the user might well wish to display all records with this field set to "yes". On the other hand, he might want to display all records referring to cheques drawn for more than £200, or all cheque numbers greater than 318472, or cheques between 8th July 1981 and 5th September 1981. In other words, he is very likely to want to consider a *range* of keys rather than just one. So we should allow both these facilities. Secondly, will he *always* want to display records found by the search? Certainly, he is at least as likely to want them printed. But there's another possibility which will dramatically improve the usefulness of the system without any significant perspiration (well, extra perspiration) on our part. It is to allow records found by the search to be written to a new file. This way, successive searches can be used to isolate combinations of conditions. For instance, if we need a list of all tax deductible items over £100 in value, we first generate a new file of tax deductible items, and then we search this for all items over £100.

I said this was simple to arrange; all it entails, having found a target record, is to call *outrec* to display it, *loutrec* to print it, or *write* to output it to file.

So let's get to it:

```
800  GO SUB extractkey
810  IF ctype THEN DIM k$ (end - begin + 1): INPUT "Target key:"; k$
820  IF NOT ctype THEN INPUT "Key range—low:"; low, "high:"; high
```

A little explanation is needed already. Obviously, the first job is to find out which field is to be used as the key for the search, hence the call to *extractkey*. Now, if the chosen key is a character field, the concept of a range is pretty meaningless (unless you want to deal with a range of alphabetic keys, like all names between BROWN and ROGERS, but I'm not allowing for that here. It's easy enough if you need to do it.) So we only ask for a single target key (which must match the length of the corresponding field in the record—hence the DIM!). If, however, the key is numeric, we ask for a range of keys.

Next, we need to know which output option is to be selected:

```
830  INPUT "Display (1), Print (2) or File (3):"; opt
```

Now we can start the search:

```
840  GO SUB read
850  IF r$ (TO 2) = "}}" AND opt = 3 THEN GO SUB close: RETURN
860  IF r$ (TO 2) = "}}" THEN INPUT "Enter c to continue"; k$: RETURN
```

That's a slightly untidy piece of code. The problem is that, on identifying the end of file there are two possibilities; if there's no output file, we want just to return to the menu, but if there is one, we have to close it first. The alternative would have been either to jump outside the loop on identifying the end of file marker, and then test for option 3, or to call a subroutine to handle the two conditions. I don't like jumping unless I'm absolutely forced into it (it can easily produce code which can most kindly be described as baroque) and a subroutine seemed a bit like overkill, here. (The problem arises because Spectrum BASIC does not have an ELSE clause to it's IF statement. Some BASICs allow you to say: IF this THEN that ELSE the other. This is quite useful sometimes, because the Spectrum equivalent is to have two IF statements. It's already occurred several times, although this is the clumsiest example so far.)

COMPARISONS

Enough of this nitpicking. (The BASIC is lovely, really—honest, Uncle Clive!)

```
870  IF ctype THEN GO SUB ckeytest
880  IF NOT ctype THEN GO SUB nkeytest
```

Now we really *do* need a couple more subroutines, because the method of handling the character and numeric comparisons is going to be significantly different. *ckeytest* will do the string comparison, and *nkeytest* will do the numeric one. All they need to return is a single value "match" which is 1 if a match was found and zero otherwise. Then we have:

```
890  IF NOT match THEN GO TO 840          get next record
895  LET bs = begin: LET es = end
900  IF opt = 1 THEN GO SUB outrec
910  IF opt = 2 THEN GO SUB loutrec       output record to
920  IF opt = 3 THEN GO SUB write         appropriate device
925  LET begin = bs: LET end = es
930  GO TO 840                            get next record
```

Note that we have to save "begin" and "end", because they are altered by *outrec*.

We now have two little routines to write:

```
ckeytest  6800
nkeytest  6600
```

ckeytest is easiest, since there's no range to worry about, only a single key:

```
6800  LET match = 0
6810  IF k$ = r$ (begin TO end) THEN LET match = 1
6820  RETURN
```

Now for *nkeytest*:

```
6600  LET match = 1
6610  IF VAL r$ (begin TO end) < low OR
      VAL r$ (begin TO end) > high THEN LET match = 0
6620  RETURN
```

So in *ckeytest* I've assumed there is no match, and then set match to 1 if there is one, whereas in *nkeytest* I've assumed there *is* a match and reset match to zero only if the key of the test record is less than the lowest key in the range or greater than the highest key in the range.

Now I have a confession to make. The delete routine only really works for c type keys, because line 690 does a straight string comparison. You probably wondered about this at the time.

The reason I chose to do this is that *search* provides a more powerful way of performing deletions on numeric keys. After all, searching for a range of keys is exactly the opposite of deleting those keys (using the "write" option in *search*), so for example, to delete all records with keys below 50, we just search for records with keys 50 or greater. You could provide more complex range tests very simply by expanding *nkeytest* to include a second range, low2 to high2, say. That way, you could delete all records with keys between two values. (This can be done with the current implementation, but it means generating two subfiles.)

MORE FUNCTIONS

What other options might be necessary, or at least desirable? Well, how about *list*, which would copy the entire input file? This could be useful, but usually we can get away without it by using search, choosing a numeric key, and giving a range which we know exceeds the actual range of values in the key field. Since we know the field width, we can guarantee to be able to do this. Now, you may wish to argue that it is unreasonable to expect the user to indulge in this kind of sleight of hand, and I am inclined to agree; but this suite of routines is beginning to get quite large, and if you've only got a 16K memory, the size of your file buffers is beginning to be squeezed. So we should have a very good reason for wanting any new options. Of course, if you've got a 48K machine, you can go on inventing new and even more esoteric routines as the whim takes you.

There is, however, one more routine which should unquestionably be present. We should be able to add together the contents of a given numeric field in every record. Remember that when I was discussing the bank account example I suggested that debits should be entered as negative and credits as positive? If we could simply sum all those fields it would give us a current balance. Clearly a useful facility!

We'll make this option 5, so we need:

```
44  PRINT AT 7, 11; "5) sum"
```

and so the routine will begin at line 1000, and we'll start by initializing the sum:

```
1000  LET sum = 0
```

Now we find out what key we're to work with:

```
1010  GO SUB extractkey
```

and check that the key is numeric. Otherwise we can't do arithmetic with it:

```
1020  IF ctype THEN PRINT "Key not numeric": PAUSE 120: RETURN
```

Now all we have to do is read each record of the file, adding the contents of the key field into sum at each stage:

```
1030  GO SUB read
1040  IF r$ (TO 2) = "}}" THEN PRINT "Sum of □"; k$; "= □"; sum: INPUT
      "Enter c to continue"; k$: RETURN
1050  LET sum = sum + VAL r$ (begin TO end)
1060  GO TO 1030
```

Only one point needs any comment here. If a RETURN were executed immediately after the PRINT on line 1040, you would have to be rather quick to see anything because of the CLS which lurks at the beginning of the menu display. So the INPUT statement just provides a way of hanging up the system until the user is ready to go on. The PAUSE 120 in line 1020 is there for the same sort of reason. In fact a PAUSE could be used in both cases, because PAUSE 65535 will hang up the machine for around 21 minutes, or until a key is hit, so for all practical purposes, the effect is the same.

TIDYING UP

And that's about as far as I'm going to take you on this particular excursion into file handling. As I've already remarked, adding new routines to the data management system isn't going to be difficult and neither is the revision of existing ones to account for minor modifications in requirements. For instance, when you get your microdrives, it should be very easy to revise the system appropriately and the resulting program will, of course, be much easier to use, since the Spectrum will handle disk control for itself.

Nevertheless, no piece of software is ever perfect, even within limited terms of reference, and I certainly make no grand claims for this one. So I'll just make a few suggestions for revisions that you might like to make that will pretty things up a bit.

First, a point about *cfs* itself. *cfs* requires the user to know about the "}}" delimiter. We're forever writing:

```
IF r$ (TO 2) = "}}" THEN . . .
```

We could build that test into *read* by having a variable called *eof* (for "end of file") which is set to zero in *initcfs* and then set to 1 in *read* if the first two bytes of r$ are "}}". Then a user program could write:

```
IF eof THEN . . .
```

which is altogether neater.

Second, the messages issued to request control of the tape recorders are messy (no pun intended!). It would have been better to write four subroutines *recon*, *recoff*, *playon*, *playoff* so that the same messages are always generated in similar circumstances. Also, it's then easy to revise the messages if necessary, or even to replace them by signals sent to a port to control the cassette motors directly, as I suggested earlier (see Appendix B for details of this). Simple revisions of the messages could include making them flash so that they're more obvious; adding a BEEP for the same reason; adding a PAUSE (so that a RETURN to the menu doesn't clear the message too fast).

Third, there is no check in *setup* to ensure that field names begin with either "c" or "n". Obviously that's vital, because the rest of the system assumes it to be the case. There are other situations of a similar nature where tests ought to be included. (For instance what happens if you give *extractkey* a field name which doesn't exist?)

Finally, at the moment you can't actually exit from the menu. Obviously, it's easily done by having an option 6 (exit) and making line 1200 STOP. I didn't do so earlier, because, of course, every time you add a new option, the exit option value moves up one, and the STOP statement moves up 200.

SORTING A FILE

Just to round off this section finally, I'm going to leave *you* with a problem.

There is one glaring omission from our data manager. Any self-respecting system should allow a file to be resorted on a new key, so that, for instance, we could take a field ordered alphabetically on a "name" field and reorder it numerically on, say, a "telephone no." field.

I've avoided this problem because it's not too easy to solve using only one input tape. In the days when large computer installations commonly used magnetic tape a popular technique was the grandly named "polyphase merge sort".

What happened was that you had two input and two output tapes. You read a record from one tape, and then compared it with records from the other, writing them out to one of the output tapes until one appeared which had a key greater than that of the reference record. Then you switched input and output tapes and repeated the process. When both input files had been read completely, you used the output files as a new set of input files, and the whole process was repeated. Eventually, after umpteen repetitions, only one of the output files is written to, and the switch never takes place. Then you know you've got a sorted file.

Sounds complicated? That's what I think; and now you know why I haven't addressed the problem till now. (Although, in a sense, the addition routine uses a kind of merge sort, using the input file and the addition buffer as equivalent to two input tapes; the difference is that these two inputs are guaranteed to be separately sorted before we start.)

So, is it *possible* to do a tape file sort with only one input tape? Well, yes it is, and there are several well-known algorithms. But there's one that I discovered recently which will suit our purposes very well, because it enables us to use the block structure built into *cfs*. (I say "discovered" rather than "invented" because, although I've never seen it described in print, it's such a simple technique that I'm sure it must have been used before.)

RATCHET SORT

It works like this:

We pull in the input file, block by block, bubble-sorting each block before writing it out to the output file. Now, if that were *all* we did, nothing very exciting would happen because, although the file would be locally sorted, there could easily be a low key at the end of the file which cannot move back beyond the beginning of the last block however many times the process is repeated. In fact, *nothing* new will happen if we use this technique to resort the output file. The reason is, of course, that the partitions between blocks remain in the same place, so records can only move within, not between, blocks. If only we could change the positions of the block boundaries . . .

Actually, it's easy. Before transferring the first block to the output file, we write a number of dummy records to it. For best results, this number should be half the number of records per block. On reading the newly located file, the block boundaries are now half-way between their previous positions, and this "overlap" allows further sorting to take place. We must ensure that the dummy records are ignored by the sort, and that they are suppressed on output so that the block boundary positions are again changed to allow another sort operation. On the next phase of the sort, the dummies are reinserted, and so on, and so on. We know we've finished when no further swaps take place.

Here's an example which illustrates the principle. I'll just write the keys for simplicity, as the numbers 1–12 in reverse order. We'll assume a block size of 4:

12	11	10	9	8	7	6	5	4	3	2	1

1st pass: Sort blocks and insert 2 dummy records:

D	D	9	10	11	12	5	6	7	8	1	2	3	4

2nd pass: Sort blocks and suppress dummies:

9	10	5	6	11	12	1	2	7	8	3	4

3rd pass: Sort blocks and insert dummies:

D	D	5	6	9	10	1	2	11	12	3	4	7	8

4th pass: Sort blocks and suppress dummies:

5	6	1	2	9	10	3	4	11	12	7	8

5th pass: Sort blocks and insert dummies:

D	D	1	2	5	6	3	4	9	10	7	8	11	12

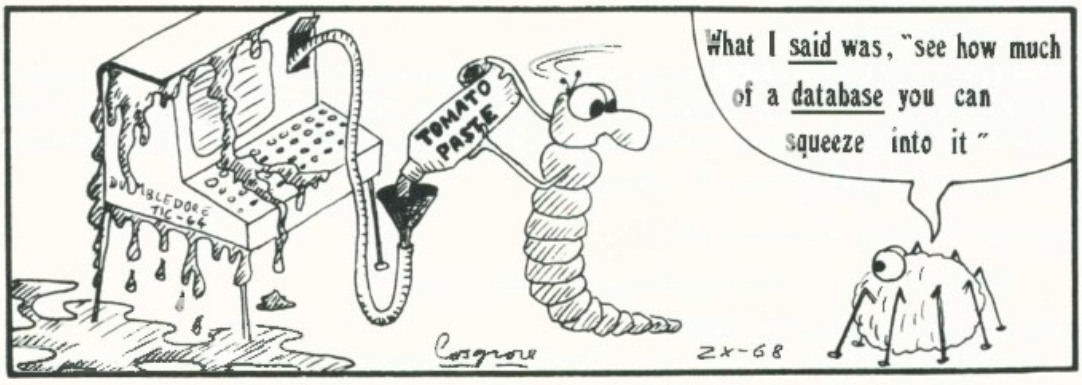
6th pass: Sort blocks and suppress dummies:

1	2	3	4	5	6	7	8	9	10	11	12

and we're there!

Provided there's enough spare memory, there's no reason why the block used for sorting should be the input buffer, in which case, there's no reason why it should be restricted to the block size for the file. Obviously, the number of passes necessary for the sort decreases as the block size increases.

I'll leave the actual coding to you.



If you thought Canopus was a tin of cat-food, read on . . .

18 Star Charts

This item will involve a fair amount of time spent inputting data—especially if you decide to extend it. So don't start unless you've a couple of hours to spare, because it's a nuisance stopping in the middle.

The idea is to write a program that plots out pictures of different constellations—Orion, Cygnus, Gemini, and so forth. It will include as options:

1. An automatic run-through of the pictures, naming them.
2. A search by name for a given constellation, plotting it out.
3. A test: the computer draws out a constellation and the user has to name it.

For this illustration I'll only put six constellations in; but the program will be set up to allow up to 20. If you want more than that, SAVE the data on tape, redimension the arrays used to hold the data, and LOAD the data back. (You'll need to think that through in more detail, but that's the general idea.)

DATA STRUCTURE

Switch the computer off for a minute, because we're going to have to do a little brainwork first. "Program first, ask questions later" is a recipe for disaster.

We will need data for:

1. Latin name of constellation (Ursa Major, etc.).
2. English name of constellation (Great Bear).
3. Positions of the stars (lots of coordinates for PLOT).
4. Magnitude (degree of brightness) of each star.

You could go further (e.g. colour of star, suitably exaggerated for a pretty display), but these will do to start with.

Leaving aside for a moment the problem of actually laying hands on the required information, and getting it into the machine, we must first decide how to format the data. Obviously we want to hold the names in arrays; so for 20 different constellations we need to set up two string arrays of size 20, say n$ for the Latin name and e$ for the English name.

We'll also want to print out things like

```
Cygnus (the Swan)
```

Now the problem with string arrays is that all of the strings in them have a fixed length. Suppose we've set that length to 12, say, to allow for names like "Camelopardus" (the Giraffe); and that Cygnus is item number 1. It's not very satisfactory to use the obvious

```
PRINT n$(1); "□ (the □"; e$ + ")"
```

because you'll get something like

```
Cygnus □□□□□□ (the Swan □□□□□□□□)
```

with the spacing all up the shoot.

There's a useful trick to avoid this. To each string we add a final, thirteenth character, whose *code* gives the actual length that we want to use. So "Cygnus" goes in as

where the ☆ is actually the character whose code is 6, the length of "Cygnus". (This is the *control* character PRINT comma, but everything's fine as long as we don't try to print it out.) To print out the name "Cygnus", we use

```
PRINT n$ (1) (TO CODE n$ (1, 13) )
```

which omits the unwanted spaces.

We use the same trick on the English names e$, of course; and we have to be careful to write the input and output routines to take this feature into account.

The obvious way to set up the coordinates for the stars within a constellation, and their magnitudes, is as an array. It needs one dimension of 3 to allow for horizontal co-ordinate, vertical coordinate, and magnitude; one dimension of (say) 20 to allow up to 20 stars per constellation; and one dimension of 20 to say which constellation we're thinking of. That leads to a command

```
DIM p (3, 20, 20)
```

However, that would require 3 × 20 × 20 blocks of memory, each 6 bytes long (for a floating-point number), or 1200 × 6 = 7200 bytes. Now a 16K Spectrum has about 9,000 bytes spare after taking care of the display and attribute files, and by the time the program has gone in, and the other data, and room for the machine to do its calculations . . . well, we'll probably run out of memory. Think again.

It's really stupid to store the coordinates in floating-point: we can only PLOT x, y when x is an integer between 0 and 255, and y is an integer between 0 and 175. Now 0–255 is, by a strange coincidence, the range of character codes . . . So, using a single character and taking its code, we can get the x-coordinate. Similarly for the y-coordinate and the magnitude.

Each star thus takes up three characters. Twenty stars take up 3 × 20 = 60, which can conveniently be stuck end to end as a string. For 20 constellations we can use:

```
DIM p$ (20, 60)
```

which takes up only 20 × 60 = 1200 bytes—an impressive degree of shrinkage.

Putting that lot together, we get the following input routine. (The actual program comes later.)

```
9000  REM input routine—can be deleted after use
9010  DIM n$ (20, 13): DIM e$ (20, 13): DIM p$ (20, 60)
9020  FOR i = 1 TO 20
9030  INPUT "Latin name?"; a$
9040  LET n$ (i) = a$: LET n$ (i, 13) = CHR$ LEN a$
9050  PRINT TAB 0; n$ (i) (TO CODE n$ (i, 13) );
9060  INPUT "English name?"; a$
9070  LET e$ (i) = a$: LET e$ (i, 13) = CHR$ LEN a$
9080  PRINT TAB 14; e$ (i) (TO CODE e$ (i, 13) ) '
9090  LET c = 1
9100  INPUT "horiz, vert, mag"; x, y, m
9110  IF x = 0 THEN GO TO 9150
9120  LET p$ (i) (3 * c − 2 TO 3 * c) = CHR$ x + CHR$ y + CHR$ m
9130  LET c = c + 1: IF c > 20 THEN GO TO 9150
9140  GO TO 9100
9150  NEXT i
```

THE DATA

The next step is to get the data into the machine. I'll explain later how to work it out for other constellations; and you may decide to skip over this section for the moment, to see the rest of the program.

RUN the input routine, and key in the following when asked. Each line represents the response to an input prompt: note that you need to ENTER each item on the line separately. The screen format won't be quite the same. ENTER Ø, Ø, Ø to stop.

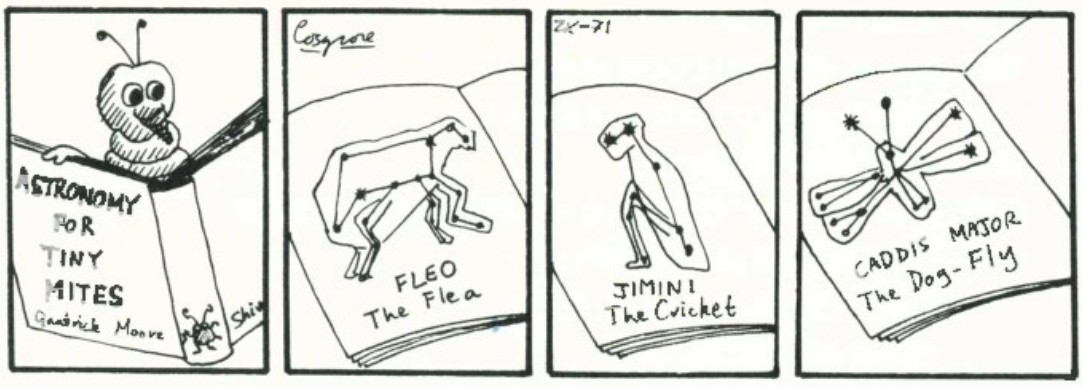
```
Cancer
Crab
   95        54        4
  111       141        4
  113        87        4
  115        96        4
  149        42        4
  156        89        4

Cygnus
Swan
   64       12Ø        4
   72        36        3
   74        85        4
   76        76        4
   91       1Ø7        4
  1Ø4        54        2
  1Ø5        66        4
  114       111        1
  122        36        4
  124        86        2
  139       123        4
  142       118        4
  161        5Ø        4
  168       1ØØ        3
  179       146        4
  188       156        4
  192        26        3

Gemini
Twins
   66       113        1
   68        94        3
   76       1Ø6        4
   78       132        1
   88       111        4
   97        82        3
  1ØØ        54        3
  1Ø5       123        4
  118        76        4
  124       144        3
  14Ø       1ØØ        3
  143        36        3
  152        54        2
  162        76        4
  168        87        3
  178        89        3
  193        94        4
```

Leo
Lion

28	90	2
56	49	4
58	74	4
72	97	3
76	122	2
78	53	4
92	118	4
100	139	4
120	52	4
144	110	2
150	128	3
153	57	1
158	91	3
181	140	4
188	126	3
188	52	4
204	121	4
216	152	4

Orion
Hunter

78	133	4
86	121	1
104	32	2
108	72	2
110	106	4
113	136	3
115	76	2
118	51	3
120	81	2
126	117	2
128	94	4
130	71	3
134	44	4
142	50	3
144	100	4
148	43	1
154	100	3
154	60	3
171	116	4
172	124	3



Ursa Major
Great Bear

27	131	2
56	140	2
73	134	2
94	128	3
104	116	2
106	87	4
124	21	3
126	15	4
133	70	3
136	142	2
137	118	2
173	60	3
176	66	3
176	140	4
180	156	4
183	117	4
200	112	3
211	162	3
225	103	3
226	109	3

Now, if you're still with me, SAVE this lot before you accidentally lose it all. The easiest way is to SAVE the whole thing, loading routine and all; this has advantages if you want to load anything else in later. Or you can use *array storage*: this takes three goes, using:

```
SAVE "Latin" DATA n$ ( )
SAVE "English" DATA e$ ( )
SAVE "Stars" DATA p$ ( )
```

which load back using similar instructions, but with LOAD in place of SAVE.

CHECKS AND CORRECTIONS

If you make a mistake while doing this, all is not lost. Suppose the fifth star in Leo is wrong. Then you type in the direct command

```
LET i = 4: LET c = 5: GO TO 9100
```

type in the corrected version of x, y, m; then STOP. To check your input, use this routine:

```
9500  FOR i = 1 TO 20
9510  PRINT n$ (i) (TO CODE n$ (i, 13) )
9520  PRINT e$ (i) (TO CODE e$ (i, 13) )
9530  FOR t = 1 TO 60 STEP 3
9540  IF p$ (i, t) = "0" OR p$ (i, t) = "□" THEN GO TO 9570
9550  PRINT CODE p$ (i, t); "□"; :
      PRINT CODE p$ (i, t + 1); "□"; :
      PRINT CODE p$ (i, t + 2)
9560  NEXT t
9570  NEXT i
```

Only use this if you're worried about your accuracy: if you've checked the display on screen during the input run, very carefully, it shouldn't be necessary.

Change each PRINT to LPRINT, if you've got a printer, and you'll have a hard-copy listing for reference.

WORKING OUT THE DATA

If you want to add extra constellations, you've got to find out what numbers to input.

The simplest way is to draw out the desired constellation on a piece of graph paper, with a 256 × 176 grid marked on it. (I actually used a 64 × 44 grid and multiplied by 4.) Start with a reference work on astronomy: *Norton's Star Atlas*, published by Gall and Inglis, is a good one. Copy the stars on to tracing-paper, transfer the result to your grid; and laboriously read off the coordinates, checking the star atlas to find the magnitude. See Figure 18.1 for Leo.

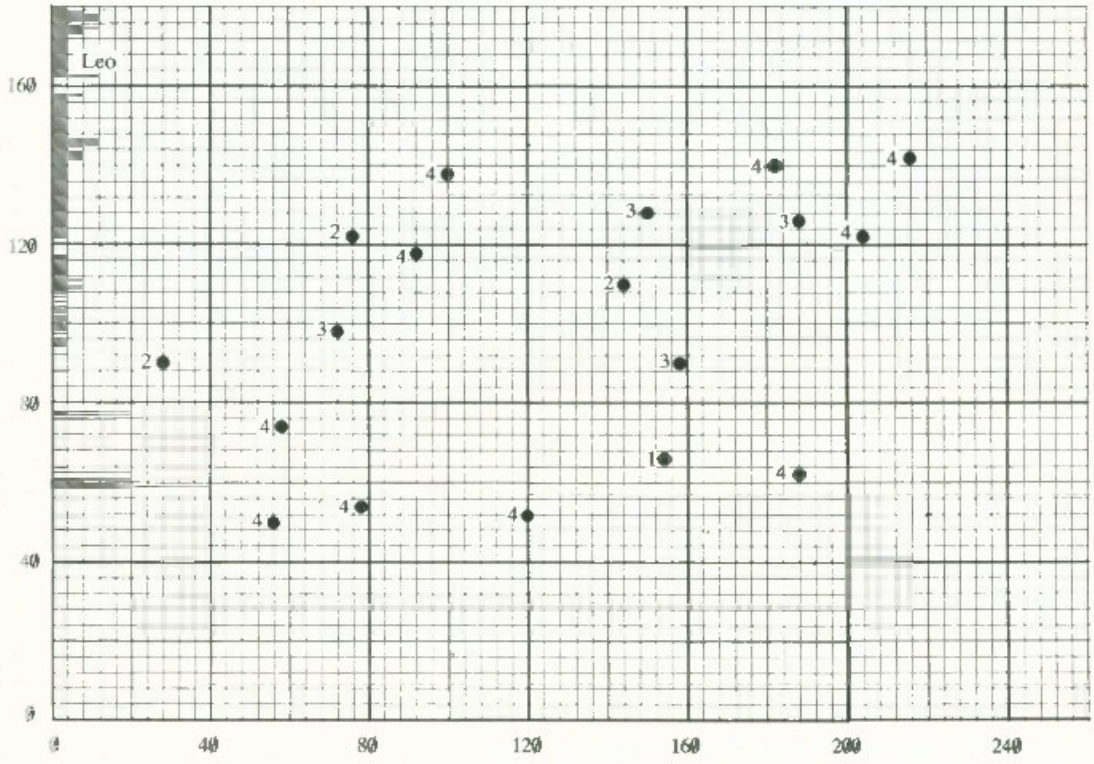
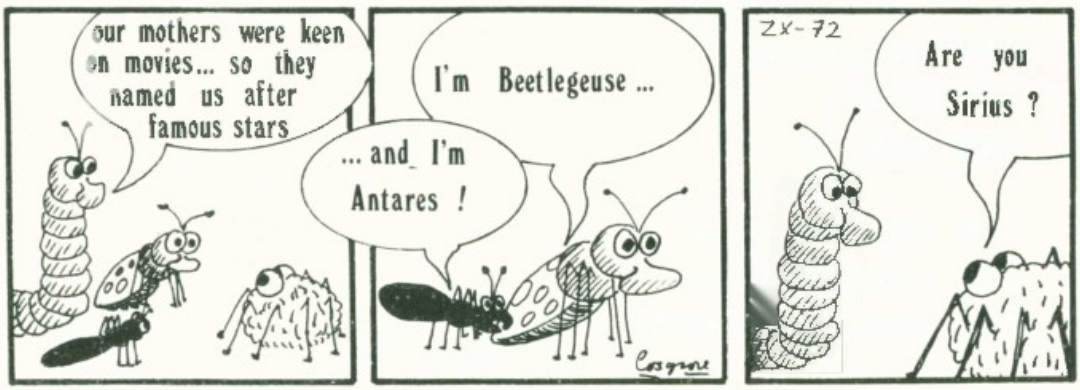


Figure 18.1 Input data for Leo.



Alternatively, draw out the constellation on a transparent sheet and use a version of the sketchpad method used to draw a map in Chapter 1. You'll need to write a little routine to transfer the pixel coordinates to p$, and to input the magnitude suitably. That's a project for you, if you're keen. The system variable COORDS (Chapter 6) may prove useful.

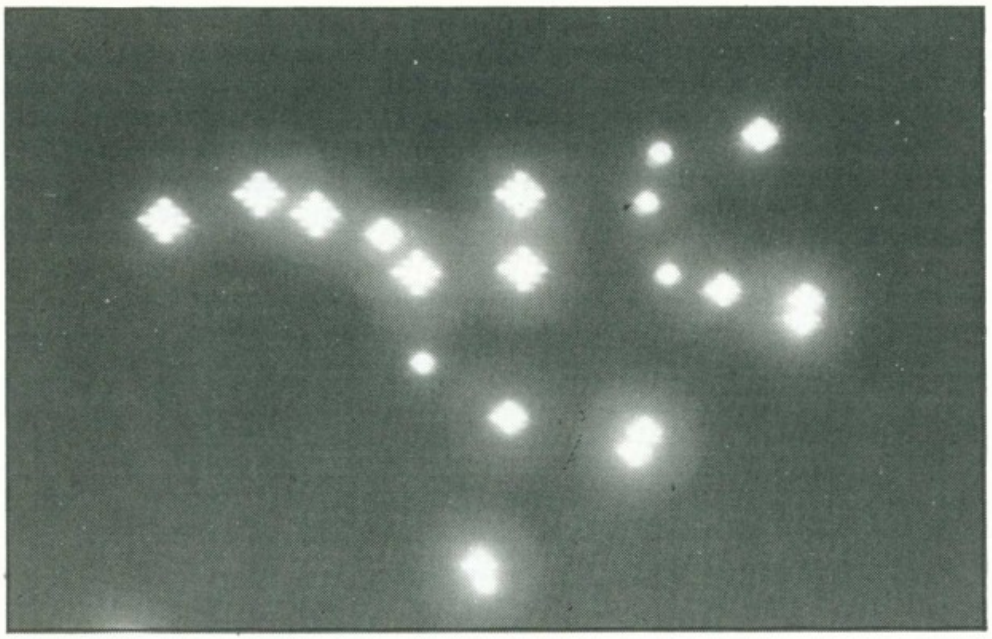


Figure 18.2 Screen display for Ursa Major. See the Plough (Big Dipper) at top left?

THE MAIN PROGRAM

You can delete the loading and checking routines, if you wish. Just *don't type* RUN, or you'll lose the data and have to reload from tape. Use GO TO 1 instead.

Obviously we need a routine to plot out the stars. This one plots a star whose size is determined by the magnitude m, at pixel position x, y. (Magnitude 1 stars are the brightest; then 2, then 3, and so on.) The constellation is number i.

```
8000  REM star plot
8010  PAPER 0: INK 7: BORDER 0: CLS
8020  FOR t = 1 TO 60 STEP 3
8030  LET k = CODE p$ (i, t)
8040  IF k = 32 OR k = 48 THEN RETURN
8050  LET x = k: LET y = CODE p$ (i, t + 1):
      LET m = CODE p$ (i, t + 2)
8060  GO SUB 8500
8070  NEXT t
8080  RETURN

8500  REM draw one star
8510  LET s = 10 - 2 * m
8520  PLOT x, y - s: DRAW 0, 2 * s
```

```
8530 PLOT x − s, y: DRAW 2 * s, 0
8540 PLOT x − s / 2, y − s / 2: DRAW s, s
8550 PLOT x − s / 2, y + s / 2: DRAW s, −s
8560 RETURN
```

Now, we wanted three options: automatic list, search by name, and random test. Let's set up a little menu:

```
100 PRINT "Star Charts" ' '
110 PRINT "Options;
        1. Automatic List
        2. Name Search
        3. Astroquiz"
120 INPUT "Option number?"; opt
```

You're an old hand at this game, so I'll leave the fancy formatting to your personal tastes.

```
130 GO SUB 1000 * opt
140 PAUSE 0: CLS: GO TO 100
```

Now write the options:

```
1000 FOR i = 1 TO 6
1010 GO SUB 8000
1020 PRINT AT 0, 0; n$ (i) (TO CODE n$ (i, 13) );
     "□ (the □"; e$ (i) (TO CODE e$ (i, 13) ); ")"
1030 INPUT "Hit ENTER to continue", d$
1040 NEXT i
1050 RETURN

2000 INPUT "Latin name of constellation?"; q$
2010 FOR i = 1 TO 6
2020 IF n$ (i) (TO CODE n$ (i, 13) ) < > q$ THEN GO TO 2045
2030 GO SUB 8000
2040 PRINT AT 0, 0; q$; "□ (the □"; e$ (i) (TO CODE e$ (i, 13) ); ")"
2045 NEXT i
2050 RETURN

3000 LET i = INT (1 + 6 * RND)
3010 GO SUB 8000
3020 INPUT "Which constellation is this?"; q$
3030 IF q$ = n$ (i) (TO CODE n$ (i, 13) ) THEN
     PRINT AT 0, 0; FLASH 1; "CORRECT!"
3040 IF q$ < > n$ (i) (TO CODE n$ (i, 13) ) THEN
```

```
        PRINT AT 0, 0; FLASH 1; "Sorry, wrong answer":
        PRINT AT 1, 0, FLASH 0; n$ (i) (TO CODE n$ (i, 13) )
3050    PAUSE 100: RETURN
```

To try this out, hit GO TO 1 (remember, *not* RUN), and follow the screen instructions.

If you have put more than 6 constellations in, you'll need to change the 6 in lines 1000, 2010 and 3000 to the new number.

Save this using something like

```
SAVE "Starch" LINE 100
```

and it will then run automatically on loading, avoiding the danger of wiping out your variables.

SOLUTIONS TO CRYPTANALYSIS PROBLEM (page 96)

1. Code abcdefghijklmnopqrstuvwxyz
owgbueplqsznhftkirdjmyvcax

There are nine and sixty ways of constructing tribal lays, and every single one of them is right.
(Kipling)

2. Code abcdefghijklmnopqrstuvwxyz
btgpxeuqfwocvzyasnrlkhidjm

Age cannot wither her, nor custom stale her infinite variety. Other women cloy the appetites they feed.
(Shakespeare)

3. Code abcdefghijklmnopqrstuvwxyz
zvetrsnbjpkcuxdfgayohlqimw

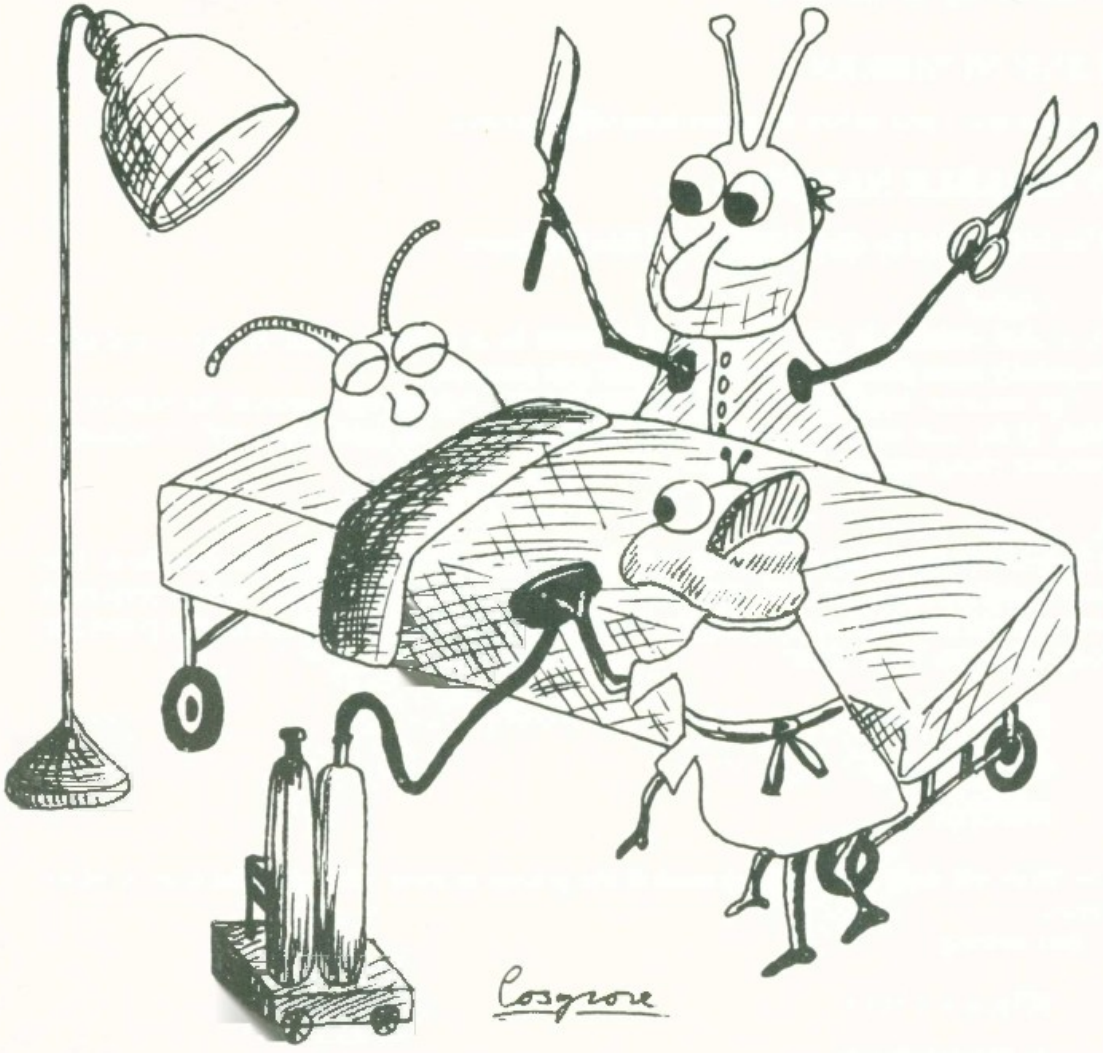
When a man has married a wife he finds out whether her knees and elbows are only glued together.
(Blake)

4. Code abcdefghijklmnopqrstuvwxyz
jkyqzfclotsndxuiegmvwrabhp

Nature's great masterpiece, the elephant: the only harmless great thing.
(Donne)

Of course, you won't be able to work out the entire code: only the letters actually *used*.

Appendices



Appendix A: The Cassette File System— A Reference Description of cfs

Although, in this book, *cfs* has been used solely inside *sdm* (the Spectrum Data Manager), there is, of course, no reason why you shouldn't build it into other programs. To do so, you simply enter all the *cfs* routines, together with their identifiers and starting line numbers, and save it as a program called "cfs". When you've entered the program which uses *cfs*, you simply enter:

```
MERGE cfs
```

and play the "cfs" tape, as if you were doing a normal LOAD. The effect is to combine the program in memory with that on tape, providing a neat way of getting *cfs* (or any other set of utility routines) into any program.

Of course, the main program must not use any of the line numbers that *cfs* uses, and care must be taken not to allow conflicts in the use of variable names (otherwise it's too easy to set q to 1, call a subroutine, and then find that q has mysteriously become 14). This appendix provides the information necessary to avoid such conflicts.

A complete listing of *cfs* is given, together with descriptions of the actions of each routine, and variable names used by it. Note that the listing is *not* identical with that given in the text. The improvements discussed elsewhere have been made, and some renumbering has been done.

LINE NUMBERS

cfs uses line 1 and all line numbers from 9000 upwards.

VARIABLE NAMES

The names used by *cfs* can be divided into four types:

1. *Global*

A global variable is one which may be used by a number of *cfs* routines, and consequently must never be redefined by the user program.

For instance, *initcfs* sets up an array called n$ which holds the names of the fields in the files. If the user redefines it anywhere in his program, all field names will immediately become blank strings!

2. *Local*

These are names which are used by a *cfs* routine, but have no significance outside it. The user program may use such names provided it does not need their values preserved across a call to the *cfs* routine which uses them locally. For instance, *inrec* uses p and c as loop counters. If we write:

```
FOR p = 1 TO 4
GO SUB inrec
NEXT p
```

the loop will only be executed once if the p-loop in *inrec* was executed four or more times!

But writing:

```
FOR p = 1 TO 4
do something else
NEXT p
GO SUB inrec
```

is perfectly legitimate.

3. *Parameters passed to cfs routines*
These are the names of variables which *cfs* is to work with. For instance, r$ must be passed to *write* for its contents to be output.

4. *Parameters returned by cfs routines*
These are the names of variables which a *cfs* routine generates for subsequent use by the user program. For instance, r$ is returned by *read*.

GLOBAL VARIABLES

Name	*Use*
bpr	Number of bytes per record
eof	End of File flag; Ø = not end of file, 1 = end of file
f$	Name of input file
g$	Name of output file
ip	pointer to next record in input buffer
i$( )	2D array acting as input buffer. Size determined by w (1) and bpr
inbc	Current input block number
n$ (11, 1Ø)	array holding names of up to 1Ø fields per record. Each field name may be up to 1Ø bytes long
op	Pointer to next record in output buffer
o$( )	2D array acting as output buffer. Size as for i$
outbc	Current output block number
switch	Set to zero when no recorder is activated; set to 1 if a recorder is activated
w (11)	Array holding the field widths, in bytes, of the fields named in n$. w (1) holds the number of records per block

ROUTINE DESCRIPTIONS

close

Action	Writes file delimiters to output buffer
Parameters passed to close	None
returned	None
Global variables affected	None (but note that r$ is overwritten)
Local variables	None
cfs routines called	write

Listing

```
9740  LET r$ = "}}"
9750  GO SUB write
9760  LET r$ = "cfsend"
9770  GO SUB write
9780  RETURN
```

getblock

Action	Inputs a file block
Parameters passed to getblock	None
returned	None
Global variables affected	inbc, i$, ip
Local variables	m$
cfs routines called	mesp

Listing

```
9800  LET m$ = STR$ inbc
9810  GO SUB mesp
9820  LOAD f$ + m$ DATA i$ ( )
9825  POKE 23692, 255
9830  GO SUB mesp
9840  LET ip = 1
9850  LET inbc = inbc + 1
9860  RETURN
```

initcfs

Action	Initializes the system and sets up file names and descriptions. Dummy files are named "null"
Parameters passed to initcfs	None
returned	None
Global variables affected	n$, w, ip, op, inbc, outbc, bpr, eof, switch, f$, g$, i$, o$
Local variables	p
cfs routines called	setup, mesp, mesr, reset

Listing

```
9500  DIM n$ (11, 10): DIM w (11): LET bpr = 0: GO SUB reset
9505  INPUT "Enter input file name"; f$
9510  IF f$ = "null" THEN GO SUB setup: GO TO 9525
9514  GO SUB mesp
9515  LOAD f$ + "h1" DATA n$ ( )
9520  LOAD f$ + "h2" DATA w ( )
9521  GO SUB mesp
9525  INPUT "Enter output filename"; g$
9530  FOR p = 2 TO 11
9535  LET bpr = bpr + w (p)
9540  NEXT p
9545  DIM i$ (w (1), bpr): DIM o$ (w (1), bpr)
9547  IF g$ = "null" THEN RETURN
9548  GO SUB mesr
9550  SAVE g$ + "h1" DATA n$ ( )
9555  SAVE g$ + "h2" DATA w ( )
9557  GO SUB mesr
9560  RETURN
```

inrec

Action	Prompts for a record entry from the keyboard, field by field, and packs the result into r$

Parameters passed to inrec	None
returned	r$
Global variables affected	None
Local variables	a$, begin, p, c, end
cfs routines called	None

Listing

```
9000  DIM a$ (bpr)
9010  CLS: LET begin = 1
9020  FOR p = 2 TO 11
9025  IF n$ (p, 1) = "□" THEN GO TO 9120
9030  PRINT AT p, 0; n$ (p, 1); ":"; AT p, 4; n$ (p) (2 TO)
9040  FOR c = 1 TO w (p)
9050  PRINT AT p, 14 + c; "_"
9060  NEXT c
9070  LET end = begin + w (p) − 1
9080  INPUT (n$ (p) (2 TO) ); a$ (begin TO end)
9090  PRINT AT p, 15; a$ (begin TO end)
9100  LET begin = end + 1
9110  NEXT p
9120  LET r$ = a$
9130  RETURN
```

mesp

Action	Displays a prompt to turn PLAY recorder on or off
Parameters passed to mesp	None
returned	None
Global variables affected	switch
Local variables	None
cfs routines called	None

Listing

```
9140  PRINT INVERSE 1; "Start" AND NOT switch; "Stop" AND switch;
      "play recorder"
9150  BEEP .2, 15: PAUSE 6: BEEP .3, 20: PAUSE 80
9160  LET switch = NOT switch
9170  RETURN
```

mesr

Action	Displays a prompt to turn RECORD recorder on or off
Parameters passed to mesr	None
returned	None
Global variables affected	switch
Local variables	None
cfs routines called	None

Listing

```
9300  PRINT INVERSE 1; "Start" AND NOT switch; "Stop" AND switch;
      "recording"
9310  BEEP .2, 20: PAUSE 6: BEEP .3, 15: PAUSE 80
9320  LET switch = NOT switch
9330  RETURN
```

outrec

Action	Displays record in r$ in fields on the screen
Parameters passed to outrec	r$
returned	None
Global variables affected	None
Local variables	p, begin, end
cfs routines called	None

Listing

```
9200  LET begin = 1
9210  FOR p = 2 TO 11
9215  IF n$ (p, 1) = "□" THEN PRINT: RETURN
9220  PRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO);
9240  LET end = begin + w (p) − 1
9250  PRINT TAB 15; r$ (begin TO end)
9260  LET begin = end + 1
9270  NEXT p
9280  PRINT: RETURN
```

putblock

Action	Outputs a block to tape
Parameters passed to putblock	None
returned	None
Global variables affected	op, outbc
Local variables	m$
cfs routines called	None

Listing

```
9900  LET m$ = STR$ outbc
9910  GO SUB mesr
9920  SAVE g$ + m$ DATA o$ ( )
9930  GO SUB mesr
9940  LET op = 0
9950  LET outbc = outbc + 1
9960  RETURN
```

read

Action	Read a record into r$

Parameters passed to read	None
returned	r$
Global variables affected	ip, eof
Local variables	None
cfs routines called	getblock

Listing

```
9600  IF ip = 0 OR ip > w (1) THEN GO SUB getblock
9610  IF i$ (ip) (TO 6) = "cfsend" THEN PRINT "Attempt to read past end
      of file": STOP
9620  LET r$ = i$ (ip)
9625  IF r$ (TO 2) = "}}" THEN LET eof = 1
9630  LET ip = ip + 1
9640  RETURN
```

Comment: Note that *eof* is set to zero by *reset*. So if an input file is to be reread without a second call to *initcfs*, and the test for end of file is of the form IF eof THEN . . . it is necessary to insert GO SUB reset at the beginning of the reread. This will also reset the buffer pointers and counters correctly.

reset

Action	Sets buffer pointers, file flags etc.
Parameters passed to reset	None
returned	None
Global variables affected	ip, op, inbc, outbc, eof, switch
Local variables	None
cfs routines called	None

Listing

```
9970  LET ip = 0: LET op = 0: LET inbc = 0:
      LET outbc = 0: LET eof = 0: LET switch = 0
9980  RETURN
```

setup

Action	Allows the user to define record and block formats
Parameters passed to setup	None
returned	None
Global variables affected	n$, w
Local variables	nf, p
cfs routines called	None

Listing

```
9400  INPUT "Enter no. of fields"; nf
9405  CLS
9410  FOR p = 2 TO nf + 1
9415  PRINT AT 10, 2; "Field □"; p−1
9420  INPUT "Name of field (1st char: c/n):"; n$ (p)
```

```
9422  IF n$ (p, 1) < > "c" AND n$ (p, 1) < > "n" THEN GO TO 9420
9425  INPUT "No. of bytes:"; w (p)
9430  NEXT p
9435  CLS
9440  INPUT "No. of records per block:"; w (1)
9445  RETURN
```

write

Action	Writes a record in r$ to file
Parameters passed to write	r$
returned	None
Global variables affected	op, o$
Local variables	None
cfs routines called	putblock

Listing

```
9700  LET op = op + 1
9710  LET o$ (op) = r$
9720  IF op = w (1) OR r$ = "cfsend" THEN GO SUB putblock
9730  RETURN
```

INITIALIZATIONS ON LINE 1

```
1  LET close = 9740: LET getblock = 9800: LET initcfs = 9500:
   LET inrec = 9000: LET mesp = 9140: LET mesr = 9300:
   LET outrec = 9200: LET putblock = 9900: LET read = 9600:
   LET setup = 9400: LET write = 9700: LET reset = 9970
```

Appendix B: Automatic Cassette Control

It has already been pointed out that, in principle, there is no difficulty about controlling cassette motors automatically. Here is a specific technique for doing so.

A suitable I/O port is the ZX Spectrum PPI port marketed by Kempston (Micro) Electronics. This has three 8-bit ports on it which can be programmed to act as input or output ports, or combinations of the two. So far as we are concerned here, only 2 bits out of the available 24 are needed! So we'll select the low bits (bit Ø) of ports B and C to control the PLAY and RECORD cassettes respectively. (This is because the connections to the sockets for these ports are the same, whereas those for port A are slightly different.)

The port cannot be used to drive a relay directly, since only very small currents may be drawn (TTL levels). So the output is used to switch a transistor, as shown in Figure B1.

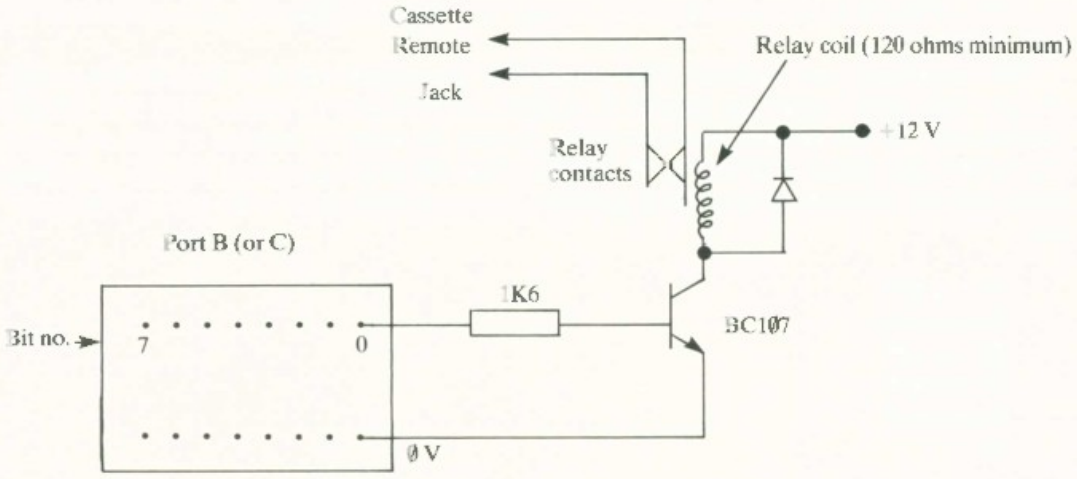


Figure B1

The ports are initialized as output mode by the statement:

```
OUT 127, 128
```

This can be inserted at the beginning of *initcfs*.

It is then necessary to turn on PORT B bit Ø to activate the PLAY cassette, and to turn on PORT C bit Ø to activate the RECORD cassette. To do this, we just replace the subroutines *mesp* and *mesr* as follows:

```
mesp:  9140  LET switch = NOT switch
       9150  OUT 63, switch
       9160  RETURN

mesr:  9300  LET switch = NOT switch
       9310  OUT 95, switch
       9320  RETURN
```

63 and 95 are the addresses allocated to ports B and C, respectively.

The circuit diagram (Figure B1) is based on one devised by Kempston (Micro) Electronics who may be contacted at 60, Adamson Court, Hillgrounds Road, Kempston, Bedford MK42 8QZ.

Appendix C: A User Guide to SDM — The Spectrum Data Manager

SDM is a simple data management system which allows the use of one input and one output file. Either of these files may be specified as non-existent by giving it the reserved name "null". Records are of fixed length format, and may consist of up to ten fields, each having a user-defined name and length. A field name may consist of up to ten characters, the first of which must be either "c" or "n" (lower case only) to indicate whether the field is of type character or numeric. This distinction is important, because it will determine how searching of, and additions to, the files are done. For instance, an attempt to search for records whose third field contains 357 will not find a record whose third field is 357.Ø or +357, if the field has been declared of type "c". Also, it is illegal to attempt to sum "c" type fields.

File names are user-defined. They must obey the normal rules for BASIC file names, except that they must not exceed 8 characters in length (2 fewer than the BASIC restriction). If the file is to be very long, it may be safer to impose a limit of, say, 6 characters per name. The reason is that the final filename passed to BASIC is formed from the user filename plus a block number. In fact the blocks of a file called "filename" are:

filenameh1
filenameh2
filenameØ
filename1
filename2
etc.

The first two blocks form a header label which describes to the system the record and field structure of the file. The data blocks follow this, and are labelled sequentially from zero upwards. Consequently, a file having more than 1ØØ blocks and an 8-character name will transgress the rules of BASIC and the program will break with error F. In practice, however, a file of this size is unlikely.

Operations allowed by SDM are:

1. The creation of files.
2. The modification of existing files by adding or deleting records, or by the creation of subfiles.
3. The searching of files for records having specified attributes.
4. The totalling of all specified numeric fields within a file.

Access to these operations is provided via a menu which is displayed after each operation is completed. Note, however, that this repetition of the menu is to facilitate several operations on the *same* file. If several operations are to be performed, but on different files, the program must be rerun.

ACTION OF SDM

After SDM has been loaded and RUN entered, the message "Spectrum Data Manager" is displayed on the screen, followed by a prompt for the input filename. If there is an input file, this should be loaded into the PLAY recorder and the name of the file entered. The system will request that the recorder is activated, and will enter the two header blocks. It will then ask for the recorder to be turned off.

If there is no input file, the word "null" should be entered. The system will then request a file definition.

The prompts are (in order):

	Prompt	Meaning
	1. Enter no. of fields	Number of fields in 1 record
These steps are repeated for each field. The field number is displayed as an additional prompt	2. Name of field (1st char: c/n)	Enter the name of a field as c or n followed by at most 9 characters, e.g. a field containing a name might be "cname", one containing a bank balance might be "nbal"
	3. No. of bytes	This is the maximum number of bytes which is anticipated to be occupied by a value stored in the field whose name has just been given
	4. No. of records per block	For simplicity of operation, this should be chosen fairly large, typically 20 or more. However, memory restrictions and the size of each record will create a practical barrier to large block sizes, particularly for a 16K machine. Maximum block sizes can fairly easily be calculated in a given case, but it is usually easier to choose a value by trial and error, simply ensuring that code 4 "Out of memory" does not occur

The system will then prompt for an output filename. If no new file is to be generated, this should be entered as "null". Otherwise a filename should be entered under the same rules as before. The system will then request the RECORD recorder to be turned on, and will save the header blocks, after which it will prompt for the RECORD recorder to be switched off.

The menu will now be displayed as shown below:

```
Options are:
  1) create
  2) add
  3) delete
  4) search
  5) sum
  6) exit
```

and the user is prompted to enter one of these (numeric) options.

Each option is now described individually.

1) create

The user is prompted to enter each field of a record in turn. The allowed size of the field is indicated by the appropriate number of underline characters appearing against the field name. (The field type is separated from the rest of the name by a colon as in n:bal). As each field is entered, the record is built up on the screen so that the user can check his entry. When a record is completed the prompt "Any more (y/n)?" will appear. If there are further records to be entered the user types "y", otherwise "n". In the former case the procedure will be repeated; in the latter case the user will be prompted to turn on the RECORD recorder and the file will be closed. (Note that the creation of a file will be punctuated by prompts to control the recorder as blocks are filled.)

2) add

The user is prompted for the number of records he wishes to add to the file. These are then entered in the same way as for the *create* routine (except that the process is repeated the specified number of times, rather than a prompt for another addition being issued at the end of each record entry). They may be entered in any order.

The user is next prompted for the key field name. The name of the field which defines the order of the records on the file should be specified. This name must *not* include the field type (e.g. write "bal" *not* "nbal"). The new records are now inserted automatically, with the system prompting the user from time to time to perform cassette control.

3) delete

The user is prompted first for the number of record types to be deleted, and then for the name of the key field (excluding the type code).

Finally, he is asked to enter each key whose associated record is to be deleted.

For example, suppose it is desired to delete the records of A. BROWN, G.N. DODDS and P. ADAMS. We enter:

```
How many records? 3
Enter key field name: name

Input deletion (key only): A. BROWN
Input deletion (key only): G.N. DODDS
Input deletion (key only): P. ADAMS
```

The system will then respond in a similar way to that to the *add* routine.

4) search

The user is first prompted for the name of the key field (i.e. that on which the search is to be made). If this is a c-type key, the next prompt is for a target key. For example, if a field called *ctaxded* exists whose contents will be "y" if an item is tax deductible, and it is desired to list out all tax deductible transactions, the answers to the prompts will be:

```
Enter key field name: taxded
Target key: y
```

If the key field is an n-type the second prompt requires a range. If only a specific target is required, both ends of the range should be entered as the same value. For instance, if there is a field called *namount*, and we wish to examine all records whose amount field is exactly 100, then the prompts will appear as:

```
Enter key field name: amount
Key range—low: 100
             high: 100
```

If, however, the search is for all values greater than or equal to 100 we could enter:

```
Key range—low: 100
             high: 1000
```

provided that we know that no values greater than 999 can exist. Such a value can always be chosen because we know the field width, so the above specification is guaranteed to be correct if *namount* has a length of 3 bytes. Similarly, if all entries less than some specified value are required, the "low" value must be set lower than any possible value (e.g. −100 for a 3-byte field).

The user is finally prompted for the device to which output is to be steered. This may be the display (device 1), a printer (device 2) or the output file (device 3).

The latter option allows the production of subfiles (e.g. a new file containing only tax deductible items can be created).

Note that, although there is no "list" option to list the entire file to a printer, *search* can mimic this function by using any numeric key with the "low" and "high" values both outside the allowable range.

5) sum

The user is prompted for a field name whose contents are to be summed throughout the file. The field must be numeric, otherwise an error message is printed and the system returns to the menu. The final total is displayed on the monitor, and the system waits until a key is hit before returning to the menu.

GENERAL COMMENTS

All the standard operations are performed on the whole file. So a question such as "What is the total of tax-deductible items?" cannot be answered directly. However, the problem can be solved by creating a file of tax-deductible items using search, and then summing the amount fields in the new file.

Similarly, complex searches (example: how many tax-deductible items are there whose values are between £50 and £100?) can be performed by doing successive searches and generating a new subfile each time.

Appendix D: Spectrum Data Manager– Program Listing

The final revised listing of SDM is given below. For simplicity, it includes the *cfs* routines; so if you already have these SAVEd, you should *not* copy lines 1 and 9000–9980 . LOAD them in first, or MERGE afterwards.

```
0  © Ian Stewart and Robin Jones 1982
1  LET close = 9740: LET getblock = 9800: LET initcfs = 9500:
   LET inrec = 9000: LET mesp = 9140: LET mesr = 9300:
   LET outrec = 9200: LET putblock = 9900: LET read = 9600:
   LET reset = 9970: LET setup = 9400: LET write = 9700
2  LET loutrec = 8800: LET extractkey = 7800: LET loosends = 7600:
   LET compc = 7400: LET compn = 7200: LET sort = 7000:
   LET ckeytest = 6800: LET nkeytest = 6600
10 CLS
20 PRINT AT 0, 5; "Spectrum Data Manager"
25 GO SUB initcfs
26 CLS
30 PRINT AT 2, 0; "Options are:"
40 PRINT AT 3, 11; "1) create"
41 PRINT AT 4, 11; "2) add"
42 PRINT AT 5, 11; "3) delete"
43 PRINT AT 6, 11; "4) search"
44 PRINT AT 7, 11; "5) sum"
45 PRINT AT 8, 11; "6)exit"

100 INPUT "Enter option:"; opt
110 GO SUB 200 * opt
120 GO SUB reset
130 GO TO 26

200 GO SUB inrec
210 GO SUB write
220 INPUT "any more? (y/n)"; q$
230 IF q$ = "y" THEN GO TO 200
240 GO SUB close
250 RETURN

400 INPUT "How many records?"; nr
410 DIM b$ (nr, bpr)
420 FOR q = 1 TO nr
430 GO SUB inrec
440 LET b$ (q) = r$
```

```
450  NEXT q
460  GO SUB extractkey
465  GO SUB sort
470  LET ap = 1
480  GO SUB read
490  IF ap > nr OR eof THEN GO SUB loosends: RETURN
500  LET s$ = r$ (begin TO end): LET t$ = b$ (p) (begin TO end)
510  IF ctype THEN GO SUB compc
520  IF NOT ctype THEN GO SUB compn
530  IF comp < 0 THEN GO SUB write: GO TO 480
540  IF comp >= 0 THEN LET t$ = r$: LET r$ = b$ (p): GO SUB write:
     LET r$ = t$: LET ap = ap + 1: GO TO 490

600  INPUT "How many records?"; nr
610  GO SUB extractkey
620  DIM d$ (nr, end - begin + 1)
630  FOR p = 1 TO nr
640  INPUT "Enter deletion (key only):"; d$ (p)
650  NEXT p
660  GO SUB read
670  IF eof THEN GO SUB close: RETURN
680  FOR p = 1 TO nr
690  IF r$ (begin TO end) = d$ (p) THEN GO TO 660
700  NEXT p
710  GO SUB write
720  GO TO 660

800  GO SUB extractkey
810  IF ctype THEN DIM k$ (end - begin + 1): INPUT "Target key:"; k$
820  IF NOT ctype THEN INPUT "key range—low:"; low, "high:"; high
830  INPUT "Display (1), print (2) or file (3):"; opt
840  GO SUB read
850  IF eof AND opt = 3 THEN GO SUB close: RETURN
860  IF eof THEN INPUT "Enter c to continue"; k$: RETURN
870  IF ctype THEN GO SUB ckeytest
880  IF NOT ctype THEN GO SUB nkeytest
890  IF NOT match THEN GO TO 840
895  LET bs = begin: LET es = end
900  IF opt = 1 THEN GO SUB outrec
910  IF opt = 2 THEN GO SUB loutrec
920  IF opt = 3 THEN GO SUB write
```

```
925  LET begin = bs: LET end = es
930  GO TO 840

1000  LET sum = 0
1010  GO SUB extractkey
1020  IF ctype THEN PRINT "Key not numeric": PAUSE 120: RETURN
1030  GO SUB read
1040  IF eof THEN PRINT "Sum of □"; k$; "= □"; sum:
      INPUT "Enter c to continue"; k$: RETURN
1050  LET sum = sum + VAL r$ (begin TO end)
1060  GO TO 1030

1200  STOP

6600  LET match = 1
6610  IF VAL r$ (begin TO end) < low OR VAL r$ (begin TO end) > high
      THEN LET match = 0
6620  RETURN

6800  LET match = 0
6810  IF k$ = r$ (begin TO end) THEN LET match = 1
6820  RETURN

7000  LET inc = 1
7005  LET flag = 0
7010  FOR p = 1 TO nr − inc
7020  LET s$ = b$ (p) (begin TO end): LET t$ = b$ (p + 1) (begin TO end)
7030  IF ctype THEN GO SUB compc
7040  IF NOT ctype THEN GO SUB compn
7050  IF comp > 0 THEN LET t$ = b$ (p): LET b$ (p) = b$ (p + 1):
      LET b$ (p + 1) = t$: LET flag = 1
7060  NEXT p
7070  IF flag > 0 THEN LET inc = inc + 1: GO TO 7005
7080  RETURN

7200  IF VAL s$ < VAL t$ THEN LET comp = − 1
7210  IF VAL s$ = VAL t$ THEN LET comp = 0
7220  IF VAL s$ > VAL t$ THEN LET comp = 1
7230  RETURN

7400  IF s$ < t$ THEN LET comp = −1
7410  IF s$ = t$ THEN LET comp = 0
```

```
7420  IF s$ > t$ THEN LET comp = 1
7430  RETURN

7600  IF ap > nr THEN GO TO 7670
7610  FOR p = ap TO nr
7620  LET r$ = b$ (p)
7630  GO SUB write
7640  NEXT p
7650  GO SUB close
7660  RETURN
7670  GO SUB write
7680  GO SUB read
7690  IF eof THEN GO SUB close: RETURN
7700  GO TO 7670

7800  DIM k$ (9): INPUT "Enter key field name"; k$
7810  LET begin = 1
7820  FOR p = 2 TO 11
7830  IF n$ (p) (2 TO  ) = k$ THEN GO TO 7860
7840  LET begin = begin + w (p)
7850  NEXT p
7860  LET end = begin + w (p) - 1
7870  IF n$ (p, 1) = "c" THEN LET ctype = 1
7880  IF n$ (p, 1) = "n" THEN LET ctype = 0
7890  RETURN

8800  LET begin = 1
8810  FOR p = 2 TO 11
8815  IF n$ (p, 1) = "□" THEN LPRINT: RETURN
8820  LPRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO  );
8840  LET end = begin + w (p) - 1
8850  LPRINT TAB 15; r$ (begin TO end)
8860  LET begin = end + 1
8870  NEXT p
8880  LPRINT: RETURN

9000  DIM a$ (bpr)
9010  CLS: LET begin = 1
9020  FOR p = 2 TO 11
9025  IF n$ (p, 1) = "□" THEN GO TO 9120
9030  PRINT AT p, 0; n$ (p, 1); ":"; AT p, 4; n$ (p) (2 TO  )
```

```
9040  FOR c = 1 TO w (p)
9050  PRINT AT p, 14 + c; "_"
9060  NEXT c
9070  LET end = begin + w (p) - 1
9080  INPUT (n$ (p) (2 TO    ) ); a$ (begin TO end)
9090  PRINT AT p, 15; a$ (begin TO end)
9100  LET begin = end + 1
9110  NEXT p
9120  LET r$ = a$
9130  RETURN
9140  PRINT INVERSE 1; "Start" AND NOT switch; "Stop" AND switch;
      "PLAY recorder"
9150  BEEP .2, 15: PAUSE 6: BEEP .3, 20: PAUSE 80
9160  LET switch = NOT switch
9170  RETURN

9200  LET begin = 1
9210  FOR p = 2 TO 11
9215  IF n$ (p, 1) = "□" THEN PRINT: RETURN
9220  PRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO    );
9240  LET end = begin + w (p) - 1
9250  PRINT TAB 15, r$ (begin TO end)
9260  LET begin = end + 1
9270  NEXT p
9280  PRINT: RETURN

9300  PRINT INVERSE 1; "Start" AND NOT switch;
      "Stop" AND switch; "recording"
9310  BEEP .2, 20: PAUSE 6: BEEP .3, 15: PAUSE 80
9320  LET switch = NOT switch
9330  RETURN

9400  INPUT "Enter no. of fields:"; nf
9405  CLS
9410  FOR p = 2 TO nf + 1
9415  PRINT AT 10, 2; "Field □"; p - 1
9420  INPUT "Name of field (1st char: c/n):"; n$ (p)
9422  IF n$ (p, 1) < > "c" AND n$ (p, 1) < > "n" THEN GO TO 9420
9425  INPUT "No of bytes:"; w (p)
9430  NEXT p
```

```
9435  CLS
9440  INPUT "No of records per block:"; w (1)
9445  RETURN

9500  DIM n$ (11, 10): DIM w (11): LET bpr = 0: GO SUB reset
9505  INPUT "Enter input filename:"; f$
9510  IF f$ = "null" THEN GO SUB setup: GO TO 9525
9514  GO SUB mesp
9515  LOAD f$ + "h1" DATA n$ ( )
9520  LOAD f$ + "h2" DATA w ( )
9521  GO SUB mesp
9525  INPUT "Enter output filename:"; g$
9530  FOR p = 2 TO 11
9535  LET bpr = bpr + w (p)
9540  NEXT p
9545  DIM i$ (w (1), bpr): DIM o$ (w (1), bpr)
9547  IF g$ = "null" THEN RETURN
9548  GO SUB mesr
9550  SAVE g$ + "h1" DATA n$ ( )
9555  SAVE g$ + "h2" DATA w ( )
9557  GO SUB mesr
9560  RETURN

9600  IF ip = 0 OR ip > w (1) THEN GO SUB getblock
9610  IF i$ (ip) (    TO 6) = "cfsend" THEN PRINT "Attempt to read past
      end of file": STOP
9620  LET r$ = i$ (ip)
9625  IF r$ (    TO 2) = "}}" THEN LET eof = 1
9630  LET ip = ip + 1
9640  RETURN

9700  LET op = op + 1
9710  LET o$ (op) = r$
9720  IF op = w (1) OR r$ = "cfsend" THEN GO SUB putblock
9730  RETURN
9740  LET r$ = "}}":
9750  GO SUB write
9760  LET r$ = "cfsend"
9770  GO SUB write
9780  RETURN
```

```
9800  LET m$ = STR$ inbc
9810  GO SUB mesp
9820  LOAD f$ + m$ DATA i$ ( )
9825  POKE 23692, 255
9830  GO SUB mesp
9840  LET ip = 1
9850  LET inbc = inbc + 1
9860  RETURN

9900  LET m$ = STR$ outbc
9910  GO SUB mesr
9920  SAVE g$ + m$ DATA o$ ( )
9930  GO SUB mesr
9940  LET op = 0
9950  LET outbc = outbc + 1
9960  RETURN
9970  LET ip = 0: LET op = 0: LET inbc = 0: LET outbc = 0:
      LET eof = 0: LET switch = 0
9980  RETURN
```

Appendix E: Make your own Load/Save Switch

How many times have you forgotten to remove an "ear" jack when saving a program? On the fifteenth occasion I did this, I decided it was high time I took steps to prevent it. Here's a simple but practical solution.

You need:

(a) 4 × 3.5 mm jack plugs.
(b) 1 miniature 2-pole, 2-way slide switch.
(c) 2 metres of screened microphone cable (single core).
(d) An old cassette case.

(a), (b) and (c) are available in any electronics shop for something between 25p and 50p each. (d) can be any convenient small plastic box.

Drill four holes in the narrow sides of the box large enough to allow the cable to pass through. Make a hole in one of the large faces to accommodate the slide switch. (The easiest way to make sure this hole is just the right size is to cut a piece of PVC adhesive tape to act as a template. Now make the connections shown in Figure E1.

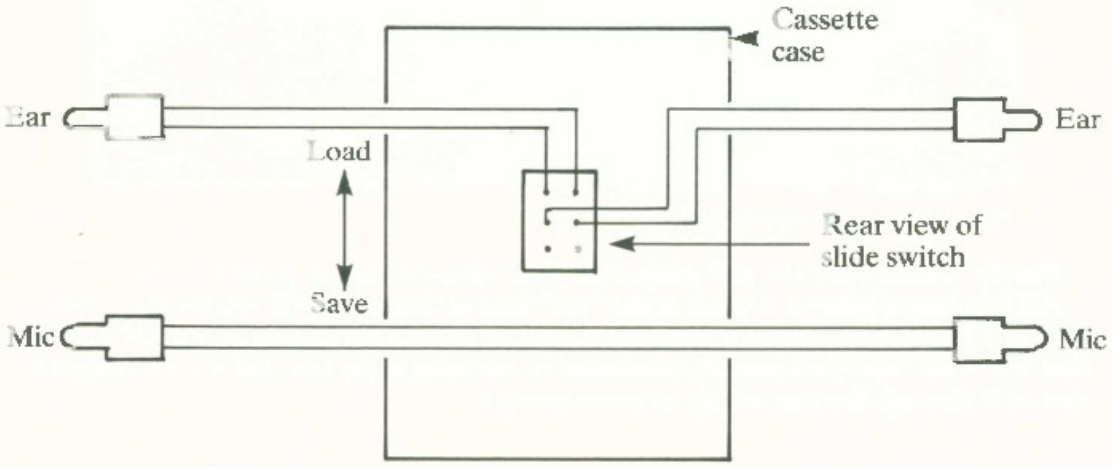


Figure E.1 Circuit diagram for LOAD/SAVE switch.

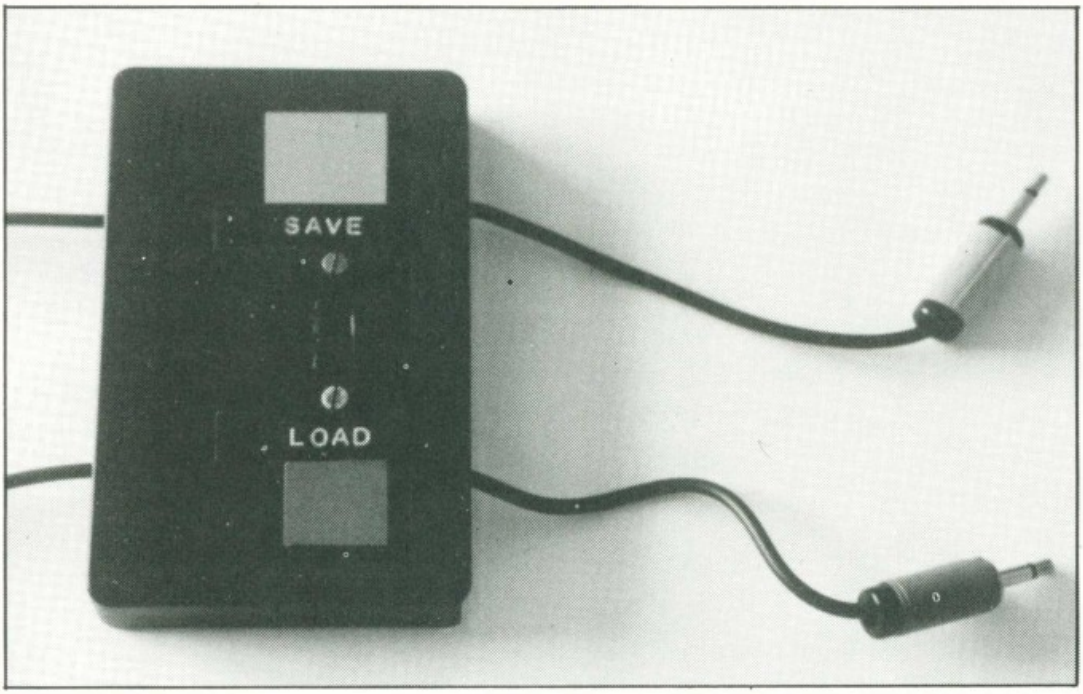


Figure E.2 Front view of switch.

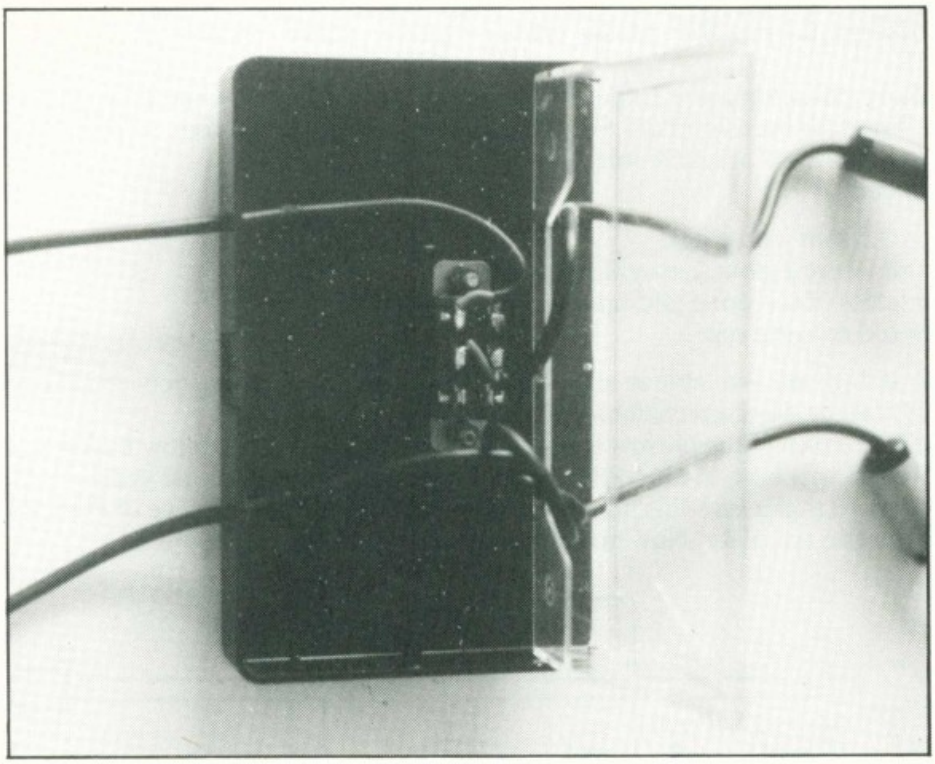


Figure E.3 Back view of switch.

Finally, colour code the jack plugs with PVC adhesive tape to avoid confusion.

All this does is provide a break in the "ear" lead so that on SAVE you use the switch in the SAVE position (logical!), and slide it back to LOAD for LOAD or VERIFY. Note that the "mic" lead isn't actually connected to anything in the box, but it's convenient to thread it through the case simply to avoid losing it.

Other titles of interest

Easy Programming for the ZX Spectrum
Ian Stewart & Robin Jones

Computer Puzzles: For Spectrum and ZX81
Ian Stewart & Robin Jones

Games to Play on Your ZX Spectrum
Martin Wren-Hilton

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*

Machine Code and better Basic
Ian Stewart & Robin Jones

The ZX81 Add-On Book
Martin Wren-Hilton

Shiva Software

Spectrum Special 1
Ian Stewart & Robin Jones

A selection of 10 educational games and puzzles.

Available from March '83

Spectrum Machine Code
Ian Stewart & Robin Jones

Spectrum in Education
Eric Deeson

Brainteasers for BASIC Computers
Gordon Lee

Shiva Software

Spectrum Specials 2 & 3
Ian Stewart & Robin Jones

This sequel to *Easy Programming* helps you get more out of your ZX Spectrum computer.

Enhance it:

- Cassette files
- Data management
- Crashproof graphics
- Flexible line-renumbering

Exploit it:

- User-defined functions
- System variables
- Attribute and display files
- Brand new character sets

Employ it:

- Star charts
- Code-breaking
- Psychology
- Statistics

An original selection of programs and applications needing only 16K of memory, and so can be RUN on either model of the Spectrum.

WHY NOT BROADEN *YOUR* SPECTRUM?

Shiva Publishing Limited ISBN 0 906812 24 0 UK price £5.95 net