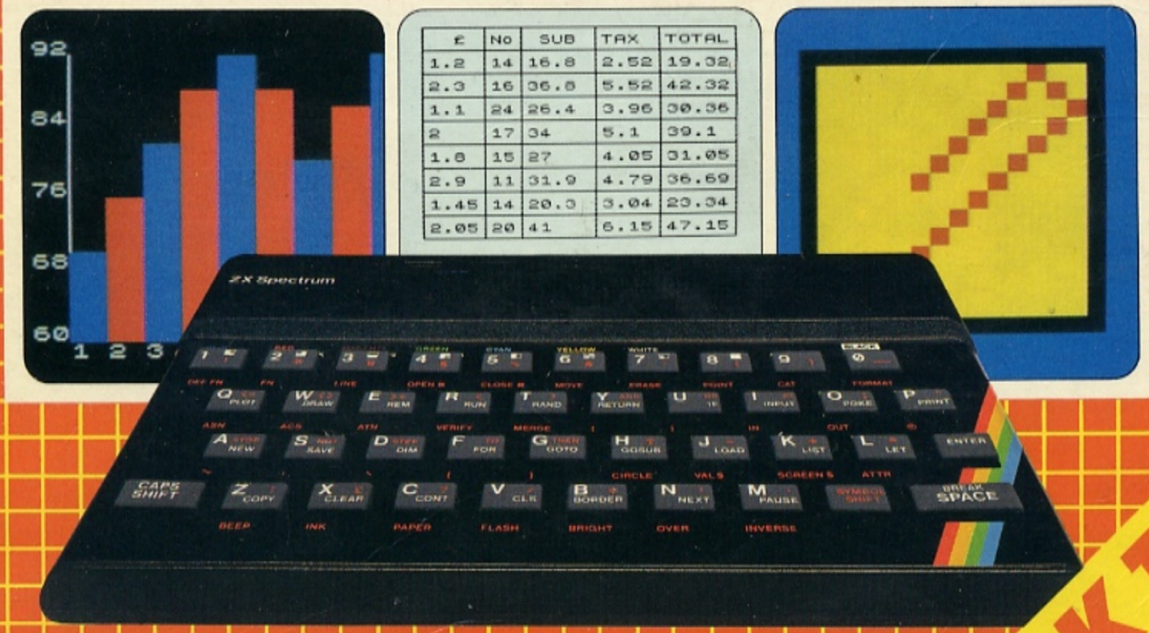
DK

ScreenShot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM



IAN GRAHAM

BOOK TWO

Further techniques for exciting games and graphics – in full colour

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM

THE DK SCREEN-SHOT PROGRAMMING SERIES

Never has there been a more urgent need for a series of well-produced, straightforward, practical guides to learning to use a computer. It is in response to this demand that The DK Screen-Shot Programming Series has been created. It is a completely new concept in the field of teach-yourself computing. And it is the first comprehensive library of highly illustrated, machine-specific, step-by-step programming manuals.

BOOKS ABOUT THE ZX SPECTRUM

This is Book Two in a series of unique step-by-step guides to programming the ZX Spectrum. Together with its companion volumes, it will build up into a self-contained teaching course that begins with the basic principles of programming, and progresses – via more sophisticated techniques and routines – to an advanced level.

BOOKS ABOUT OTHER COMPUTERS

Additional titles in the series will cover each of the world's most popular computers. These will include:

Step-by-Step Programming for the **BBC Micro**

Step-by-Step Programming for the **Commodore 64**

Step-by-Step Programming for the **Acorn Electron**

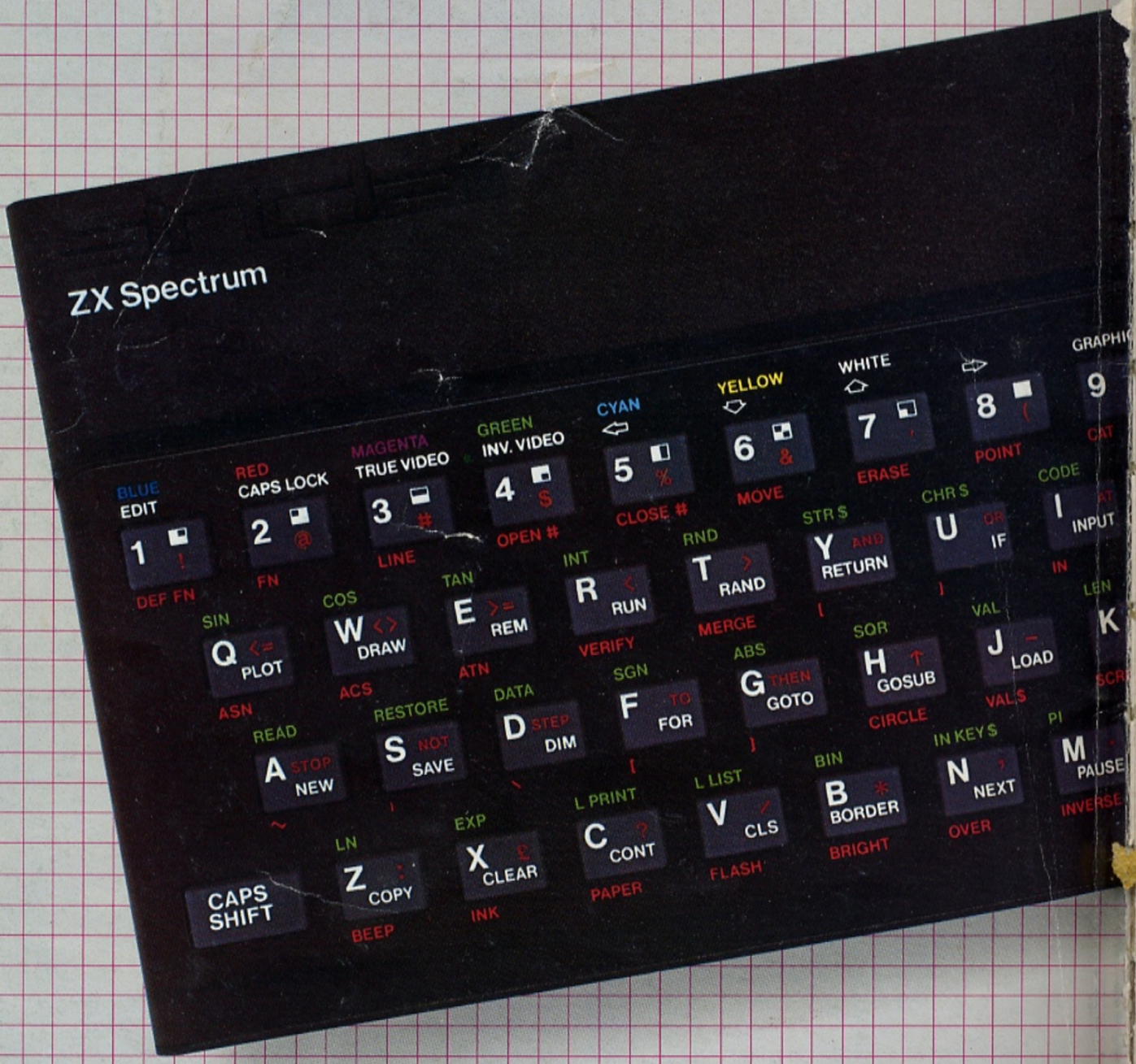
Step-by-Step Programming for the **Apple II**

Step-by-Step Programming for the **IBM PCjr**

IAN GRAHAM

After taking a B.Sc. in Applied Physics and a postgraduate diploma in Journalism at The City University, London, Ian Graham worked as assistant editor of *Electronics Today International* and deputy editor of *Which Video?* Since becoming a full-time freelance writer in 1982, he has contributed to a wide range of technical magazines (including *Computing Today, Video Today, Video Search, Hobby Electronics, Electronic Insight, Popular Hi-Fi, Science Now,* and *Next...)* and has also written a number of popular books on computers and computing. These include *Computer & Video Games, Information Technology, The Inside Story – Computers,* and *The Personal Computer Handbook* (co-written with Helen Varley).

BOOK TWO


ZX Spectrum
BLUE
EDIT
1
DEF FN
!
RED
CAPS LOCK
2
FN
@
MAGENTA
TRUE VIDEO
3
LINE
#
GREEN
INV. VIDEO
4
OPEN #
$
CYAN
5
CLOSE #
%
YELLOW
6
MOVE
&
WHITE
7
ERASE
'
8
POINT
(
9
CAT
)
0
SIN
Q
ASN
PLOT
<=
COS
W
ACS
DRAW
<>
TAN
E
ATN
REM
>=
INT
R
RUN
<
RND
T
RAND
>
STR $
Y
AND
RETURN
CHR $
U
OR
IF
CODE
I
INPUT
AT
IN
LEN
K
READ
A
STOP
NEW
RESTORE
S
NOT
SAVE
DATA
D
STEP
DIM
SGN
F
TO
FOR
ABS
G
THEN
GOTO
SQR
H
GOSUB
VAL
J
LOAD
VAL $
CIRCLE
PI
IN KEY $
N
NEXT
M
PAUSE
INVERSE
OVER
BRIGHT
FLASH
PAPER
INK
BEEP
CAPS SHIFT
LN
Z
COPY
EXP
X
CLEAR
£
L PRINT
C
CONT
?
L LIST
V
CLS
/
BIN
B
BORDER
*

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING ZX SPECTRUM

IAN GRAHAM

DORLING KINDERSLEY · LONDON

BOOK TWO

CONTENTS

The DK Screen-Shot Programming Series was conceived, edited and designed by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

Project Editor David Burnie
Art Editor Peter Luff
Designer Hugh Schermuly
Photography Vincent Oliver
Managing Editor Alan Buckingham
Art Director Stuart Jackman

First published in Great Britain in 1984 by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.
Second impression 1984

British Library Cataloguing in Publication Data

Graham, Ian, 1953–
Step-by-step programming for the ZX Spectrum. Book 2.
1. Sinclair ZX Spectrum (Computer)—— Programming
I. Title
001.64'2 QA76.8.S625

ISBN 0-86318-031-0

Typesetting by The Letter Box Company (Woking) Limited, Woking, Surrey, England
Reproduction by Reprocolor Llovet S.A., Barcelona, Spain
Printed and bound in Italy by A. Mondadori, Verona

DEFINING FUNCTIONS

All computers feature a range of built-in functions, commands which can be used to transform one number into another in a specific way. Functions produce a result that can be used later in a program. SQR (SQuare Root) or INT (INTeger) are two examples of functions that are pre-programmed on the Spectrum. When you use these commands, they take a number and operate on it to produce another number.

The range of keyboard functions on the Spectrum is quite wide. But if you want to use a function that does not appear on a key, you don't have to type out a chain of instructions every time. The Spectrum allows you to program it to carry out specific sequences of calculations. These functions are "called" by the command FN (FunctioN) and defined by the command DEF FN (DEFine FunctioN).

How to write a function

In order to use a function, you must first define what it is going to do. That is done with a defining statement. For instance:

```
120 DEF FN a(x)=4*x+36
```

defines a function called "a". The number that a function operates on is known as its argument. In this case the argument of the function is x. The function takes whatever value of x it is given, multiplies it by 4, and then adds 36. If in a program you want to put the number 10 into the function, you would do so by using the keyword FN like this:

```
200 PRINT FN a(10)
```

This would PRINT the value of the function when 10 is substituted for x – which is 4*10+36, or 76.

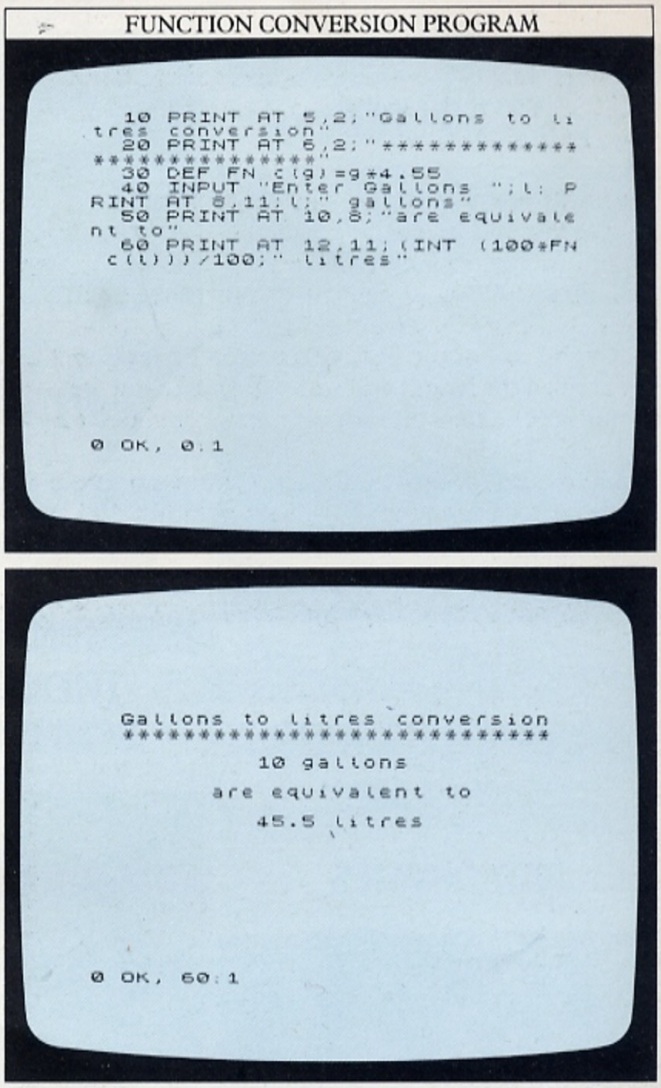
DEF FN is obtained by pressing CAPS SHIFT together with SYMBOL SHIFT, followed by SYMBOL SHIFT again to get the extended mode cursor, together with key 1. The same procedure, but using key 2 instead of key 1, produces FN.

Once a function has been defined in a program, you can use it and its argument just like any other number or numeric variable. For example, you can add, subtract, divide and multiply functions and their arguments together, and even make functions work on numbers that are themselves produced by functions. Unless you are doing mathematical research you are unlikely to get this far, but for more straightforward tasks functions are easy to use and helpful in making programs simpler.

Why use a function?

The following program shows a simple way in which you can put functions to work in a program to produce a numerical result which is then PRINTed:

FUNCTION CONVERSION PROGRAM



The program carries out a conversion. The function that actually does the converting is defined in line 30. Line 40 waits for you to type in a number of gallons, which is then converted into the equivalent number of litres by FN c(1) in line 60. Multiplying and dividing by 100 may look odd, but it's just a way of producing an answer to two places of decimals with INT. Using INT on its own removes all decimals, and so reduces accuracy.

Going to the trouble of using a function here might seem a bit unnecessary, and in fact it's unlikely that you would use FN in such a simple program. But imagine what would happen if you wanted to do the calculation a number of times at different places in the same program, and with different numbers. It is then that the user-defined function really comes into its own. When the function is long and complicated, defining it just once enables you make calculation lines much simpler to write and check. FN is very much like a one-

command subroutine that deals only with numbers.

Because an expression containing FN actually represents a number, you can use it to replace any kind of complex calculation. When you write your own functions, you are in effect giving the computer functions that its resident programming language (BASIC) doesn't already have – extending the capabilities of the language.

Calculation sequences with functions

Imagine that you want to calculate the cost of something that is sold by area – perhaps carpets to cover the floor of a house. You would need to multiply the length and width of each room to get its area, and then multiply that by the cost of the floor covering per square metre. If you called the length and width X and Y, and the cost per unit area C, then the cost per room would be worked out by (X*Y)*C.

In the next program, the cost for each room is calculated by a function. It is defined in line 190, and used in lines 180 and 200:

CARPET COSTER PROGRAM

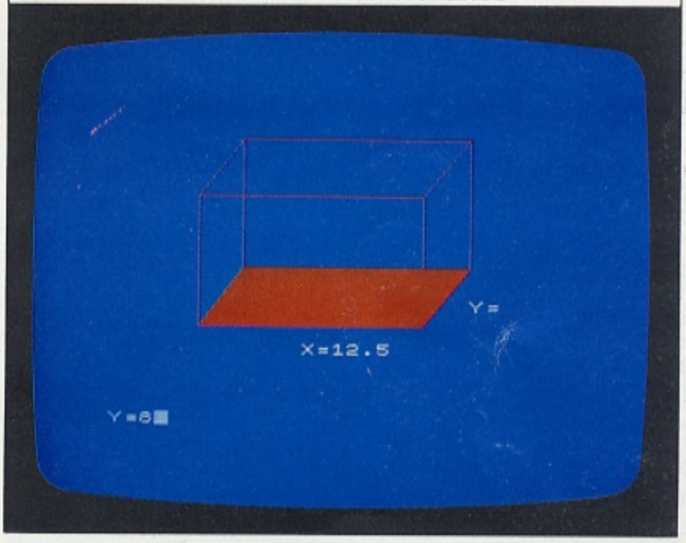
```
  10 LET A=0
  20 BORDER 1: PAPER 1: INK 2: C
LS
  30 PLOT 72,144
  40 DRAW 120,0: DRAW -24,-32: D
RAW -120,0: DRAW 24,32: DRAW 0,-
72: DRAW 120,0: DRAW -24,-32: DR
AW -120,0: DRAW 24,32
  50 PLOT 192,72: DRAW 0,72: PLO
T 168,40: DRAW 0,72: PLOT 48,40:
 DRAW 0,72
  60 FOR X=48 TO 168
  70 PLOT X,40
  80 DRAW 24,32
  90 NEXT X
 100 INK 7
 110 PRINT AT 18,13;"X=";AT 15,2
4;"Y="
 120 INPUT "X=";X: PRINT AT 18,1
5;X
 130 INPUT "Y=";Y: PRINT AT 15,2
6;Y

scroll?
```

```
 140 PAUSE 150: PAPER 0: BORDER
0: CLS
 150 PRINT AT 6,3;"LENGTH= ";X;A
T 8,3;"WIDTH= ";Y
 160 INPUT "COST PER SQUARE METR
E? ";C
 170 PRINT AT 4,3;"COST PER SQUA
RE METRE= ";C
 180 PRINT AT 12,3;"COST= ";FN T
(C)
 190 DEF FN T(C)=(X*Y)*C
 200 LET A=A+FN T(C)
 210 PRINT AT 14,3;"TOTAL COST=
";A
 220 PAUSE 250
 230 GO TO 20



0 OK, 0:1
```

Lines 20 to 130 set up a graphics display which produces an outline of the room, and then wait for you to ENTER values for length and width. Once you have done this, the program INPUTs the values and then clears the screen.

The next display asks you for the cost per square metre of the floor covering. Once you have keyed this in, the program tells you what it would cost to cover the room. As well as using the function T to produce this, it uses it in line 200 to update a running total. If you keep keying in values every time the program returns to the first display, you will find that the TOTAL COST line in the second display will be updated to show the running total of all the costs calculated:

CARPET COSTER DISPLAYS




```
COST PER SQUARE METRE= 6.5
LENGTH= 12.5
WIDTH= 8

COST= 650
TOTAL COST= 1750
```

You can define a function at any point in a program, although if the DEF FN and FN lines are far apart in a long program, this will increase the program's RUNning time. In the carpet coster program, using a function makes lines 180 and 200 less complicated than they otherwise would be. On page 33 you will see how using a function can make graphics far easier as well.

EXTENDING DECISIONS

The BASIC keywords IF and THEN let a program operate in one way until the condition specified by the IF statement is encountered. When this happens, the program is then triggered to follow another course of action. But the capabilities of IF ... THEN do not stop at making a straightforward "yes" or "no" decision. By combining IF ... THEN with the keywords AND and OR you can make it tackle much more complicated situations. Because BASIC is designed to reflect how words are used in ordinary language, you can use IF ... THEN just as you would when describing a set of conditions to someone. Here is a program which shows how you can take IF ... THEN decision-making to a more advanced level:

BOUNCING PROGRAM

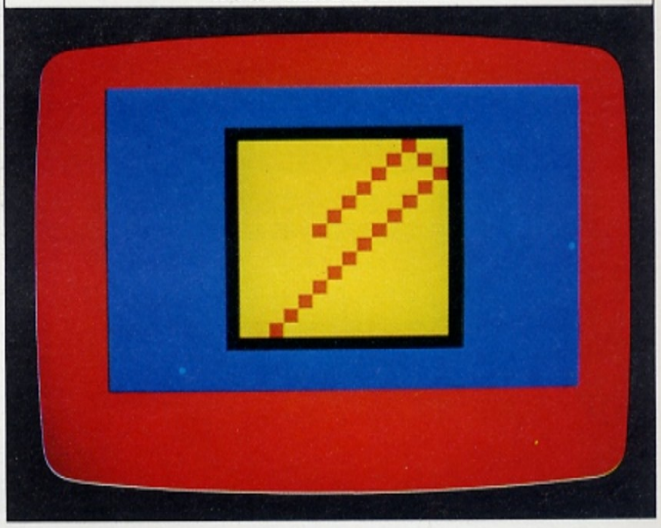
```
 10 BORDER 2: PAPER 1: INK 0: C
LS
 20 FOR r=3 TO 18
 30 PRINT AT r,8;"■"
 40 PRINT AT r,23;"■"
 50 PRINT AT 3,r+5;"■"
 60 PRINT AT 18,r+5;"■"
 70 NEXT r
 80 PAPER 6: INK 2
 90 FOR r=4 TO 17
100 FOR c=9 TO 22
110 PRINT AT r,c;" "
120 NEXT c
130 NEXT r
140 LET r=5+INT (RND*12)
150 LET c=10+INT (RND*12)
160 LET a=1: LET b=1
170 PRINT AT r,c;" "
180 LET r=r+a
190 LET c=c+b
200 IF r=4 OR r=17 THEN LET a=-
a

scroll?
```

```
210 IF c=9 OR c=22 THEN LET b=-
b
220 PRINT AT r,c;"■"
230 BEEP 0.1,3*r
240 GO TO 170









0 OK, 0:1
```

Lines 10 to 130 simply set up the display – a black outline box in the middle of the screen. Lines 140 and 150 produce a random starting position for a "ball" (a graphics square) inside the box. To make the ball appear to move, line 170 erases the image at r,c and lines 180 and 190 produce new co-ordinates where line 220 will rePRINT the ball.

Before this happens, lines 200 and 210 check whether the ball has reached any of the box's walls. They examine the ball's position to see if the row number is one below the box lid or one above the box bottom, or if its column number is one more than the left side or one less than the right side. If any of these conditions is satisfied, lines 200 or 210 reverse the ball's vertical or horizontal motion, whichever is necessary:

BOUNCING DISPLAY



Adding decisions together

You can now move on a stage further from the previous program to see how a second option can be incorporated in an IF ... THEN statement. The next program features two balls moving independently:

DOUBLE BOUNCING PROGRAM (1)

```
 10 BORDER 2: PAPER 1: INK 0: C
LS
 20 FOR r=3 TO 18
 30 PRINT AT r,8;"■";AT r,23;"■
"
 40 PRINT AT 3,r+5;"■";AT 18,r+
5;"■"
 50 NEXT r
 60 PAPER 6: INK 2
 70 FOR r=4 TO 17
 80 FOR c=9 TO 22
 90 PRINT AT r,c;" "
100 NEXT c
110 NEXT r
120 LET r=5+INT (RND*12)
130 LET s=5+INT (RND*12)
140 LET c=10+INT (RND*12)
150 LET d=10+INT (RND*12)
160 LET a=1: LET b=1
170 LET k=-1.3: LET l=1
180 PRINT AT r,c;" "
190 PRINT AT s,d;" "

scroll?
```

This program is very similar to the first one, except that now there are two lines that PRINT a graphics square at changing row and column numbers. As before, each of the squares starts at a random co-ordinate which is defined in lines 120 to 150. The squares are then animated by lines 160 to 300:

DOUBLE BOUNCING PROGRAM (2)

```
 200 LET r=r+a: LET c=c+b
 210 LET s=s+k: LET d=d+l
 220 IF r=4 OR r=17 THEN LET a=-
a
 230 IF s<5 OR s>16 THEN LET k=-
k
 240 IF c=9 OR c=22 THEN LET b=-
b
 250 IF d=9 OR d=22 THEN LET l=-
l
 260 PRINT AT r,c;"■"
 270 PRINT AT s,d;"■"
 280 IF r>=INT (s+0.5)-1 AND r<=
INT (s+0.5)+1 AND c>=d-1 AND c<=
d+1 THEN BEEP 0.5,-10: GO TO 10
 290 BEEP 0.02,3*r: BEEP 0.02,3*
s
 300 GO TO 160



0 OK, 0:1
```

The second ball is made to set off in a different direction from the first and at a slightly faster vertical speed, so that the two balls have a greater chance of meeting. Otherwise, they would just follow each other around the box in the same tracks. Line 280 is the one in which the computer makes a multiple decision about the position of both of the squares. Without this line, when the squares met, they would just carry on through each other as if nothing had happened. This isn't a very convincing simulation of what would really occur, so line 280 decides whether the squares are close enough together to have collided. The line includes an IF ... THEN decision with three ANDs to see if r and s are sufficiently close together, and then if c and d are also within the same limits. It does this by taking r, for example, and then deciding whether it is greater or equal to INT(s+0.5)−1 and simultaneously smaller or equal to INT(s+0.5)+1.

If all these conditions are met, then it means that the two balls are either occupying the same square or are on adjacent squares, in which case they can be assumed to have collided. A BEEP is sounded and the whole process starts again.

IF ... THEN in games programming

Decision-making by this method is very useful if you want to know whether or not two characters are occupying the same screen location. This is often used in program where a character is "shot down" by another. The next program shows how you can incorporate the technique in a simple game which has a number of characters moving simultaneously:

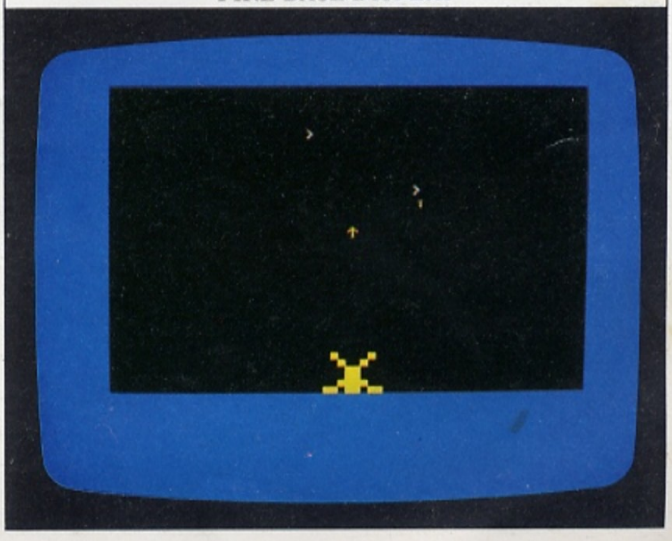
FIRE-BASE PROGRAM

```
 10 BORDER 1: PAPER 0: INK 6: C
LS
 20 PRINT AT 19,15;"▗▄▖";AT 20,
16;"▀";AT 21,14;"▄▀▄▀▄"
 30 LET c=0: LET d=7: LET r=18
 40 PRINT AT 3,c;" ";AT 7,d;" "
 50 LET c=c+1: LET d=d+1
 60 IF c>31 THEN LET c=0
 70 IF d>31 THEN LET d=0
 80 PRINT INK 7;AT 3,c;">";AT 7
,d;">"
 90 PRINT AT r,16;" "
100 IF r=0 THEN LET r=19
110 LET r=r-1
120 PRINT AT r,16;"↑": BEEP 0.0
2,800/(3*r+15)
130 IF r=3 AND c=16 OR r=7 AND
d=16 THEN BEEP 0.5,r+2: GO TO 10
140 GO TO 40



0 OK, 0:1
```

The program PRINTs a fire-base at the bottom of the screen. It fires upward arrows at two horizontal arrows that repeatedly fly across the screen. Line 130 checks whether the screen co-ordinates of the upward arrow are the same as those of either of the horizontal arrows. If they are, the program jumps right back to line 10 and begins again. If not, it jumps back to line 40 and moves all the characters on one space. Line 100 checks whether the upward arrow has reached the top of the screen. If it has, a new arrow is launched from just above the fire-base. Lines 60 and 70 check whether either of the horizontal arrows have reached the right edge of the screen. If either has, its column co-ordinate is reset to zero.

When you RUN the program, you should find that the fire-base's arrow finds its target on the horizontal arrows' seventh pass across the screen. This happens because the program is working with fixed figures. If you use RND instead, the results become unpredictable and will change with each RUN:

FIRE-BASE DISPLAY



CHANGING STEP IN GRAPHICS

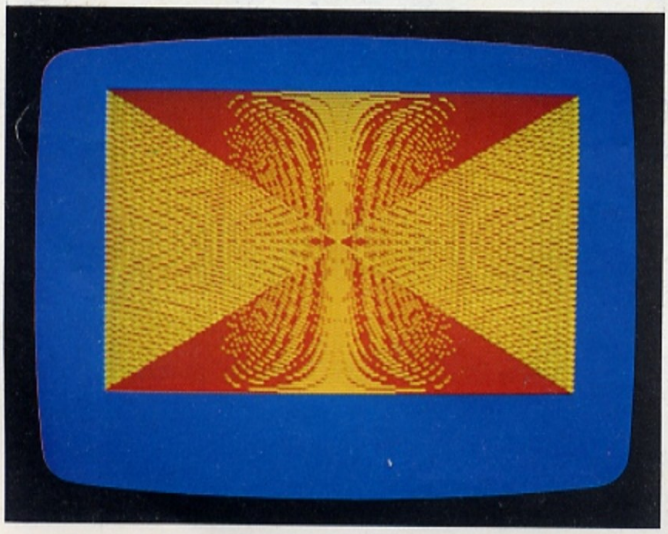
FOR ... NEXT loops are very useful for repeating a sequence of program statements a predictable number of times. If you want to increase the value of a variable by more than 1 on every cycle through a loop, you could replace FOR ... NEXT by a GOTO statement preceded by N=N+2 or whatever change you want to make to N. However, if you have already started writing a program and then find that the standard FOR ... NEXT loop is unsuitable, it is sometimes awkward and time-consuming to substitute a completely new programming technique like this. Fortunately, there is a much more straightforward way of jumping forwards or even backwards in a loop.

How to change jump sizes with STEP

The BASIC keyword that deals with this is STEP, which you may already have come across in Book 1. Here is a program which uses STEP with graphics:

GRAPHICS WITH **STEP**

```
LS
  10 BORDER 1: PAPER 6: INK 2: C
  20 FOR y=0 TO 175 STEP 4
  30 PLOT 0,0
  40 BEEP 0.1,y/3
  50 DRAW 255,y
  60 PLOT 255,0
  70 DRAW -255,y
  80 PLOT 0,175
  90 DRAW 255,-y
 100 PLOT 255,175
 110 DRAW -255,-y
 120 NEXT y


0 OK, 0:1
```



The graphics cursor is moved to each of the four corners of the screen in turn. A line is DRAWn from each corner to a point on the opposite side of the screen. The line separation is determined by the STEP size in line 20. STEP 4 gives the best results. This makes the values of y 0, 4, 8, 12, 16, and so on, instead of y increasing by 1 in every circuit of the loop.

Using STEP to DRAW grids

One display that can be produced very easily using STEP is a games grid. You could make a grid just by DRAWing a series of criss-crossing lines against a contrasting background. But instead of specifying each line, you can use loops with STEP to do most of the work for you:

GAMES GRID PROGRAM

```
LS
  10 BORDER 6: PAPER 2: INK 7: C
  20 FOR y=25 TO 151 STEP 21
  30 PLOT 26,y
  40 DRAW 204,0
  50 NEXT y
  60 FOR x=26 TO 230 STEP 34
  70 PLOT x,25
  80 DRAW 0,126
  90 NEXT x


0 OK, 0:1
```

Lines 20 to 50 DRAW a series of lines across the screen at vertical intervals of 21. Then lines 60 to 90 DRAW lines down the screen with a separation of 34. (The co-ordinates and line separations given here may be easier to follow if you refer to the graphics layout grid on page 59.)

Next is a program that will produce a games grid with as many squares as you like. The program asks you what sort of grid you want (how many squares across and down) and then feeds those numbers into a subroutine describing a generalized grid pattern.

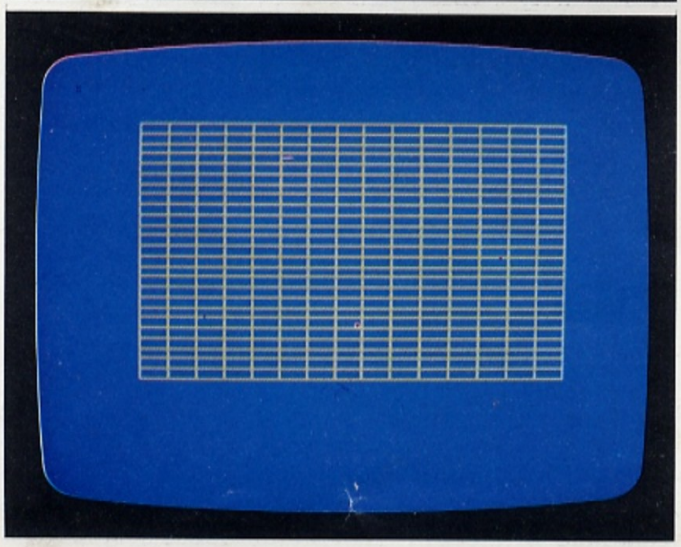
The program is divided into three sections. First the INPUT section (lines 10 to 100) asks you for the information necessary for the second half of the program. This takes the numbers you supply and uses them to calculate the separations of the lines DRAWn in the final display section (lines 140 to 240). It does this with two variables; w, which represents the number of squares across, is divided into the screen's display panel width in line 130. In the same line h, the number of

squares from top to bottom, is divided into the height. With this program you can produce an almost endless variety of grids containing either squares or rectangles, depending on the figures keyed in:

VARIABLE GRID PROGRAM

```
  10 BORDER 1: PAPER 7: INK 0: C
LS
  20 PRINT AT 9,9;"Do It Yoursel
f"
  30 PRINT AT 11,11;"Games Grid"
  40 PAUSE 150
  50 CLS
  60 PRINT AT 9,6;"Enter the gri
d size"
  70 PRINT AT 11,4;"How many squ
ares across?"
  80 INPUT w
  90 PRINT AT 13,5;"How many squ
ares high?"
 100 INPUT h
 110 PAPER 1: INK 6: CLS
 120 LET x=16: LET y=16
 130 LET a=224/w: LET b=144/h
 140 FOR n=1 TO w+1
 150 PLOT x,y
 160 DRAW 0,144
 170 LET x=x+a

scroll?
```

```
 180 NEXT n
 190 LET x=16: LET y=16
 200 FOR n=1 TO h+1
 210 PLOT x,y
 220 DRAW 224,0
 230 LET y=y+b
 240 NEXT n



0 OK, 0:1
```



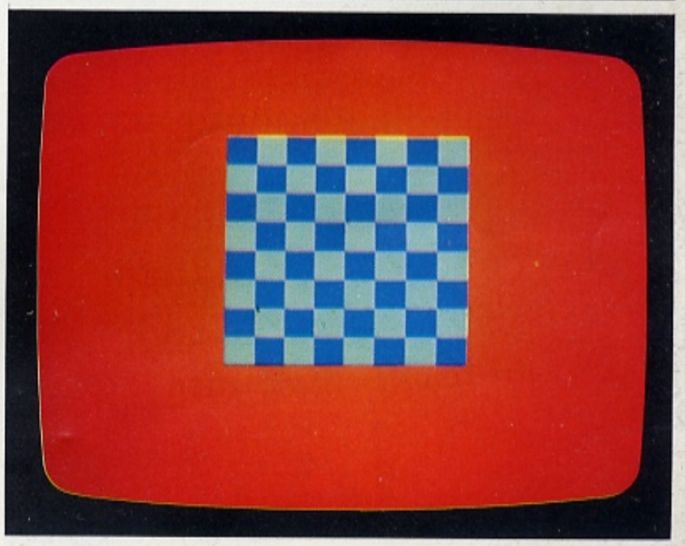
You can use this type of program as the basis for a chart, and then PRINT figures inside each square or rectangle. By using a technique which you will encounter on pages 22–23, you can then make the figures within the chart respond to different INPUTs.

How to PRINT a chessboard

To program the computer to PRINT a chessboard, you can PRINT alternate white and black squares, working your way down the board from top to bottom. It is then a simple matter then to add some colour to the display by introducing BORDER, PAPER and INK:

CHESSBOARD PROGRAM

```
  10 PAPER 2: BORDER 2: INK 5: C
LS
  20 FOR r=3 TO 18
  30 FOR c=8 TO 23
  40 PRINT AT r,c;"■"
  50 NEXT c
  60 NEXT r
  70 INK 1
  80 FOR r=3 TO 15 STEP 4
  90 FOR c=8 TO 20 STEP 4
 100 PRINT AT r,c;"■■"
 110 PRINT AT r+1,c;"■■"
 120 PRINT AT r+2,c+2;"■■"
 130 PRINT AT r+3,c+2;"■■"
 140 NEXT c
 150 NEXT r



K
```



Here PRINT AT is used to build up rows of dark blue squares over a cyan background. The program selects a blue foreground colour (INK 1) and sets the row co-ordinate to the top line of the board. Having PRINTed one blue square, it skips the second one, leaving it cyan, and PRINTs a blue square in the third position. The column STEP size is set to 4 to deal with this. As each loop draws two rows of squares, the row STEP size of 4 skips every other row.

INFORMATION FROM THE KEYBOARD

To key in new information while a program is RUNning you have – until now – used the command INPUT. With INPUT you must press ENTER after keying the information in. This technique has its disadvantages. Even when you know that it's necessary to use ENTER, you can occasionally forget, and using two keys in sequence also slows programs down. It is much more useful if the computer responds without waiting for you to press ENTER, just as arcade machines respond every time you press a control button. To make the Spectrum do this, you can use the keyword INKEY$.

ASCII codes

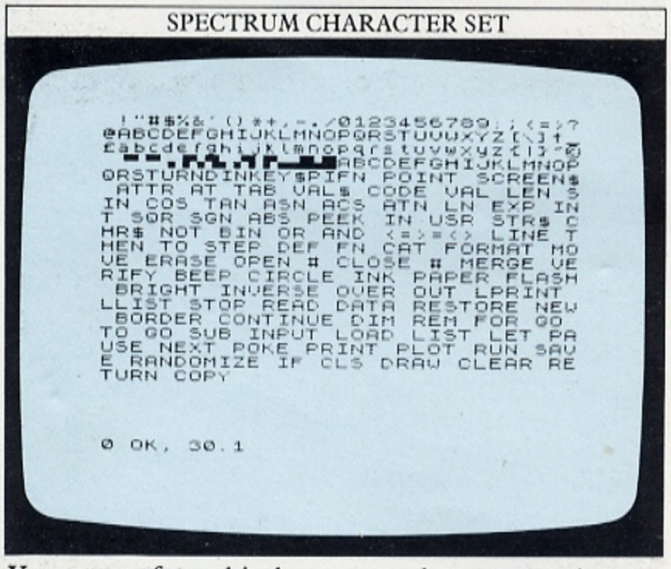
Keywords, numbers, letters, punctuation marks and mathematical symbols – in fact, all the symbols that the computer recognizes – are stored inside the computer as numbers from 0 to 255. Most computers, including the Spectrum, store these numbers according to a code called the American Standard Code for Information Interchange (ASCII). So, every computer that uses the ASCII standard stores a letter A as the decimal number 65, a "+" sign as 43, and the decimal number 1 as 49. (A complete table of ASCII codes is shown on page 60.)

You can see all the characters that the Spectrum uses by keying in a very simple program:

```
10 FOR n=33 TO 255
20 PRINT CHR$ n;
30 NEXT n
```

This makes the computer take each ASCII code in turn and then PRINT the character that the code represents, missing out the initial "blanks" and control codes:

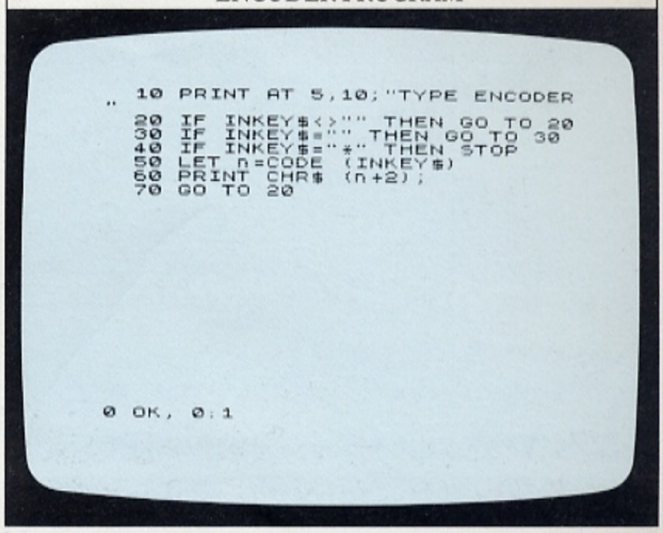
SPECTRUM CHARACTER SET



You can see from this that some codes represent just one letter or number, whereas others represent complete BASIC keywords.

The next program uses INKEY$ to respond to keys as they are pressed, and then uses ASCII numbers to PRINT in a simple code:

ENCODER PROGRAM



Line 10 PRINTs the title and lines 20 and 30 use INKEY$ to wait for you to release a key before continuing. Line 20 may puzzle you. Change it to:

```
20 REM IF INKEY$<>"" THEN GOTO 20
```

so that it doesn't work. You should find it impossible to press a key and get a single character on the screen. Unlike INPUT, INKEY$ doesn't wait for you – it checks if any key has been pressed and then moves on to the next line. So line 20 keeps on being carried out until you release a key, then line 30 keeps on RUNning until you press the next key.

Line 40 then tests the keyed character to see if it is an asterisk. If it is, the program ends. If not, line 50 converts your character into its ASCII code. Line 60 adds 2 to the code for the key you have pressed and then PRINTs the equivalent character. So, the program appears to PRINT nonsense when you key something in. There's no point in producing a coding program if you can't crack the code. You can quickly turn the encoder program into a decoder by inserting:

```
10 PRINT AT 5,10;"TYPE DECODER"
60 PRINT CHR$(n-2);
```

To see if it works try keying in the following when the program is RUNning:

```
ngctp"vq"rtqitco"ykvj"vjg"FM"uetggp"ujqv"ugtkgu
```

Testing your reactions with INKEY$

You can use INKEY$'s ability to check the keyboard in combination with ASCII codes to write a program that

enables you to time the speed of your reactions.

This program produces a random symbol on the screen, and then times how long it takes you to press the symbol it has selected:

REACTION TESTER

```
  10 BORDER 7: PAPER 7: INK 0: C
LS
  20 PRINT AT 8,7;"Test your ref
lexes"
  30 PRINT AT 10,10;"against the
"
  40 PRINT AT 12,8;"REACTION TES
TER"
  50 PAUSE 150: CLS
  60 PRINT AT 11,9;"Find this ke
y"
  70 LET k=33+RND*94: LET n=0
  80 BEEP 0.1,30
  90 PRINT AT 13,16;CHR$ (k)
 100 LET n=n+1
 110 IF INKEY$<>CHR$ (k) THEN GO
TO 90
 120 CLS
 130 PRINT AT 11,5;"You took ";(
INT ((n*2.68))/100;" seconds"


0 OK, 0:1
```

```
You took 1.25 seconds



0 OK, 130:1
```

Lines 10 to 50 PRINT the title frame and PAUSE for 3 seconds. Then the game begins. Line 70 sets k equal to a randomly generated number between 33 and 127, the ASCII code numbers for the Spectrum's letters and symbols (this excludes the graphics symbols and keywords). Line 70 also sets n equal to zero – n is used later to work out the time taken to press the correct key. Line 80 makes a sound to signal the beginning of the timing period. Line 90 PRINTs the character equivalent to the random code found by line 70. Line 110 checks whether you have pressed the correct key. If not, it cycles back to line 90, and on each loop, n is increased by 1.

If you do press the correct key, the time elapsed since the BEEP is calculated by (INT(n*2.68))/100 seconds. If you want to know why this is the line to use, try inserting the following line:

```
105 IF n=1000 THEN BEEP 0.3,20
```

in the program. This lets you time 1000 loops with your watch. They should take roughly 26.85 seconds, so each loop takes 0.02685 seconds. To find the elapsed time, multiply n by 0.02685. To limit the time PRINTed by line 130 to two decimal places, multiply (n*0.02685) by 100 (=n*2.685), take its integer value (to get rid of all figures after the decimal point) and finally divide the result by 100.

Using INKEY$ to control movement

INKEY$'s quick responses make it ideal for controlling movement on the screen. In the following program, instead of going through a predetermined series of movements, the computer waits until either the Z or M keys are pressed and then moves the character left or right respectively. Lines 170 and 180 stop the character being moved beyond either side of the screen:

INKEY$ ANIMATION PROGRAM

```
  10 BORDER 1: PAPER 2: INK 6: C
LS
  20 DATA 0,0,59,220,0,0,25,152,
64,2,15,240,32,4,63,252
  30 DATA 16,72,79,242,11,208,95
,250,13,176,129,129,31,248,131,1
93
  40 FOR n=0 TO 7
  50 READ a,b,c,d
  60 POKE USR "e"+n,a
  70 POKE USR "f"+n,b
  80 POKE USR "g"+n,c
  90 POKE USR "h"+n,d
 100 NEXT n
 110 LET c=16
 120 GO TO 210
 130 IF INKEY$<>"" THEN GO TO 13
0
 140 IF INKEY$="" THEN GO TO 140
 150 IF INKEY$="z" THEN LET c=c-
1
 160 IF INKEY$="m" THEN LET c=c+
1
 170 IF c<0 THEN LET c=0
 180 IF c>30 THEN LET c=30
 190 PRINT AT 10,c-1;"  "
 200 PRINT AT 11,c-1;"  "
 210 PRINT AT 10,c;"EF"
 220 PRINT AT 11,c;"GH"
 230 PAUSE 5
 240 GO TO 130


0 OK, 0:1
```

If you RUN the program, you will find that the character responds only to keys Z and M.

RANDOM NUMBERS AND ODDS

The Spectrum has two separate keywords which both deal with random numbers – RND and RANDOMIZE. As you have already seen, RND is used to generate random numbers. A line like this:

```
80 LET a=RND
```

will produce a random number between 0 and 0.99999999.

Actually that's not quite true. The Spectrum has a sequence of 65,536 different numbers between 0 and 0.99999999 stored in its memory. They are all mixed up, so that there is no obvious pattern. Because of this, RND is said to be a "pseudo-random" function. There is a sequence behind the numbers, but for most purposes the numbers can be taken as completely random. Although a number between 0 and 0.99999999 isn't much use, a line like the following produces whole (integer) random numbers in other ranges:

```
100 n=1+INT(2*RND)
```

Here the line produces a 1 or 2, for a coin toss program perhaps. 2*RND produces a number between zero and 1.99999998. INT rounds that down to the next lowest integer (0 or 1), and 1 is added to this to produce a 1 or 2. In the same way:

```
100 n=1+INT(6*RND)
```

produces numbers between 1 and 6.

How to start a "random" sequence

Debugging a program that uses RND can be difficult, because no two RUNs of the program produce the same result. It is easier to find the errors if the RND statements are made to produce the same numbers each time the program is RUN. First type in:

RND PROGRAM

```
120 PRINT AT 5,3;"PREDICTING RA
NDOM NUMBERS"
130 FOR n=1 TO 6
140 PRINT AT 2*n+6,11;14*RND
150 NEXT n
```

This very simple listing produces a series of six random numbers. Each time you RUN the program you should get a different series of numbers. The first screen below however produces a series which is always identical. It is programmed by adding:

```
125 RANDOMIZE 10
```

By using the keyword RANDOMIZE (which appears as RAND on the keyboard) each RUN produces the same numbers. This command works in a simple way – by controlling where RND starts selecting from its store of 65,536 numbers:

RANDOM/REPEATING DISPLAYS

```
PREDICTING RANDOM NUMBERS

          4.5458984
          4.953064
          7.4900513
          1.7611084
          6.0970154
          9.2851257


0 OK, 150:1
```

```
PREDICTING RANDOM NUMBERS

          0.17602539
          0.17602539
          0.17602539
          0.17602539
          0.17602539
          0.17602539


0 OK, 150:1
```

The second screen shows what happens when you move RANDOMIZE. Using the screen editor, change line 125 to line number 135 and delete 125. Now you should get the same number (0.17602539) every time. RANDOMIZE 10 starts the random sequence off at the tenth stored number on each RUN. You can follow

RANDOMIZE with any number between 1 and 65,536. RANDOMIZE on its own uses the time since the computer was switched on to set the starting point for RND. So, using RANDOMIZE with RND makes RND even more random!

Setting odds in adventure games

Whatever you do with RND, it only produces one number at a time. That's fine for coin-tossing or dice programs where each of the possible results (heads or tails, for instance) is as likely as any other result. But imagine you want to build probability into a program, so that some results are more likely than others. This is how many adventure games are written, making the programs much more interesting than ones that deal with predictable sequences of events.

The following program does this. It demonstrates how, in an adventure game, you can make it likely that the player will encounter some characters or symbols more frequently than others. You can also set the odds so that more often than not, nothing happens:

ADVENTURE ODDS PROGRAM

```
 10 DATA 0,0,1,128,1,64,123,128
,0,0,66,192
 20 DATA 0,0,3,192,3,160,123,12
8,0,0,153,224
 30 DATA 0,0,7,224,1,144,122,12
8,0,64,31,240
 40 DATA 0,0,15,240,1,8,50,128,
12,100,31,240
 50 DATA 7,224,15,240,7,196,2,1
28,18,216,15,240
 60 DATA 3,192,15,240,15,234,2,
128,34,240,15,224
 70 DATA 1,128,7,224,19,144,2,1
28,70,104,7,208
 80 DATA 1,128,1,128,123,128,5,
0,75,68,3,12
 90 FOR n=0 TO 7
 100 READ o,p,q,r,s,t,u,v,w,x,y,
z
 110 POKE USR "a"+n,o
 120 POKE USR "b"+n,p
 130 POKE USR "c"+n,q

scroll?
```

```
 140 POKE USR "d"+n,r
 150 POKE USR "e"+n,s
 160 POKE USR "f"+n,t
 170 POKE USR "g"+n,u
 180 POKE USR "h"+n,v
 190 POKE USR "i"+n,w
 200 POKE USR "j"+n,x
 210 POKE USR "k"+n,y
 220 POKE USR "l"+n,z
 230 NEXT n
 240 BORDER 2: PAPER 6: CLS
 250 PRINT INK 3;AT 5,5;"*******
*********"******"
 260 PRINT INK 3;AT 14,5;"******
*****************"
 270 PRINT INK 3;AT 19,14;"*****
"
 280 PRINT INK 0;AT 7,5;"AB";AT
8,5;"CD";AT 7,16;"EF";AT 8,16;"G
H";AT 7,26;"IJ";AT 8,26;"KL"
 290 LET k=0: LET d=0: LET t=0
 300 FOR n=1 TO 100

scroll?
```

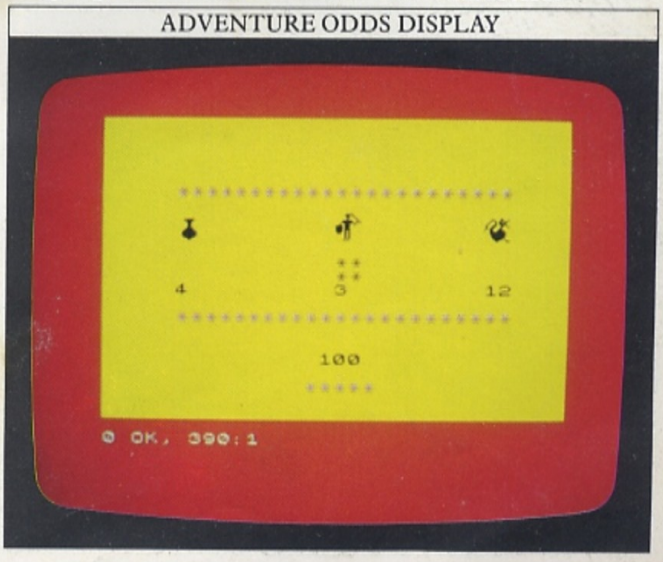
```
 300 FOR n=1 TO 100
 310 LET a=INT (20*RND+1)
 320 IF a=1 THEN LET A$="AB": LE
T B$="CD": LET t=t+1
 330 IF a=2 OR a=3 THEN LET A$="
EF": LET B$="GH": LET k=k+1
 340 IF a=4 OR a=5 THEN LET A$="
IJ": LET B$="KL": LET d=d+1
 350 IF a>5 THEN LET A$="**": LE
T B$="**"
 360 PRINT AT 10,16;A$;AT 11,16;
B$
 370 PRINT AT 12,5;t;AT 12,16;k;
AT 12,26;d;AT 17,15;n
 380 BEEP 0.2,15
 390 NEXT n


0 OK, 0:1
```

Lines 10 to 230 reprogram the keys A to L with the 12 characters that make up the three game symbols – a bag of treasure, an armoured knight and a dragon. Each symbol is made up from four characters, the first two above the second two.

Line 310 produces a number between 1 and 20. If it equals 1, then the treasure symbol is selected. If it equals 2 or 3, the knight is selected and if it equals 4 or 5, the dragon is selected. If it equals anything else, nothing (represented by four asterisks) is selected. So, a knight or a dragon are twice as likely to appear as a bag of treasure.

When you RUN the program, 100 random selections are made. The running totals of treasure, knights and dragons are shown building up on the screen. You can change the relative probabilities of the three symbols appearing by changing the numbers in lines 320 to 350. Inserting a PAUSE statement in the loop, or increasing the duration of the BEEP in line 380, slows the program down. By the end of each RUN, you can see how often the program has chosen each of the characters:

ADVENTURE ODDS DISPLAY

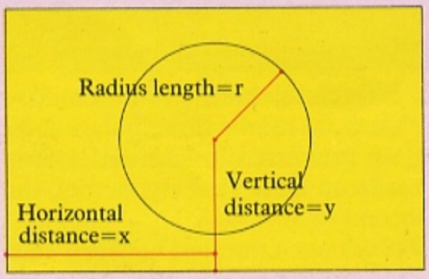


CURVES AND CIRCLES

On many personal computers, drawing curves and circles is quite an involved process. The Spectrum makes drawing circles very easy – its CIRCLE command does all the mathematics that you would otherwise have to do yourself. To produce a circle, all you have to do is specify the co-ordinates of the point at the centre of the circle – x and y, and the length of its radius, r. This lets you produce any size of circle in any position, within the limits of the graphics grid.

POSITIONING A CIRCLE

Any circle can be produced with the command CIRCLE x,y,r



The following direct command will produce a circle on the screen:

```
CIRCLE 128,88,50
```

The centre of this circle is at the centre of the screen (128,88) and it has a radius of 50 units. So, the horizontal diameter goes from 78 to 178 units across the screen and the vertical diameter from 38 to 138 up it.

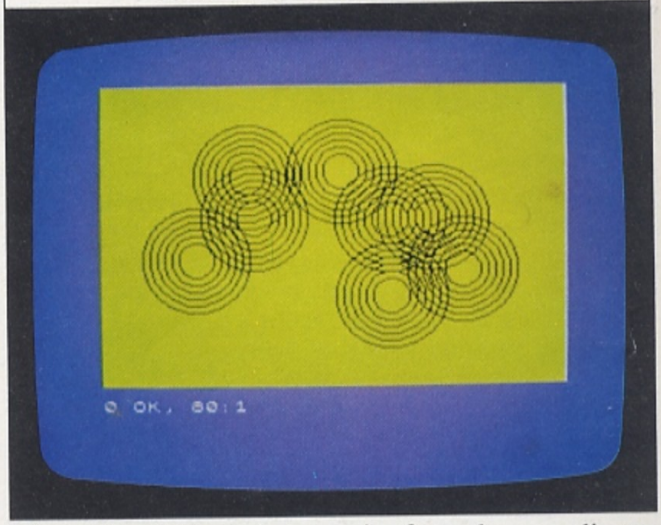
Producing patterns with CIRCLE

You can use the CIRCLE command to make up patterns by selecting the co-ordinates of a circle's centre at random. Here is a program that builds up a display by this method:

CONCENTRIC CIRCLES PROGRAM

```
  10 BORDER 3: PAPER 6: INK 0: C
LS
  20 FOR n=1 TO 8
  30 LET x=50+INT (155*RND)
  40 LET y=50+INT (75*RND)
  50 FOR r=10 TO 30 STEP 4
  60 CIRCLE x,y,r
  70 NEXT r
  80 NEXT n
```

CONCENTRIC CIRCLES DISPLAY



Lines 30 and 40 produce a pair of x and y co-ordinates at random so that neither is within 50 units of any of the screen edges. It is set like this because the program will then draw concentric circles up to a radius of 50. The loop at lines 50 to 70 repeatedly draws circles with the same centre but with gradually increasing radii (r). Line 80 starts the whole process off again but with a new pair of random co-ordinates.

You can change the maximum radius of the circle and the STEP size between radii by changing line 50:

```
50 FOR r=10 to 20 STEP 3
```

or even STEP 2, which produces smaller patterns.

Drawing arcs and waves

An arc (part of the edge of a circle) can be produced on the Spectrum's screen, but you cannot use the CIRCLE command to do this. But adding another number to the DRAW command – making it DRAW x,y,a – DRAWs an arc of a circle, starting from the last point PLOTted or DRAWn to, and finishing at a point specified by x,y. The "a" value is more complicated. This is the angle that the arc turns through (if you imagine it as being part of a circle). You can see the command at work if you key in these two lines (you will find the keyword PI above the M key):

```
PLOT 10,88
DRAW 230,0,PI/4
```

This produces a display like a piece of rope suspended at both ends and sagging in the middle. The first statement PLOTs a point near the left edge and halfway up the screen. The DRAW statement then DRAWs an arc from that point in an anticlockwise direction to (10+230),(88+0) or 240,88.

Whether the arc forms a shallow sag or a semicircle is determined by the size of "a". This angle is measured not in degrees but in radians. Radians are simply an alternative way of measuring angles based on the geometry of a circle. A complete circle is produced by turning around through 360 degrees. That is exactly equivalent to 2*PI radians. PI is an important mathematical constant, which has a value of 3.14159625 ... (you can see this by keying in PRINT PI). It is the ratio of the length of the circumference of a circle to its diameter. There are 2*PI radians in a circle. PI/2 radians are therefore equivalent to a quarter of a circle (90 degrees), PI/4 radians are equivalent to one eighth of a circle (45 degrees) and so on.

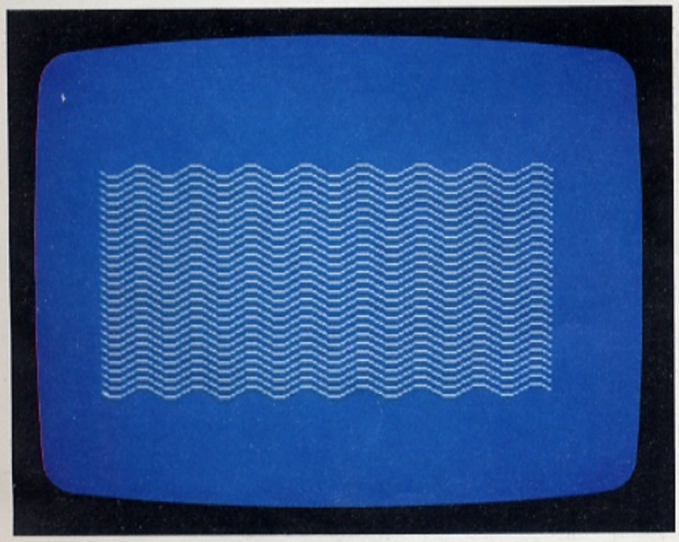
Now, without erasing the first arc, try adding:

```
PLOT 10,88
DRAW 230,0,-PI/4
```

The second arc is DRAWn between the same two points to form an eye-shaped figure. You can build up shapes using arcs to produce wave-like patterns:

WAVE PATTERN PROGRAM

```
  10 BORDER 1: PAPER 1: INK 7: C
LS
  20 LET y=128
  30 FOR f=1 TO 26
  40 FOR x=0 TO 225 STEP 30
  50 PLOT x,y
  60 DRAW 15,0,PI/2: DRAW 15,0,-
PI/2
  70 NEXT x
  80 LET y=y-5
  90 NEXT f



0 OK, 0:1
```



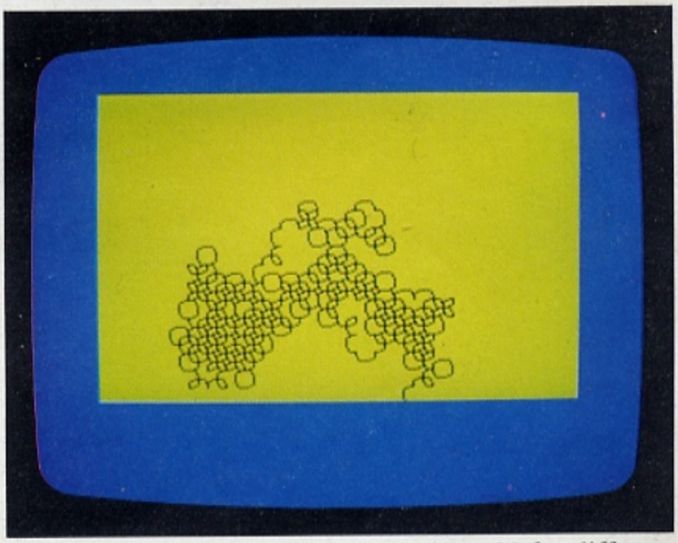
Line 20 sets the first y co-ordinate value of the waves. Line 40 STEPs the x co-ordinate across the screen. It's important that the STEP size is the same as the combined x lengths of the two parts of the wave formed by each loop. Line 50 PLOTs a point at the x,y co-ordinates set at the beginning of each loop. Line 60 DRAWs a small arc of a circle and then it goes on to DRAW a second arc of the same size. The first arc sags downwards, the second upwards and so on, building up a row of waves. When that is complete, line 80 moves the y co-ordinate downwards. The program continues until the waves reach the bottom of the screen.

Programming creeping curves

On the previous page CIRCLE was used to produce concentric circles at random. Here is a program that DRAWs semicircles, also at random, each of which goes up, down, left or right, to creep over the screen:

CREEPING CURVES PROGRAM

```
  10 BORDER 1: PAPER 6: INK 0: C
LS
  20 PLOT 128,88
  30 LET x=0: LET y=0
  40 LET q=1+INT (4*RND)
  50 IF q=1 THEN LET x=10
  60 IF q=2 THEN LET x=-10
  70 IF q=3 THEN LET y=10
  80 IF q=4 THEN LET y=-10
  90 DRAW x,y,PI
 100 GO TO 30



0 OK, 0:1
```



Try changing the arc size in lines 50 to 80 for different effects. You'll find that if you reduce the arc size to 4, it looks more like a letter v.

NATURAL GRAPHICS

Most of the shapes you have DRAWn so far have involved working out beforehand where the key points of the shape (the corners of a square, for example) should lie, and then DRAWing lines between them. Fortunately, much more complicated geometrical patterns can be DRAWn by using functions that do all the work for you. You don't have to PLOT a single point. Using the functions SIN (short for the mathematical term "sine") and COS (short for "cosine"), you can produce some spectacular graphics with quite short programs.

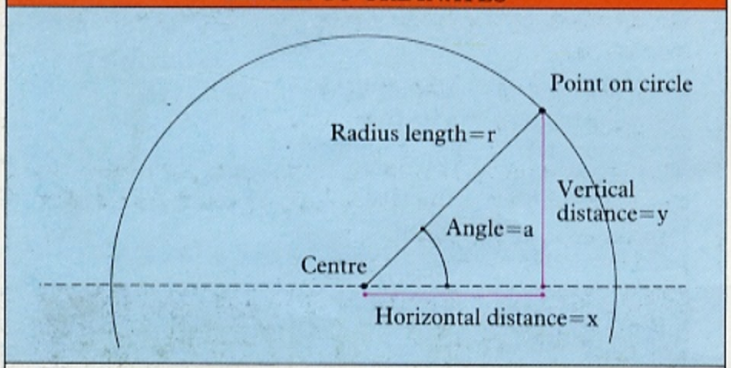
All of the programs on these two pages use a combination of SIN and COS, so it's worth trying to understand what these commands actually mean.

Another way to PLOT circles

On page 16, you saw how the command CIRCLE can be used to produce a circle. You can actually PLOT a circle by another method, using SIN and COS. Although the program is longer, it then enables you to branch out into some much more elaborate graphics.

If you sketch out part of a circle, you can relate each point on the circle to an angle at the circle's centre.

CIRCLE CO-ORDINATES



The distances x and y can be expressed in another way, as multiples of the angle and SIN or COS. Every angle has its own value of SIN and COS, and the co-ordinates of any point on the circle can be written as:

r*COS(a),r*SIN(a)

Once you know this, you can get the computer to PLOT a circle. By working through all the possible values of the angle "a" with a loop, a program can use the functions SIN and COS to PLOT the co-ordinates of every point on the circle.

You can, of course, do this much more easily by using CIRCLE. However, if you key in the next program, you can develop it to produce graphics that CIRCLE itself cannot produce. Line 20 sets the radius of the circle to 80. The angle has to vary from zero to a full circle (360 degrees):

CIRCLE PROGRAM

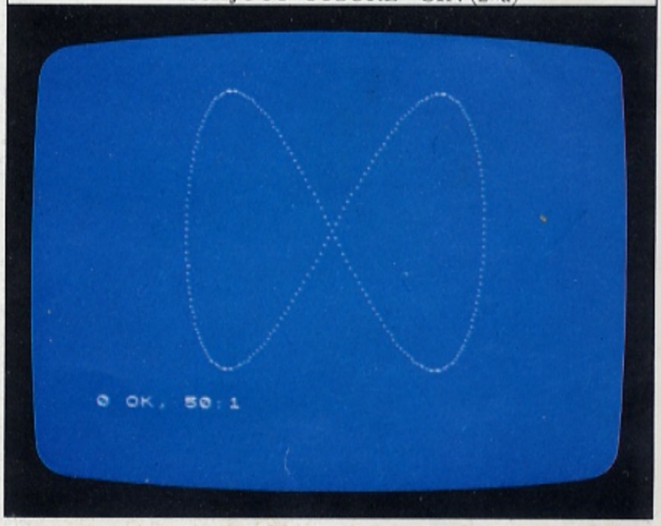
```
 10 BORDER 1: PAPER 1: INK 7: C
LS
 20 LET r=80
 30 FOR a=0 TO 2*PI STEP PI/120
 40 PLOT r*COS (a)+128,r*SIN (a
)+88
 50 NEXT a


0 OK, 0:1
```

Because the BASIC functions SIN and COS operate on angles measured in radians, like the command DRAW x,y,a on page 16, the range of the angles again uses the keyword PI. The STEP of PI/120 (line 30) is chosen so that the dots are relatively close together but the program doesn't take too long to RUN.

Patterns with SIN and COS

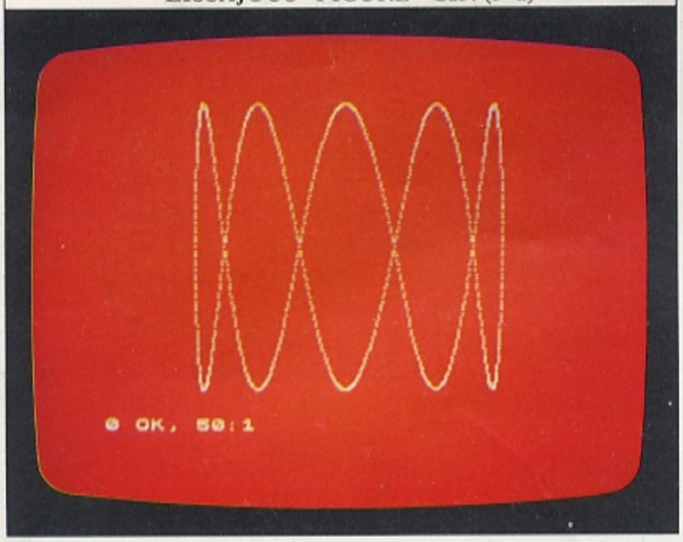
The circle program, above, produces a circle because the x and y co-ordinates vary exactly out of step with each other. When x is zero, y is at its maximum value and vice versa. What would happen if you varied x and y at different rates? Try altering the angle after the SIN command. Here the angle has been doubled in the first display, and multiplied by five in the second:

"LISSAJOUS" FIGURE – **SIN** (2*a)



The number of loops in each display depends on how many times the angle after SIN has been multiplied:

"LISSAJOUS" FIGURE – **SIN** (5*a)



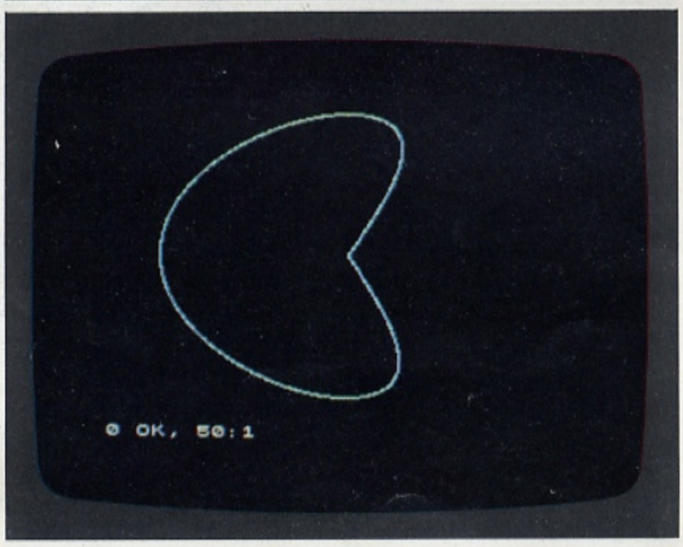
Complex curves

Now you can make a different sort of change:

KIDNEY PROGRAM

```
  10 BORDER 0: PAPER 0: INK 5: C
LS
  20 LET r=10
  30 FOR a=0 TO 2*PI STEP PI/480
  40 PLOT (r*COS (a)*r*SIN (0.5*
a))+128,8*r*SIN (a)+88
  50 NEXT a
```

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



In this program, the colours and STEP size have been changed as well as the co-ordinate values. The program will continue DRAWing if you increase the range of angle values. The next program roughly doubles the range. It also accelerates the RUNning speed by DRAWing short lines instead of PLOTting individual points as in the CIRCLE program opposite:

HOURGLASS PROGRAM

```
  10 BORDER 3: PAPER 2: INK 7: C
LS
  20 LET r=60
  30 LET x=0: LET y=0: PLOT 128,
88
  40 FOR a=0 TO 12.6 STEP 0.2
  50 LET i=r*COS (a)*SIN (0.5*a)
  60 LET j=r*SIN (a)
  70 DRAW i-x,j-y
  80 LET x=i: LET y=j
  90 NEXT a
```

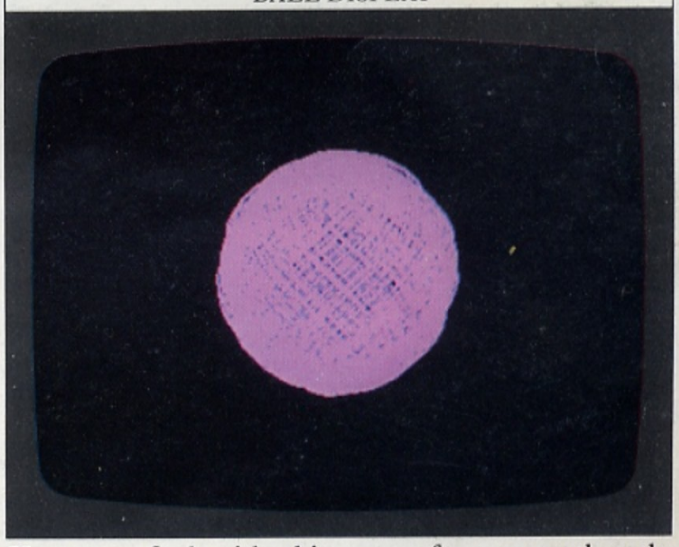
```
0 OK, 0:1
```

You can experiment with this program to produce a whole variety of more complicated shapes. The next design is produced by changing the colours and lines 40 and 50 to:

```
40 FOR a=0 TO 1000 STEP 0.1
50 LET i=r*COS(a)*SIN(0.98*a)
```

The shape starts off by being quite open but eventually the program fills in the circle (this screen shows it after half an hour of RUNning):

BALL DISPLAY



You may find with this type of program that the graphics resolution cannot cope with too much detail as every curve has to be shown as a series of dots.

THE COMPUTER CLOCK

Time delays are among the most frequently used commands in computer programs. One of the problems of working with a micro is that it often RUNs through programs too quickly. Time delays slow programs down to a speed that is useful. A delay may keep some text on the screen for a moment, long enough to read it, before it disappears and the program continues, or it may slow down an otherwise too rapid sequence of images – in an animation program for example.

The quickest and easiest time delay to write into a program is the PAUSE statement. It takes the form of the following program line:

```
100 PAUSE 50
```

where the number after PAUSE represents the time delay in fiftieths of a second. It's a simple matter to RUN a program containing the above PAUSE statement and see for yourself if the delay is long enough. If not, then increase and test the number following PAUSE again. It's approximate, but convenient. However, if you want to time an event in a program you can use a clock that has been built into the Spectrum itself.

Using the internal clock

Some computers have a timing facility, a clock that runs regardless of whatever the user's programs are doing. The Spectrum doesn't have any command or function that enables you to use a timing facility with a single program statement.

However, in common with every other personal computer, the Spectrum has an internal clock which synchronizes its activities. Linked to this is a counting device. Three "addresses", or slots, in the Spectrum's memory count the number of television pictures (or frames) that have been produced since the computer was switched on. The first address (numbered 23672) counts frames up to a maximum of 255. Then it is reset to 0 and the contents of the next address (23673) are increased by 1. When that reaches 255, it is reset to 0 and the contents of the third address (23674) are increased by 1. Because, in Europe, television pictures are produced at the rate of 50 per second (the rate is 60 per second in the United States), the first address (23672) in effect counts in fiftieths of a second, the next address in 5.12-second intervals and the third address in 1310.72-second intervals.

If this internal clock/counter is only accurate to a fiftieth of a second, you might wonder what the advantage is of using it instead of the much more convenient PAUSE statement, which has roughly the same order of accuracy. The reason is that PAUSE is a BASIC statement which makes up part of a program. The program is unable to do anything else during the PAUSE. It can never be more than a temporary interruption to the program.

The frame counter works in a different way. It keeps on counting whatever the program is doing – with two exceptions. If you make the computer BEEP or you use any piece of hardware connected to the computer – a cassette recorder or printer, for example – the frame counter stops while you are doing so, and then resumes its count. So, if you want to use the frame counter as a clock, it would lose time during these operations.

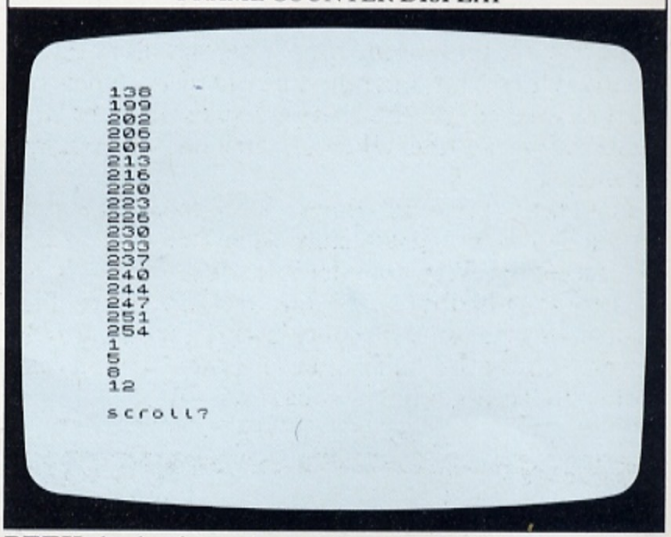
Timing with the frame counter

To see the frame counter working, try keying in the following two lines:

```
10 PRINT PEEK 23672
20 GOTO 10
```

The screen should fill up with increasingly large numbers (up to 255), until the "scroll?" prompt appears, as shown on the display:

FRAME COUNTER DISPLAY



PEEK looks into the address specified to see what number is stored there. It's the counterpart of POKE, which puts a number into an address. PEEK 23672 means "the contents of" address number 23672. Note that the numbers PRINTed aren't consecutive. There is a difference of three or four between each pair. That's because the PRINT and GOTO statements themselves take some time to be carried out, so the program isn't able to "catch" every single fiftieth of a second "tick" of the frame counter. That can cause problems when you want to use the frame counter to count in small time intervals, as you can see from the following program. When you RUN it, a BEEP should sound every second. The variable t is set equal to the contents of 23672:

FRAME COUNTER "CLOCK"

```
  10 LET t=PEEK 23672
  20 IF PEEK 23672-t=50 THEN BEE
P 0.05,20: GO TO 10
  30 GO TO 20




0 OK, 0:1
```

When the difference between the PEEK 23672 and the fixed t reaches 50, then one second has passed and the BEEP should be sounded.

Unfortunately, as you probably discovered, when you RUN this program, it hardly produces more than a couple of BEEPs. The trouble is that PEEK 23672−t is unlikely to be equal to 50 at the instant when line 20 is being carried out. Also, the BEEP statement stops the frame counter for a twentieth of a second every time it sounds. This sort of program can be used as a timer, but the time interval (the time between ticks) must be large compared to the time taken to carry out the program statements.

The next program times a 5-second interval, although it is not completely accurate. The second address counts every time the first address is full – that is, every 256 fiftieths of a second, or 5.12 seconds. That is adequate for a timer that only has to be accurate to the nearest minute or half minute. Line 10 resets the address to 0 every time the program is RUN:

5-SECOND TIMER

```
  10 POKE 23673,0
  20 LET t=PEEK 23673
  30 IF PEEK 23673>t THEN PRINT
"tick"
  40 GO TO 30




0 OK, 0:1
```

Using time in programs

Because PEEKing the frame computer requires a number of calculations, you can substitute a function FN t() to represent the number of seconds that have elapsed since the computer was switched on. A variable T is set equal to this, and, later in the program, when FN t() exceeds T, the seconds display is increased by 1. T is then reset to FN t() for the next count. A program can also incorporate tests to increase the minutes and hours displays when necessary.

This relatively accurate timing is of more value in programs. Here is an example of one way to use it, in a program that times the speed of your reactions:

REACTION TIMER

```
  10 DEF FN t()=65536*PEEK 23674
+256*PEEK 23673+PEEK 23672
  20 PRINT AT 5,6;"Time your rea
ctions"
  30 PRINT AT 10,10;"Press a key
"
  40 PRINT AT 12,4;"when you hea
r the tone"
  50 PAUSE 100+4*(INT 100*RND)
  60 LET T=FN t(): BEEP 0.1,25
  70 PRINT AT 16,6;"Your time st
arts now"
  80 IF INKEY$="" THEN GO TO 80
  90 CLS
 100 PRINT AT 8,12;"You took";AT
 10,13;(FN t()-T)/50;AT 12,12;"s
econds"
 110 PAUSE 150
 120 CLS : GO TO 20


0 OK, 0:1
```

```
Time your reactions


Press a key
when you hear the tone


Your time starts now
```

The time function FN t() is defined in line 10. It PEEKs all three frame counters. The final division by 50 is omitted, so that the function counts in fiftieths of a second, instead of in seconds. Line 50 causes a random delay of at least 2 seconds before the timing period starts. Line 80 waits until you press a key before clearing the screen (line 90) and then calculating and PRINTing your reaction time.

USING ARRAYS

An array is a way of storing facts and/or figures in the computer's memory in the form of a table, so that you can locate any one item in the table without having to go through all the others first. Each item in an array is specified by one or more numbers. In the following array, each item is given a pair of co-ordinates which identify it and nothing else:

	1	2	3	4	5	6
1	FRED	KATE	JOHN	JANE	ALAN	JUDY
2	100	250	840	125	223	691

This is a 6×2 array, so-called because it has 6 columns and 2 rows. Item (2,2) is 250, item (1,3) is JOHN, and so on. Because two numbers are needed to identify each item, this array is known as a two-dimensional array. If it was composed of only one row of names or numbers, it would only need one number to identify each item and so it would be called a one-dimensional array. The BASIC keyword DIM is used to tell the computer how big an array is to be.

A one-dimensional array can be used to store a list of frequently used numbers or strings:

ONE-DIMENSIONAL ARRAY

```
  10 DATA "January","February","
March","April","May","June","Jul
y","August","September","October
","November","December"
  20 DIM m$(12,9)
  30 FOR n=1 TO 12: READ m$(n):
NEXT n
  40 CLS
  50 FOR n=1 TO 12
  60 PRINT AT 4+n,12;m$(n)
  70 NEXT n


0 OK, 0:1
```

Here line 20 tells the computer that the array m$ is 12×9 entries (9 is the maximum number of letters in the name of any one of the 12 months). The program PRINTs out the list of the months of the year given in line 10. Although there are easier ways of doing this, later on in a program, you might want to match up a month with other information or the result of calculations. Using this listing, you can pick out any month by using m$(n) where n is the month number. When the program is RUN, the display it should produce looks like this – a month chart ready for more information:

ONE-DIMENSIONAL ARRAY DISPLAY

```
          January
          Febryary
          March
          April
          May
          June
          July
          August
          September
          October
          November
          December


0 OK, 70:1
```

How to add a dimension

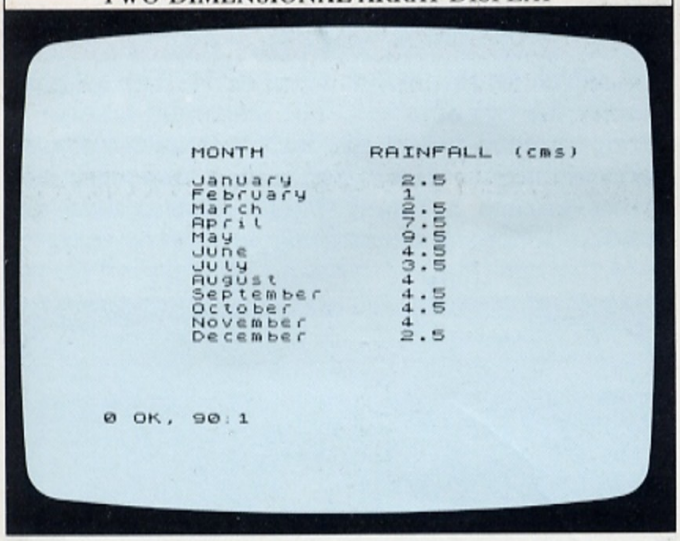
Now you build upon the calendar array program to make it do something useful. Add a second array, a numeric array, so that you can list some values against each month. The following program uses the array to represent monthly rainfall totals.

The table now has two headings. You don't have to PRINT all the members of the string array – m$(n) – before moving on to select the numeric array – r(n). Line 80 takes one item from each array. As these are to be PRINTed on consecutive rows of the screen, they can easily be identified by relating them to the row number. For each value of n, m$(n) and r(n) are PRINTed at different column numbers of row (5+n):

TWO-DIMENSIONAL ARRAY

```
  10 DATA "January","February","
March","April","May","June","Jul
y","August","September","October
","November","December"
  20 DATA 2.5,1.0,2.5,7.5,9.5,4.
5,3.5,4.0,4.5,4.5,4.0,2.5
  30 DIM m$(12,9): DIM r(12)
  40 FOR n=1 TO 12: READ m$(n):
NEXT n
  50 FOR n=1 TO 12: READ r(n): N
EXT n
  60 PRINT AT 4,6;"MONTH";AT 4,1
8;"RAINFALL (cms)"
  70 FOR n=1 TO 12
  80 PRINT AT 5+n,6;m$(n);AT 5+n
,20;r(n)
  90 NEXT n


0 OK, 0:1
```

TWO-DIMENSIONAL ARRAY DISPLAY



Writing tables with arrays

You can now make a table that is more ambitious:

TAX TABLE PROGRAM

```
  10 DATA 1.98,2.40,5.60,1.05,4.
35,2.99,1.92,7.20,5.45
  20 DATA 20,19,11,45,15,15,24,4
,6,44,84,108,164,204
  30 LET t=15
  40 DIM q(9,2)
  50 FOR c=1 TO 2
  60 FOR r=1 TO 9
  70 READ q(r,c)
  80 NEXT r
  90 NEXT c
 100 PLOT 4,12: DRAW 248,0: DRAW
0,160: DRAW -248,0: DRAW 0,-160
 110 FOR n=12 TO 156 STEP 16
 120 PLOT 4,n: DRAW 248,0
 130 NEXT n
 140 FOR n=1 TO 5
 150 READ x
 160 PLOT x,12: DRAW 0,160
 170 NEXT n
 180 PRINT AT 1,1;"ITEM";AT 1,6;
"COST";AT 1,11;"No";AT 1,14;"SUB

scroll?
```

```
 190 FOR n=1 TO 9
 200 PRINT AT 2*n+1,2;n;AT 2*n+1
,6;q(n,1);AT 2*n+1,11;q(n,2)
 210 PRINT AT 2*n+1,14;q(n,1)*q(
n,2);AT 2*n+1,21;(INT (((q(n,1)*
q(n,2)*t/100)+0.005)*100))/100;A
T 2*n+1,26;(INT (((q(n,1)*q(n,2)
*(1+t/100))+0.005)*100))/100
 220 NEXT n
 230 PAUSE 150
 240 PRINT AT 21,7;"Try a new ta
x rate": INPUT t
 250 PRINT AT 21,7;"
     "
 260 GO TO 180

0 OK, 0:1
```

In this financial planning program, the columns are interrelated and you have the option of changing some of the information displayed, should you need to do so.

Line 10 contains the DATA for the first part of the array, a series of prices, and line 20 the DATA for a second part – a series of quantities. Line 20 also contains some co-ordinates which will be used later in the program. Lines 40 to 90 dimension the 9×2 array and READ in its DATA. Lines 100 to 170 simply DRAW the grid of lines that frames the DATA. The co-ordinates of the bottom end of the vertical lines in the grid are stored in line 20. Line 180 PRINTs the column headings.

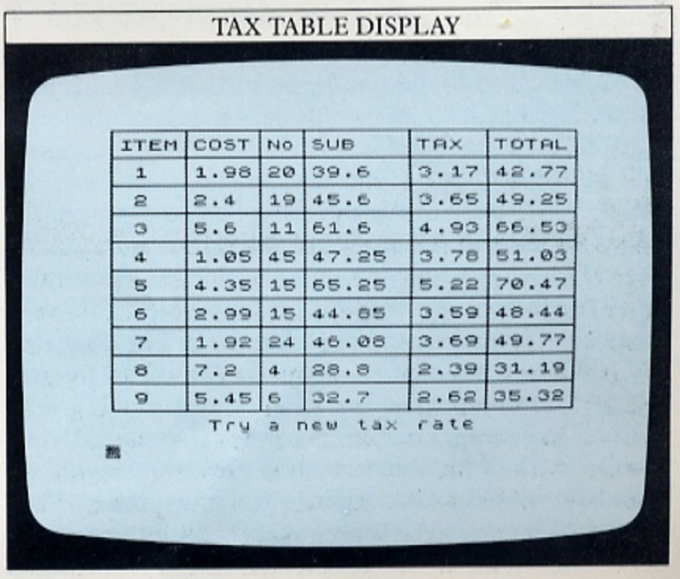
The DATA is PRINTed in the grid by lines 190 to 220. It is to be PRINTed every other line from rows 3 to 19. The number of the item, n, (from 1 to 9) is related to this by:

row=2*n+1

and this appears throughout lines 200 and 210 in the PRINT AT statements. The last two items in line 210 look particularly complex. If the subtotal was 8.25, the tax would be calculated as 0.15*8.25=1.2375 – too many decimal places. To solve that, the tax is multiplied by 100, the INTeger value of it is taken (removing all the decimal places) and it is divided by 100 again. The 0.005 is added to ensure that the final figure is rounded down to the nearest unit.

Lines 240 to 260 invite you to enter a new tax rate. If you do and press ENTER, all the figures in the table that use the tax rate are recalculated. This instant recalculation facility is the principle behind a type of financial planning program called a spreadsheet. Inter-related columns of figures representing income, raw material/production costs, overheads and so on can be entered. Then the effects of changing one or more of these parameters can be observed as all the totals are recalculated throughout the display:

TAX TABLE DISPLAY



WORKING WITH WORDS 1

Almost every program you have looked at so far has taken numerical data and performed some calculations to arrive at a result. But the strings used have almost invariably been left in their original order. However, as you saw in the reaction test program on page 13, the computer stores letters as well as numbers as ASCII codes (a table of the Spectrum's ASCII codes appears on page 60). Because these codes have numerical values, the computer can examine strings and then reorder them in a similar way to numbers.

How to rearrange numbers

To see how words – or any strings – are sorted or reordered, it's helpful if you understand how the same thing is done with numbers. Here is a program which asks you for six numbers, and then rearranges them in numerical order. You could easily extend it to deal with many more numbers:

NUMERICAL SORTER PROGRAM

```
 10 DIM a(6)
 20 PRINT AT 5,10;"NUMBER SORT"
 30 FOR n=1 TO 6
 40 PRINT PAPER 0; INK 7;AT 10,
2+4*n;n
 50 PRINT AT 20,1;"Type in a 1
or 2 digit number"
 60 INPUT a(n): PRINT AT 12,2+4
*n;a(n)
 70 NEXT n
 80 FOR t=1 TO 5: FOR n=1 TO 5
 90 IF a(n+1)<a(n) THEN GO SUB
150
100 NEXT n: NEXT t
110 FOR n=1 TO 6
120 PRINT AT 16,2+4*n;a(n)
130 NEXT n
140 PRINT AT 20,1;"-------- SORT
COMPLETE --------": STOP
150 LET b=a(n): LET a(n)=a(n+1)
: LET a(n+1)=b
160 RETURN

0 OK, 0:1
```

Line 50 asks you to type in a one- or two-digit number. Each time you do this and press ENTER, the next prompt (from 1 to 6) is PRINTed above the next INPUT position. This is where your chosen number will be PRINTed by line 60. All the numbers you ENTER are loaded into an array, so that they can be tagged with a number to identify them.

After you have ENTERed six numbers, the sorting procedure begins automatically at line 80. First look at the n loop (from the second statement in line 80 to line 100). For each value of n, a(n+1) is compared to a(n). If a(n+1) is the smaller, line 150 reverses them. So, on the first pass round the loop (n=1), if your first two numbers were 34 and 16, the reversal subroutine would be called, because the second number is smaller than the first. It would set b equal to a(1) – 34 in this case – set a(1) equal to a(2) – 16 here – and finally set a(2) equal to b, which on this loop is equal to 34. This juggling reverses the two numbers. The maximum number of sortings needed to put the list of numbers into the correct numerical order is five, so the t loop repeats the sorting process 5 times. These screens show the numbers before and after sorting:

NUMERICAL SORTER DISPLAYS

```
Number sort

 1   2   3   4   5   6
23   5   67  55  29  40


Type in a 1 or 2 digit number
```

```
NUMBER SORT

 1   2   3   4   5   6
23   5   67  55  29  40

 5   23  29  40  55  67

-------- SORT COMPLETE --------

9 STOP statement, 140:2
```

This reversal process can be used to manipulate strings in the same way. It's easy to see how the computer can compare two numbers and test whether the second is smaller than the first. We do it ourselves all the time – when comparing prices for example. But the computer can also decide whether "London" is less than "New York" – that is, whether one comes before the other in the alphabet. A line like:

```
50 IF "New York"<"London" THEN GOSUB 300
```

doesn't seem to make much sense at first glance. But because the computer stores a string like New York or

London as a series of numbers, these numbers can be compared and reordered. So comparisons like a<b and Stockholm>Paris both make perfectly good sense in BASIC.

Rearranging in alphabetic order

One of the most useful applications for string sorting is rearranging into alphabetic order. The following program shows one method – it works on strings already built into the program, but you can easily adapt the program to accept different strings by using INPUT:

ALPHABETIC SORTER PROGRAM

```
  10 DATA "Paris","Stockholm","N
ew York","London","Rome","Amster
dam"
  20 DIM a$(6,9)
  30 FOR n=1 TO 6: READ a$(n): N
EXT n
  40 PRINT AT 5,9;"Alpha sort 1"
  50 GO SUB 120
  60 FOR t=1 TO 5
  70 FOR n=1 TO 5
  80 IF a$(n+1)<a$(n) THEN GO SU
B 150
  90 NEXT n
 100 GO SUB 120
 110 NEXT t: STOP
 120 FOR n=1 TO 6
 130 PRINT AT 2*n+7,10;a$(n)
 140 PAUSE 25: NEXT n: RETURN
 150 LET b$=a$(n)
 160 LET a$(n)=a$(n+1)
 170 LET a$(n+1)=b$
 180 RETURN

0 OK, 0:1
```

Line 20 dimensions a string array using the method demonstrated on page 22. Line 30 then READs the contents of line 10 – a series of capital cities. The subroutine at lines 120 to 140 PRINTs the cities in their original order. Then the sorting routine moves the strings around until they are in alphabetic order. You could add a PAUSE to slow things down so that you can actually follow what is going on:

ALPHABETIC SORTER DISPLAYS

```
Alpha sort 1

 Paris
 Stockholm
 New York
 London
 Rome
 Amsterdam
```

```
Alpha sort 1

 Amsterdam
 London
 New York
 Paris
 Rome
 Stockholm

9 STOP statement, 110:2
```

The sorting routine at lines 150 to 180 is basically the same as that used to sort numbers. The variables used here are of course string variables, but the computer uses the same mathematical comparisons as it uses when operating with numbers. The program then STOPs at line 110.

Note that the temporary store used in both the number and string reordering programs (b and b$) need not be an element of an array. The variable is only used once on each sorting and then is not required again.

How to turn strings into numbers

The Spectrum uses a number of keywords which enable it to use numerical information from strings. In addition to taking string characters and comparing their ASCII values, the computer can decide how long a string is. To program this, you need to use the keyword LEN. The following program line for example produces a number:

```
100 n=LEN"Spectrum"
```

Here n is equal to 8, the LENgth of the string. You can use this in programs to reject an INPUT word or name that is too long, or make the computer take different courses of action depending on how long certain strings are. You can add LENs together to find out the length of a number of names or any other piece of text.

The Spectrum also has a keyword which operates on strings that are themselves numbers. VAL coverts a number-string containing a calculation into a number. The number produced by this is itself the result of the calculation. For example:

```
150 LET a$="2*34.5*0.3"
160 PRINT a$;"=";VAL a$
```

PRINTs both the calculation and its result – a useful programming technique. You can use this command to evaluate a string and then pass this value on to another part of a program.

WORKING WITH WORDS 2

Until now, you have treated strings – or words that make up strings – as indivisible units. Some of the programs so far have added strings together, but none of them have "looked inside" the quotation marks that begin and end every string to work on the characters that are there. With the Spectrum you can take strings apart and reassemble their characters in a number of different ways. This means that you can program the computer to take out part of a word or group of words and examine it – a process that can be very useful.

Most computers that work with BASIC have a family of commands that can be used to manipulate strings – LEFT$, RIGHT$, MID$ and so on. They are used to pick out the first, last or middle character of a string respectively. Although the Spectrum doesn't have any of these commands on its keyboard, it can do everything that these commands do, if you know how to program it.

How to cut up words

You can make the Spectrum break into a word by slicing parts off the string. It's fairly straightforward. To see how to do it, first type in the listing on the following screen:

STRING SLICER PROGRAM

```
 10 LET a$="Slicing words with
the Spectrum"
 20 CLS
 30 FOR n=1 TO 31
 40 PRINT a$( TO n)
 50 NEXT n


0 OK, 0:1
```

The special technique here is in line 40, where a TO command appears as part of a string variable. For each value of n, line 40 PRINTs a string n characters long, from the first character to the nth character. So, the first line contains the string "S", the second line "Sl" and so on, until n equals the length that is set. With this program you can use any string – a group of words, numbers or other symbols; it's best if the value of n is not more than one screen line (32 characters). If you use a different string, make sure that the maximum value of n in line 30 is no more than the length of your string:

STRING SLICER DISPLAY

```
S
Sl
Sli
Slic
Slici
Slicin
Slicing
Slicing
Slicing w
Slicing wo
Slicing wor
Slicing word
Slicing words
Slicing words
Slicing words wi
Slicing words wit
Slicing words with
Slicing words with
Slicing words with t
Slicing words with th
Slicing words with the

scroll?
```

You can use this kind of technique to pick out strings that all begin with the same letter or word, and then perhaps PRINT them out in a series of lists.

The reverse effect is just as easy to produce. Try adding the following lines to the program, then RUN it to see what happens:

```
60 FOR n=1 TO 31
70 PRINT a$(TO 32-n)
80 NEXT n
```

Now, as n increases from 1 to 31 on each loop, the length of the string PRINTed decreases from 31 characters to only 1.

Picking out parts of a phrase

Now you can explore this technique. Type in and RUN the listing on the following screen:

SELECTIVE STRING SLICER PROGRAM

```
 10 LET a$="DK Screen Shot Seri
es"
 20 CLS
 30 PRINT AT 5,9;"String Choppe
r"
 40 PRINT AT 7,6;a$
 50 PRINT AT 10,6;a$( TO 2)
 60 PRINT AT 12,6;a$(4 TO 14)
 70 PRINT AT 14,5;a$(15 TO )


0 OK, 0:1
```

You aren't limited to dealing with the first n characters of a string. In fact, you can take any consecutive group of characters from a word or sentence. In this program line 50 works in the same way as line 40 of the slicing program. Line 60 forms a string from characters 4 to 14 out of the middle of a$. Finally, line 70 forms a third string from character number 15 to the end of a$. Although these three "substrings" are formed from parts of a$, a$ itself is still intact. This technique lets you take a group of words and pick out any of them for use on their own in a program.

Word games with string commands

The next program shows how you can use these methods of handling words in a game. It's a computerized word-guessing contest in which one player enters a word and the other has to guess it; the computer PRINTs the letters that the user has guessed correctly in their right positions in the word:

"HANGMAN" PROGRAM

```
  10 BORDER 0: CLS : PRINT AT 1,
12;"HANGMAN"
  20 PRINT AT 10,2;"Ask a friend
 to type a word"
  30 PRINT AT 12,3;"or phrase fo
r you to guess"
  40 PRINT AT 18,4;"DON'T LOOK A
T THE SCREEN"
  50 PRINT AT 20,0;"Press ENTER
when you're finished"
  60 INPUT a$
  70 CLS : LET l=LEN a$: LET s=0
  80 FOR n=1 TO l
  90 IF a$(n)=" " THEN PRINT AT
11,(32-l)/2+n;"■": NEXT n
 100 PRINT AT 11,(32-l)/2+n;"-":
 NEXT n
 110 PRINT AT 2,12;"HANGMAN";AT
5,6;"-:letter    ■:space"
 120 PRINT AT 16,6;"      Try a le
tter      ";AT 19,0;"Press 1 to gu
ess the whole thing"

scroll?
```

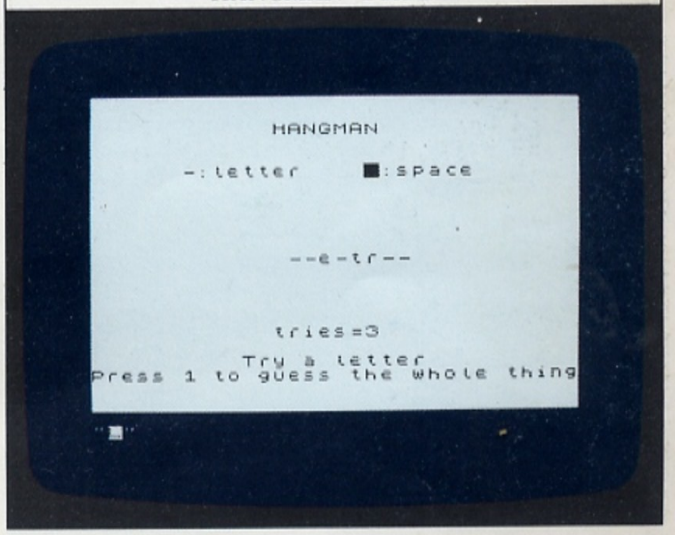
```
 130 INPUT t$
 140 IF t$="2" THEN STOP
 150 IF t$="1" THEN GO TO 210
 160 LET s=s+1: PRINT AT 16,12;"
tries=";s
 170 FOR n=1 TO l
 180 IF t$=a$(n) THEN PRINT AT 1
1,(32-l)/2+n;t$
 190 NEXT n
 200 GO TO 130
 210 PRINT AT 18,6;"Try the whol
e thing": INPUT t$
 220 IF t$=a$ THEN CLS : PRINT A
T 11,12;"CORRECT";AT 13,12;"scor
e=";s+1: STOP
 230 GO TO 120



0 OK, 0:1
```

Lines 10 to 50 PRINT the title frame. When a friend has typed in the test string that you will have to guess, line 70 calculates the length of this test string using the command LEN, and sets the score, s, to zero.

The program now has to PRINT symbols on the screen to represent the letters in the test string. As you guess the letters, any correctly guessed letters will replace these symbols. Also, to allow for test phrases rather than just words, the positions of the spaces between the words are shown. Line 100 PRINTs hyphens to represent the characters. Line 90 PRINTs black squares to represent any spaces in the test string. In lines 90 and 100, (32−l)/2+n works out where the characters that represent the test string should be PRINTed so that they lie in the middle of the screen, not off to either side (a similar effect is incorporated in commercial word-processing programs).

If you want to guess the whole word or phrase instead of keying in individual letters (you can do this at any point in the game), press 1. The program jumps to line 210. The word or phrase that you type in (t$) is compared to the stored string (a$). Then a "CORRECT" frame is PRINTed or if the guess is wrong, the program returns to single letter entry. When a single letter is tried, lines 170 to 190 compare it to each character of the stored string in turn. If the guess is correct, the letter is PRINTed in the appropriate position in the display. Here is an example of the display you should see when a game is in progress:

"HANGMAN" DISPLAY



You can easily limit the number of guesses by adding the commands:

```
IF s>n THEN STOP
```

after the statements where the score, s, is calculated. If you should make a mistake when using the program, it can be difficult to interrupt using the shifted BREAK key, so to make this easier, add one extra line:

```
145 IF t$="3" THEN STOP
```

To stop the program at any point, simply press 3.

FACT-FINDING

You can use your Spectrum to store information, rather like an electronic filing cabinet. However, it would be time-consuming if you had to display the complete contents of a long file every time you wanted to look up a single item. You would still have to scan the screen visually to find the piece of information you were looking for, as if you were using paper files. A good program lets you pick out information selectively, so that the computer, and not you, does the searching.

How to program a serial search

The program that follows uses a simple but effective method to locate one item in a long list of DATA. It's called a "serial search" because it searches the DATA in a series of stages. It takes the test string (T$) that you type in and compares it to each string in the program's DATA statements in turn. It carries on until the test string matches one of the stored strings:

SERIAL SEARCH PROGRAM

```
  10 DATA "Anne","93-446 2039","
Dentist","920 4263","Doctor","29
2 3042","Elizabeth","021-111 488
4","Fred","032-244 9220"
  20 DATA "Jim","021-437 2466","
John","01-444 9000","Mac","01-60
0 0024","Pat","429866","Police",
"866529","Richard","01-626 1900"
  30 DATA "Ross","816594","Schoo
l","986881","Sheila","01-626 824
4","Taxi","629 4306","Vet","7044
6","Wendy","260 8377"
  40 BORDER 0: PAPER 7: INK 0: C
LS
  50 PRINT AT 5,9;"Serial search
"
  60 PRINT AT 10,6;"Enter name:-
----------"
  70 PRINT AT 20,2;"Press ENTER
to start search"
  80 INPUT T$: PRINT AT 10,17;T$
  90 LET t=1

scroll?
```

```
  100 FOR n=1 TO 17
  110 READ N$,P$
  120 IF N$=T$ THEN GO SUB 190: S
TOP
  130 LET t=t+1
  140 NEXT n
  150 PRINT AT 20,2;"------Search
  complete-------"
  160 BEEP 0.2,10
  170 PRINT AT 10,6;"---Name not
found----"
  180 STOP
  190 PRINT AT 20,2;"------Search
  complete------"
  200 BEEP 0.2,25
  210 PRINT AT 12,13;P$
  220 RETURN



0 OK, 0:1
```

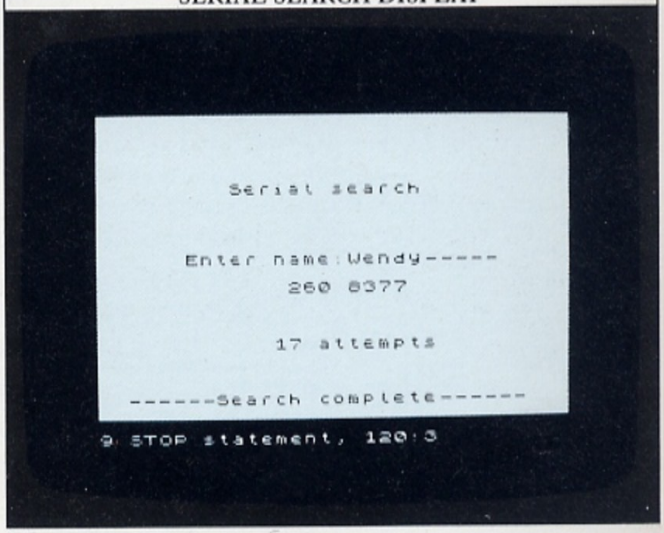
Lines 10 to 30 hold a list of names and telephone numbers. Lines 100 to 140 repeatedly READ a name (N$) and number (P$) from the DATA statements (note that the numbers are treated as strings). This carries on until a stored name is found to match the test string. Line 210 then PRINTs the phone number.

If the computer cannot match the test string with any string in the DATA lines, line 170 announces that the name has not been found. If you now add the following line you can count the number of searches made before a successful match is found:

```
215 PRINT AT 16,12;t;" attempts"
```

Now try RUNning the program to find the last telephone number on the list:

SERIAL SEARCH DISPLAY



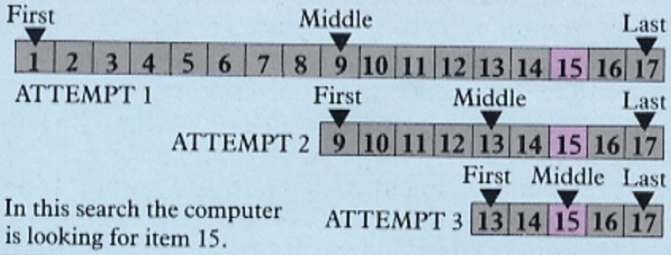
You will find that the program takes 17 searches and a fraction of a second to find the last name in the list. That's acceptable in a short program like this with just a few pieces of stored DATA. But for a serial search program with more than about 200 names or other items of DATA, the delay caused by looking at every single piece of DATA to find the correct one becomes quite noticeable. The program is simple but slow.

Speeding up a DATA search

Fortunately, another DATA searching method can speed things up considerably. It relies on the storage of DATA in strictly alphabetical or numerical order. The DATA stored in the program is repeatedly divided in two and the first item found there is READ and compared with the test string. This produces three possibilities. First, the two strings may be identical – a successful match at the first attempt. Second, the test string may come later in the alphabet or numerical order than the item found. Third, the test string may

come before the item found. If the match is not successful, the half of the DATA in which the test string lies is further divided in two and searched in the same way until the strings are matched.

STAGES IN A BISECTION SEARCH



In this search the computer is looking for item 15.

Here is a program which carries out this "bisection search" of a DATA bank:

BISECTION SEARCH PROGRAM

```
  10 DATA "Anne","Dentist","Doct
or","Elizabeth","Fred","Jim","Jo
hn","Mac","Mark","Pat","Police",
"Richard","Ross","Sheila","Taxi"
,"Vet","Wendy","ABCDE"
  20 DATA "93-445 2039","920 426
3","292 3042","021-111 4884","03
2-244 9220","021-437 2456","01-4
44 9000","01-600 0024","946882",
"429866","866529","01-628 1900",
"816594","01-626 8244","629 4306
","70446","260 8377"
  30 BORDER 0: PAPER 7: INK 0: C
LS
  40 DIM N$(18,9): DIM P$(17,11)
  50 FOR n=1 TO 18: READ N$(n):
NEXT n
  60 FOR n=1 TO 17: READ P$(n):
NEXT n
  70 LET f=1: LET l=18
  80 PRINT AT 5,8;"Bisection sea
rch"

scroll?
```

```
  90 PRINT AT 10,6;"Enter name:-
---------"
 100 PRINT AT 20,2;"Press ENTER
to start search"
 110 INPUT T$: PRINT AT 10,17;T$
 120 LET t=1
 130 LET X=INT ((f+l)/2)
 140 IF N$(X)=T$ THEN GO TO 190
 150 IF N$(X)<T$ THEN LET f=X
 160 IF N$(X)>T$ THEN LET l=X
 170 IF X=INT ((f+l)/2) THEN GO
TO 220
 180 LET t=t+1: GO TO 130
 190 PRINT AT 20,2;"------Search
complete------": BEEP 0.2,20
 200 PRINT AT 12,13;P$(X)
 210 STOP
 220 PRINT AT 20,2;"------Search
complete------": BEEP 0.2,10
 230 PRINT AT 10,6;"---Name not
found----"
 240 STOP

> OK, 0:1
```

The DATA has been reorganized into two separate listings – one of names and a second of telephone numbers. In order to be able to locate one item in these DATA statements without having to READ through every item, the DATA must be numbered or tagged in some way. Line 40 does this by creating two numbered lists of DATA, or arrays. Lines 50 and 60 READ the relevant pieces of DATA into the two arrays. Line 130 sets the conditions for the first search. The first item in the range to be halved, f, is one and the last item, 1, is 18. If a match is not made at line 140, lines 150 and 160 set the new values of f and 1 for the next search.

If you RUN the program in the form shown here, it will answer "Name not found" to every test string. That's because line 40 tells the computer that every name string (N$) is 9 characters long and every number string (P$) is 11 characters long. A name like "Wendy" is therefore stored as "Wendy " which, as far as the computer is concerned, is not equal to the test string. To deal with this add:

```
111 LET T=LEN T$
112 IF T<>9 THEN LET T$=T$+" ":GOTO 111
205 PRINT AT 16,12;t;" attempts"
```

This keeps adding spaces to the test string until it is 9 characters long. Now the program RUNs and PRINTs the number of attempts to find a match:

BISECTION SEARCH DISPLAY

```
Bisection search

Enter name:Wendy-----
260 8377

5 attempts

------Search complete------
9 STOP statement, 210:1
```

The last item in line 10 is not a misprint. Because of the way the program divides the DATA into halves, the last name (WENDY in this case) would actually never be located. The maximum value of x is one less than the number of DATA items. To get around this, the dummy item ABCDE is added, so that all of the DATA can be searched successfully.

Using this method, the last name can be found after only 5 attempts. To save time, the phone number, P$, is only READ when a successful match for the name is found. This time saving makes the bisection technique much more suitable if you want to search long lists of DATA. You could also use programs like those on pages 24–25 to order your DATA before you use it in the bisection search program. By combining the programs you would have an accessible DATA bank.

PIE CHARTS

Computer graphics are invaluable for displaying information in a way that can be understood at a glance, and one of the most easily understood displays that computers can produce is a pie chart. Pie charts are particularly useful for showing the relationship between a quantity and the amount of which it is a part.

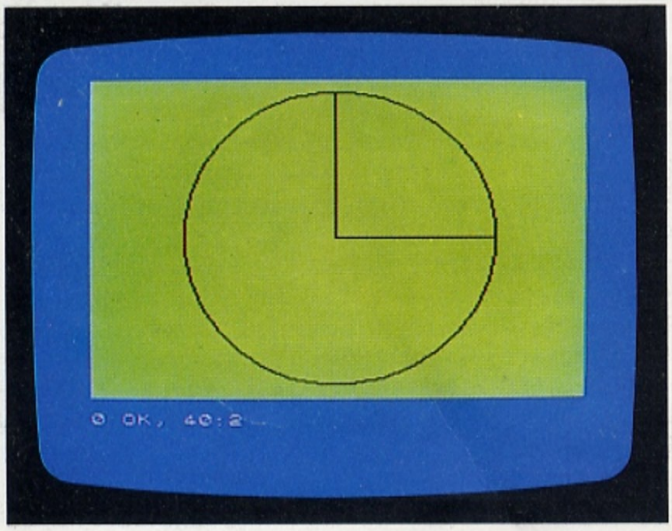
Drawing a fixed pie chart

To produce a pie chart, the first job is to draw a circle, then you can start to put in the edges that mark off the "slices". Each edge is a radius of the circle. Once the first radius is DRAWn, all the other radii can be DRAWn relative to the first one. It's like cutting a pie; it doesn't matter where the first cut is made, but after that, the size of each slice is determined by the angle the slice makes with the previous one.

In the following program, one right-angled slice is DRAWn in a circle:

SINGLE SLICE CHART

```
 10 BORDER 1: PAPER 4: INK 0: C
LS
 20 CIRCLE 128,88,80
 30 PLOT 128,88: DRAW 80,0
 40 PLOT 128,88: DRAW 0,80

0 OK, 0:1
```



The program constructs the circle at the centre of the screen (128,88). Line 30 DRAWs the first radius from the centre to the right. Line 40 then DRAWs the second radius straight up to form the slice.

Adding more slices

You could go on to DRAW a second quarter-circle slice by adding the line:

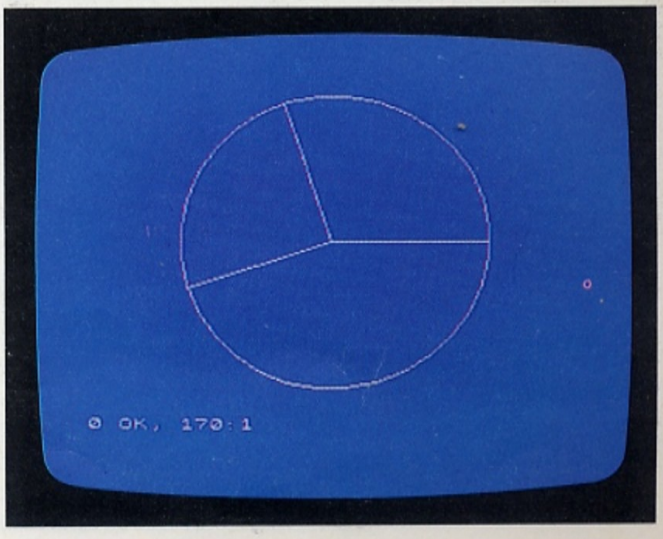
```
50 PLOT 128,88:DRAW -80,0
```

and from there you could go on adding lines to DRAW further radii and build up a pie chart with a number of slices. But this sort of program isn't really very useful because you have to work out the positions of all the edges of the slices beforehand.

However, the principle behind the first program can be used to write another program which will work out the positions of all the radii for you:

VARIABLE CHART

```
 10 DIM s(3): LET s=0
 20 PRINT AT 5,12;"PIE MAKER"
 30 PRINT AT 6,12;"*********"
 40 PRINT AT 8,7;"Total pie siz
e?"
 50 INPUT t: PRINT AT 8,26;t
 60 FOR n=1 TO 3
 70 PRINT AT 3*n+9,7;"What size
 slice ";n;"?"
 80 INPUT s(n): PRINT AT 3*n+9,
26;s(n)
 90 NEXT n
100 BORDER 1: PAPER 1: INK 7: C
LS
110 CIRCLE 128,88,80
120 PLOT 128,88: DRAW 80,0
130 FOR n=1 TO 3
140 LET s=s+s(n)
150 PLOT 128,88
160 DRAW 80*COS (s*2*PI/t),80*S
IN (s*2*PI/t)
170 NEXT n

0 OK, 0:1
```



Lines 20 to 90 set up the title and INPUT frame. Line 10 tells the computer that the program will use a one-dimensional numeric array called s which will have three items in it. Each of these items will be the size of one of three slices that the loop at lines 60 to 90 asks you to type in.

First, though, you must type in the total pie size in response to lines 40 and 50. Lines 100 to 170 DRAW the circle and radii in positions calculated using the slice sizes you typed in. The variable s, which was set to zero in line 10, fixes the position of each slice relative to the first radius. When n=1 for example, s=s+s(1). When n=2, then s=s+s(2). Finally, when n=3, s=s+s(3). For each value of s, a line is DRAWn from the circle's centre to the point calculated using COS and SIN in line 160.

You might wonder how this can DRAW lines which end to the left of, or below, the centre. When COS is used with angles between PI/2 and 3*PI/2 radians (90 and 270 degrees), the number it produces is negative. So if COS(s*2*PI/t) is equal to −1, the radius is DRAWn to an x co-ordinate of 128+(−80), or 48.

Labelling the slices

The previous program works well enough, but there's nothing to identify which slice is which – you need a good memory to know which slice represents which of the quantities you keyed in. So, next you can add a routine to label the slices:

LABELLING ROUTINE

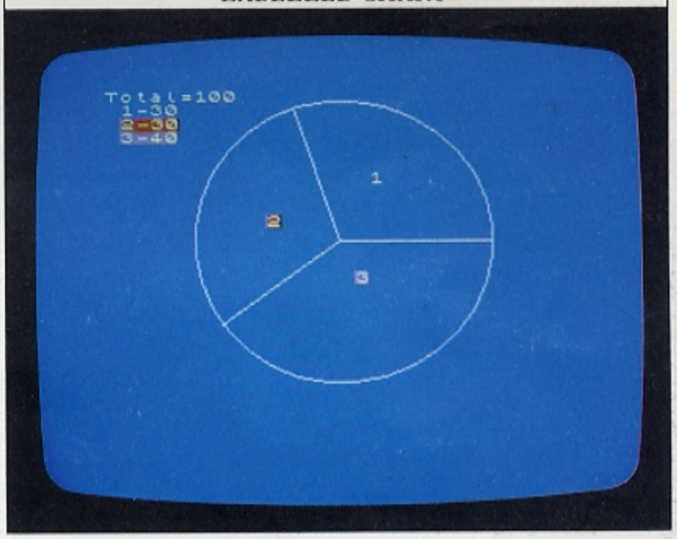
```
115 PRINT AT 0,0;"Total=";t
121 DIM x(4): DIM y(4)
122 LET x(1)=50: LET y(1)=0
170 LET x(n+1)=80*COS (s*2*PI/t
)
180 LET y(n+1)=80*SIN (s*2*PI/t
)
190 LET x=(x(n)+x(n+1))/2
200 LET c=16+INT (x/8+1)
210 LET y=(y(n)+y(n+1))/2
220 LET r=11-INT (y/8+1)
230 PRINT PAPER n; INK 7;AT r,c
;n
240 PRINT PAPER n; INK 7;AT n,1
;n;"-";s(n)
250 NEXT n


0 OK, 0:1
```

Line 121 sets up two arrays, x and y, which are used to store the co-ordinates of the ends of the radii DRAWn on the circle. There are four pairs, not three – those for the first fixed radius plus three for the slices whose sizes you have typed in. The co-ordinates of the end of the first radius are known (50,0), and these are set in line 122.

For each value of n, the co-ordinates of the end of that radius are picked out of the arrays by x(n+1),y(n+1). The program uses these co-ordinates to PRINT a label on the screen. The position for the label is given by two co-ordinates x and y. These co-ordinates are half the distance between x(n) and x(n+1) respectively. The two graphics co-ordinates are then translated into text row and column numbers by lines 200 and 220.

The two co-ordinates x and y are each divided by 8 because there are 8 graphics pixels to each character position. But there is a disadvantage to calculating the label position this way. Since the label is to be PRINTed between the ends of the two radii, it will not work properly if the slice is more than half of the circle. However, for values less than this, the program works correctly and labels the pie chart for you:

LABELLED CHART



For each label, line 240 PRINTs the same number against a corresponding coloured background and PRINTs the size of the slice beside it to give you a display of the figures keyed in.

Try keying in the following answers in response to the question frame:

500 – total income
160 – bills
100 – insurance
180 – travel

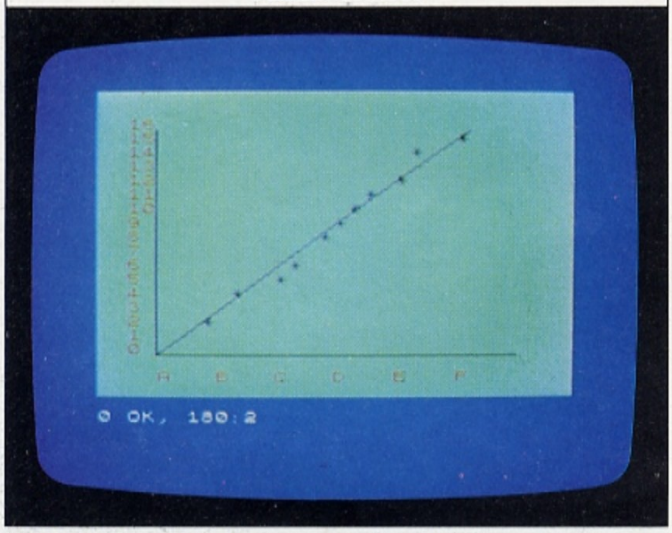
If you try these figures, you will end up with a chart that has a black unlabelled slice. This represents the amount of income that you have left over after the payments you have keyed in.

You can adapt this program to show more than three slices by changing the number of INPUTs and the dimensions of the arrays. It is possible to make the Spectrum DRAW sections of a circle that are filled in with colour. However, if you use routines that DRAW with INK to fill in sections on the pie chart, you will find that you have difficulty getting smooth edges to the sections. This is because the Spectrum's INK works at text rather than graphics resolution.

DRAWING GRAPHS

Because they are so good for games, computer graphics are often only partially exploited for more serious uses. But as you saw on the previous two pages, you can use your Spectrum to produce high-resolution graphics to show any information that you can put into number form. Pie charts are useful for showing how something is split up. Graphs, on the other hand, show how two sets of items are related. Here is a simple graph display:

FIXED GRAPH



You don't need to be a mathematician to get some useful information from this graph. As time goes by (along the horizontal axis), the value measured by the vertical axis is steadily increasing.

How to set up a graph

The program that produces the graph above has to DRAW the horizontal and vertical axes, label them, PLOT the points and finally DRAW a line:

FIXED GRAPH PROGRAM

```
  10 BORDER 1: PAPER 5: CLS
  20 DATA "A","B","C","D","E","F
"
  30 DATA 16,7,14,9,13,12,12,13,
10,15,9,16,8,17,7,18,6,20,4,21,5
,24
  40 PLOT 224,24: DRAW -192,0: D
RAW 0,128
  50 FOR c=4 TO 24 STEP 4
  60 READ m$
  70 PRINT INK 2;AT 20,c;m$
  80 NEXT c
  90 LET l=0
 100 FOR r=18 TO 2 STEP -1
 110 PRINT INK 2;AT r,2;l
 120 LET l=l+1
 130 NEXT r
 140 FOR n=1 TO 11
 150 READ r,c
 160 PRINT AT r,c;"*"
 170 NEXT n
 180 PLOT 32,24: DRAW 168,128

0 OK, 0:1
```

The sets of information are contained in the DATA in line 30. Line 40 DRAWs the two axes of the graph. The two loops that follow, between lines 50 and 80 and between lines 100 and 130, PRINT the labels along each axis. The first loop takes each letter in turn out of the DATA in line 20 and then PRINTs it along the horizontal axis. The position along the axis is determined by c in line 70. This increases in STEPs of 4 (line 50) so that there is a gap between each of the labels. Line 110 PRINTs numbers up the vertical axis. This time, because the labels are a sequence of numbers, there is no need for them to be stored in a DATA line. Instead, they are produced by the STEP in line 100.

Programming graphs to order

The main disadvantage of the previous program is that it will only ever produce the same graph. You can change the information in the DATA lines, but this is a laborious way of altering the display. Ideally, you want a program that will allow you to type any co-ordinates you wish while the program is RUNning. Also, you don't want to have to translate the graph's co-ordinates into Spectrum screen co-ordinates before using them. So, the program must be able to do this conversion for you.

The next program does all this. Although it produces axes that have set labels, you can key in any co-ordinates you like, as long as they fall within the graph's limits:

VARIABLE GRAPH PROGRAM

```
  10 BORDER 2: PAPER 6: INK 1: C
LS
  20 PLOT 224,24: DRAW -192,0: D
RAW 0,128
  30 FOR c=4 TO 28 STEP 4
  40 PRINT AT 19,c;c-4
  50 NEXT c
  60 LET n=0
  70 FOR r=18 TO 2 STEP -2
  80 PRINT AT r,2;n
  90 LET n=n+4
 100 NEXT r
 110 PRINT AT 0,2;"Y"
 120 PRINT AT 20,31;"X"
 130 INPUT "X=";h,"Y=";t
 140 BEEP 0.2,20
 150 PRINT AT 18-16*t/32,4+h;"*"
 160 GO TO 130


0 OK, 0:1
```

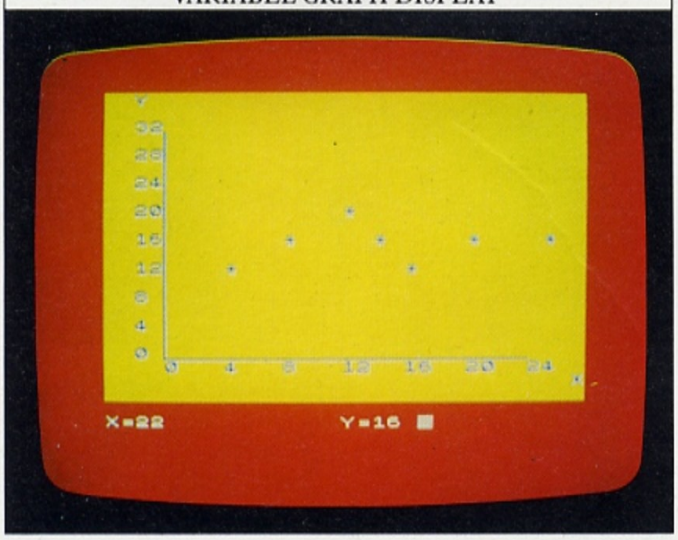
Line 20 DRAWs the axes as before. Lines 30 to 50 PRINT the x axis labels. Here the program produces labels that could be time in hours. The program again uses FOR ... NEXT loops to determine not only what the labels are to be, but also where they are to go. As the column numbers of the horizontal label positions go

from 4 to 28 and the values go from 0 to 24, the program uses the column number to produce the label in line 40.

Lines 60 to 100 PRINT the labels up the vertical axis. Here they are chosen so that they can represent a range of temperatures in Centigrade.

Line 130 invites you to enter a pair of co-ordinates. Type in a time, press ENTER and type in a temperature followed by ENTER again. The computer BEEPs and line 150 PRINTs an asterisk at the screen position:

VARIABLE GRAPH DISPLAY



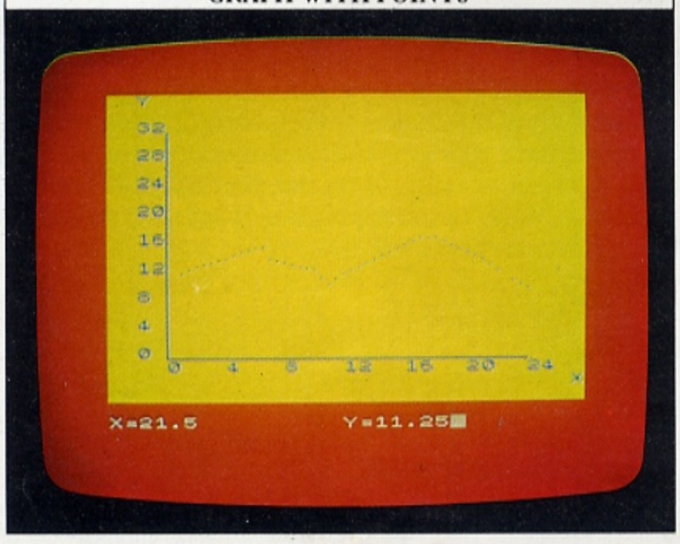
How to alter the display

The graph display is quite coarse, because asterisks can only be PRINTed in the 384 character positions within the graph's area of 24 columns by 16 rows. For a more detailed graph, you could replace line 150 by:

```
150 PLOT 32+192*h/24,24+128*t/32
```

This will produce a display with points instead of asterisks, each positioned on a graphics grid that allows higher resolution:

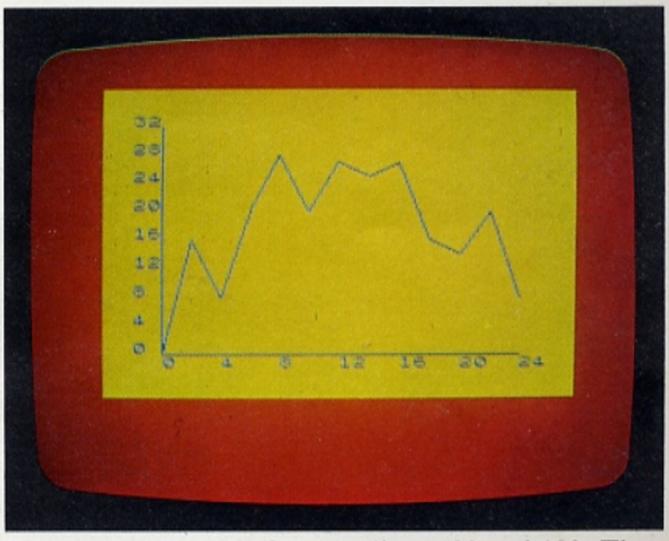
GRAPH WITH POINTS



Altering the program so that it produces connected lines, rather than isolated points, is a little more difficult. What you need to do is tell the computer to PLOT a point, DRAW back from there to the previous point and then PLOT the second point again so that it in turn can be DRAWn back to. A program like this is ideal for writing with functions. Without them, the program lines would have to include repetition of a pair of quite cumbersome calculations for establishing co-ordinates. Here are the line changes and additions that are needed to make the program produce the line graph, together with a display:

LINE GRAPH CHANGES

```
 110 LET a=32: LET b=24: PLOT a,
b
 120 INPUT "X=";c,"Y=";d
 130 PLOT FN X(c),FN Y(d)
 140 DRAW -(FN X(c)-a),-(FN Y(d)
-b)
 150 PLOT FN X(c),FN Y(d)
 160 LET a=FN X(c): LET b=FN Y(d
)
 170 GO TO 120
 180 DEF FN X(c)=32+192*c/24
 190 DEF FN Y(d)=24+128*d/32


0 OK, 0:1
```



Two functions are defined by lines 180 and 190. They convert horizontal and vertical graph co-ordinates into Spectrum screen co-ordinates so that the program can PLOT and DRAW with them. Line 110 sets the first point (a,b) at the bottom left corner of the graph. Then every time you INPUT a pair of co-ordinates, the program converts them, PLOTs a point at x,y, DRAWs back to a,b, PLOTs x,y again and lastly, at line 160, makes a,b equal to x,y.

BAR CHARTS

Graphic information can be presented in a number of different ways. Graphs, as you saw on the previous two pages, are good for showing trends, while bar charts are particularly useful for showing differences in levels.

Bar charts are so named because the information in them is displayed not as single points, but as columns whose height corresponds to the size or level of the item shown. You will frequently see bar charts used on television to show changes in currency exchange values, numbers of votes in elections and so on.

Writing a bar chart program

Because a bar chart is essentially a graph, you can use much the same techniques as used in a conventional graph program to produce one. The main difference is that instead of PLOTting a single point when fed with co-ordinates, the program must construct a column. With the Spectrum, these are most easily constructed from a series of square graphics characters (using the graphics cursor plus shifted key 8). INK can then be used to add colour to the chart. The next program incorporates this technique:

SIMPLE BAR CHART PROGRAM

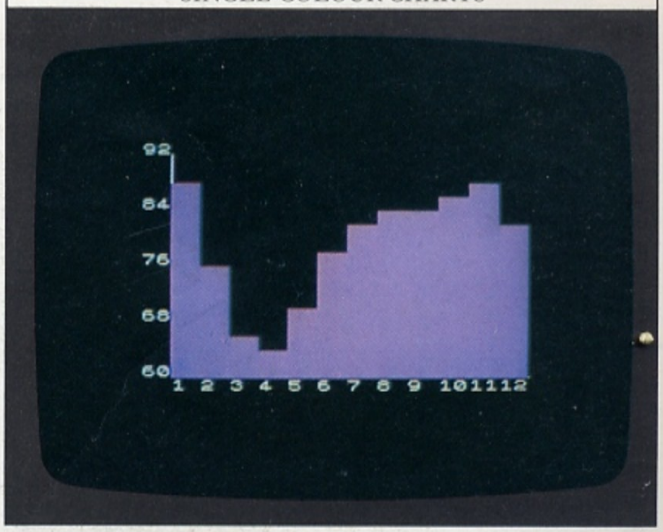
```
 10 BORDER 0: PAPER 0: INK 7: C
LS
 20 PLOT 224,24: DRAW -192,0: D
RAW 0,128
 30 FOR c=4 TO 26 STEP 2
 40 PRINT AT 19,c;(c-2)/2
 50 NEXT c
 60 LET n=60
 70 FOR r=18 TO 2 STEP -4
 80 PRINT AT r,2;n
 90 LET n=n+8
100 NEXT r
110 FOR m=1 TO 12
120 INPUT (m);"=? ";t
130 BEEP 0.2,10
140 FOR r=18 TO 18-(t-60)/2 STE
P -1
150 PRINT INK 3;AT r,2*m+2;"■"
160 NEXT r: NEXT m


0 OK, 0:1
```

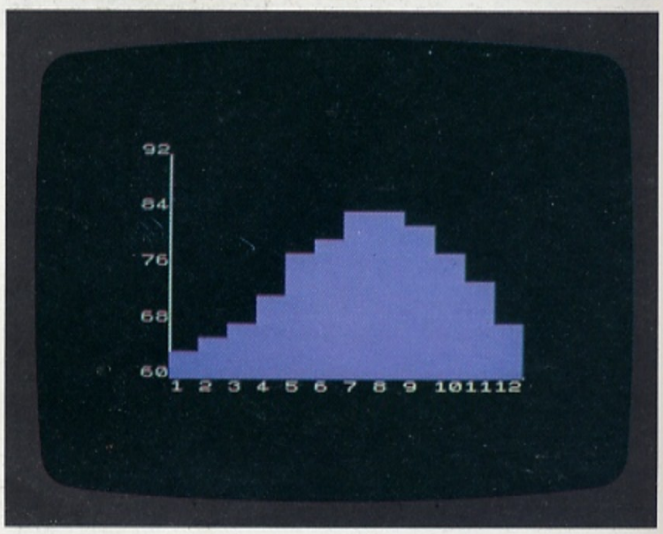
The axes are DRAWn by line 20 in the same positions as those for the graphs on the previous pages. The FOR ... NEXT loop at lines 30 to 50 labels the x axis with the numbers 1 to 12, which could represent the months of the year. If the columns that the labels are to be PRINTed under go from 4 to 26 (STEP 2) then the month number is given by:

```
month=(column−2)/2
```

Try it – column 4, at the beginning of the axis, is equivalent to month 1. Column 26, at the end of the axis, is equivalent to month 12. There are 12 STEPs altogether. Here is the program in action:

SINGLE-COLOUR CHARTS





Combining charts

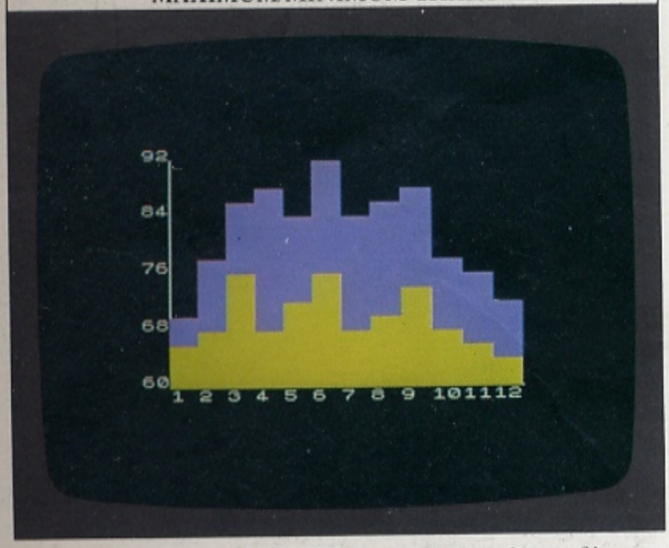
The bar charts so far have shown just one list of items. But it is possible to reorganize the first program so that it shows more than one set of information. You may for instance want to show both maximum and minimum figures like temperatures on the same chart. You don't have to rewrite the first program from scratch. A few additions will do the job:

```
105 FOR n=1 TO 2
150 IF n=1 THEN PRINT INK 3; AT r,2*m+2;
    "■"
155 IF n=2 THEN PRINT INK 6; AT r,2*m+2;
    "■"
170 NEXT n
```

This RUNs as before until you have finished keying in the first set of data. It then sets n to 2 and PRINTs

columns of yellow squares instead of magenta ones. The second set of data must be composed of figures smaller than the first set, otherwise the magenta chart will be overwritten by the yellow squares:

MAXIMUM/MINIMUM CHART



As soon as you start ENTERing the second set of items, you'll notice that the x axis and part of the y axis of the chart disappear. You can get around this very easily by reDRAWing the axes each time a column is DRAWn:

```
156 INK 7:PLOT 224,24:DRAW -192,0:DRAW
    0,128
```

You will need to PRINT the columns as PAPER. This slows down the program considerably, so you may prefer to reDRAW the axes only once at the end, ENTERing the above line as line number 180.

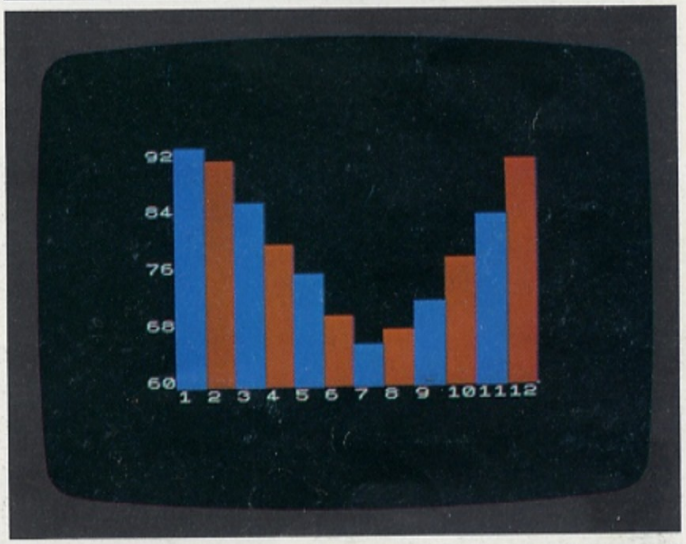
Splitting bars by changing INK

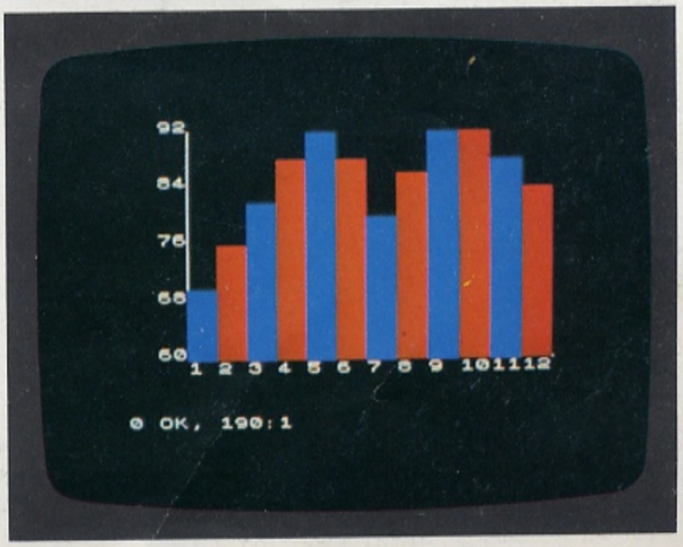
One of the problems with the previous displays is that you cannot distinguish each bar on the charts, making it difficult to connect the bars with the scale on the x axis. You can get around this by using two different INK colours again, but this time alternating them as the bars for one set of INPUTs are PRINTed. It is then quite easy to see which bar relates to which figure on the x axis.

The following program is an adaptation of the first one. If you take out the lines used for the double display, you can then edit the first program to produce the following one. Instead of having the INK colour fixed, it is now controlled by a variable a. A loop is used in conjunction with IF ... THEN to set the INK colour to either blue or red. When a is 1, the INK colour for that bar is blue, then for the next INPUT, when a becomes 2, the INK colour changes to red. You can use this type of colour-changing loop with as many of the INK colours as you like. If you want to increase the number of bars that can be made to appear on a chart, you can reduce the width of each bar by using a single graphics square only. The program would also have to be altered to change the PRINTing positions. Here is the two-colour chart program and some sample displays that it can produce:

TWO-COLOUR CHART PROGRAM

```
  10 BORDER 0: PAPER 0: INK 7: C
LS
  20 PLOT 224,24: DRAW -192,0: D
RAW 0,128
  30 FOR c=4 TO 26 STEP 2
  40 PRINT AT 19,c;(c-2)/2
  50 NEXT c
  60 LET n=60
  70 FOR r=18 TO 2 STEP -4
  80 PRINT AT r,2;n
  90 LET n=n+8
 100 NEXT r
 110 LET m=1
 120 FOR a=1 TO 2
 130 INPUT (m);"=? ";t
 140 BEEP 0.2,10
 150 FOR r=18 TO 18-(t-60)/2 STE
P -1
 160 PRINT INK a;AT r,2*m+2;"■"
 170 NEXT r
 180 LET m=m+1: NEXT a
 190 IF m<12 THEN GO TO 120

0 OK, 0:1
```





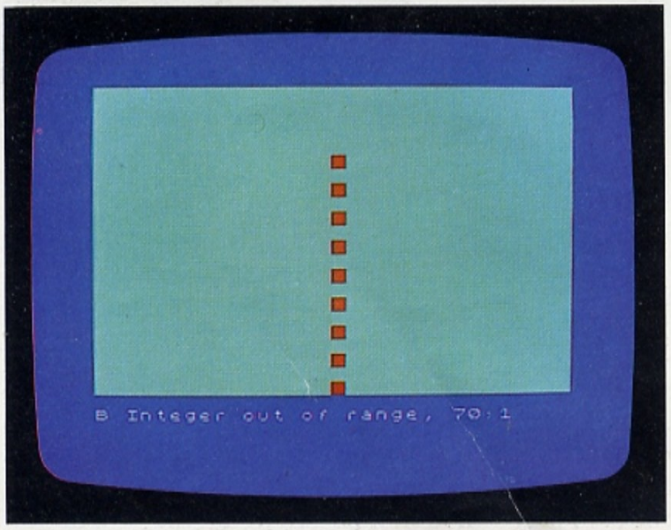
GRAPHICS WITH GRAVITY

On pages 18–19 you saw how SIN and COS could be used to make "natural" graphics, shapes that are sometimes seen in the natural world. To make these shapes you can just experiment with the graphics commands and see what happens. But if you want the computer to simulate something moving in a realistic way, an understanding of how it moves in real life will help you a great deal when you are trying to simulate that object's movements on the computer screen.

Let's take a simple simulation using a bouncing ball and go through the steps necessary to build up different types of program. On pages 8–9 you saw how IF ... THEN could be used to "bounce" a ball in straight lines moving at constant speed. However a ball doesn't move in straight lines. On the screen below is a short program to demonstrate how you could begin simulating a more realistic fall (the display beneath it includes after-images normally deleted):

SIMPLE FALL PROGRAM

```
LS
  10 BORDER 1: PAPER 5: INK 2: C
  20 LET r=5: LET c=16: LET v=1
  30 PRINT AT r,c;"■"
  40 PAUSE 100
  50 PRINT AT r,c," "
  60 LET r=r+v
  70 PRINT AT r,c;"■"
  80 PAUSE 5
  90 GO TO 50

0 OK, 0:1
```



Falling objects are influenced by several forces – gravity, air resistance, surface friction and something called the "coefficient of restitution", which make them move in a complex way. However, you don't have to be a physicist to write a more realistic "bouncing" program. If you drop a ball, it falls to the ground and bounces up again, and that's all you need to know to simulate bouncing on the screen.

In the previous program listing, line 30 PRINTs the "ball" near the top of the screen. After a 2-second PAUSE, the "ball" starts to move downwards. Line 50 erases it. Next, the row number is increased by 1 and last, the "ball" is PRINTed again.

Movement in two directions

If you RUN this program, you will find that although the "ball" is indeed falling to the bottom of the screen, its movement doesn't look very realistic. The program also ends with an error message when the "ball" falls out of the bottom of the screen. The next program improves the display considerably by making the "ball" move sideways as well:

SIDEWAYS FALL PROGRAM

```
LS
  10 BORDER 1: PAPER 5: INK 2: C
  20 LET r=5: LET c=16: LET h=1:
LET v=1
  30 PRINT AT r,c,"■"
  40 PAUSE 100
  50 PRINT AT r,c," "
  60 LET r=r+v: LET c=c+h
  70 PRINT AT r,c;"■"
  80 PAUSE 5
  90 GO TO 50

0 OK, 0:1
```

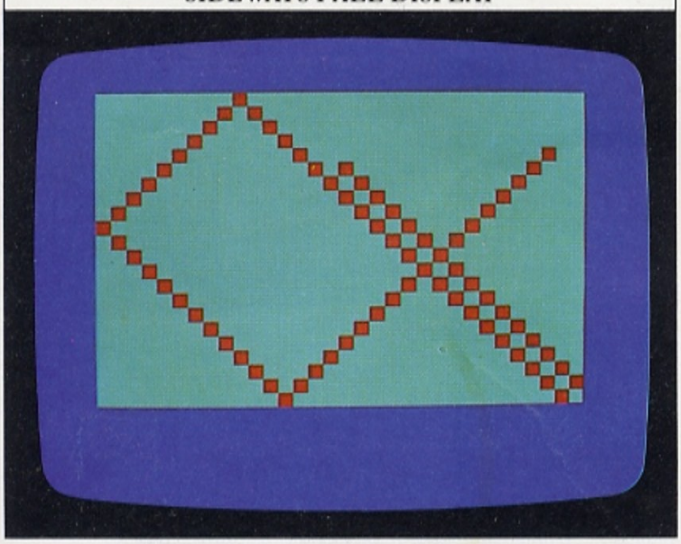
The variable h represents the change in horizontal position and v the change in vertical position. On each loop, v is added to the row number and h to the column number. Now it's easy to modify the motion in any direction. For instance, you can make the "ball" bounce by adding:

```
65 IF c=0 OR c=31 THEN LET h=-h:BEEP
0.05,20
66 IF r=0 OR r=21 THEN LET v=-v:BEEP
0.05,20
```

If you take out the lines which erase the "ball" as it moves, you will see a display like this. Note that the

BEEP is only sounded when the "ball" encounters the edge of the screen:

SIDEWAYS FALL DISPLAY



Simulating gravity

Although the "ball" now bounces around the screen, it does not yet look completely realistic. The reason for this is that the "ball" on the screen does not mimic the effects of an object falling under gravity.

You can add a "force" like gravity that acts in any direction, or that even changes direction during the program's RUN. Gravity acts downwards, so, as the "ball" moves from top to bottom of the screen it should accelerate. When it bounces up from the bottom, it should begin to slow down until it falls back down to the bottom again. The next program imitates this effect. Type in the following listing:

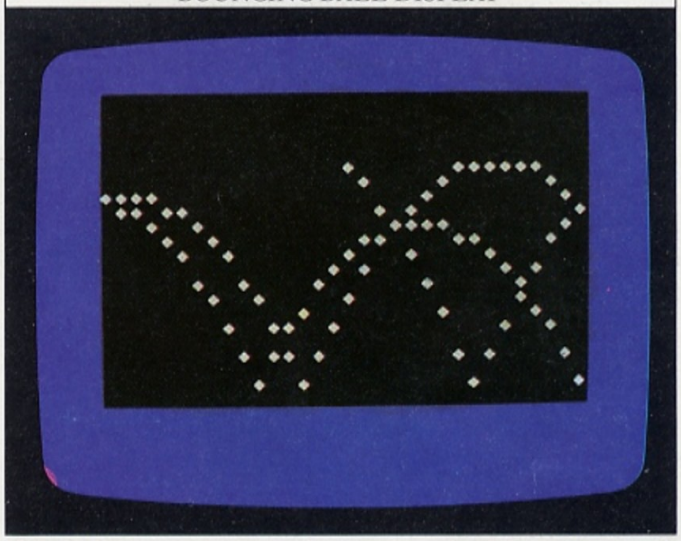
BOUNCING BALL PROGRAM

```
  10 BORDER 1: PAPER 0: INK 7: C
LS
  20 DATA 48,120,252,252,120,48,
0,0
  30 FOR n=0 TO 7
  40 READ x
  50 POKE USR "a"+n,x
  60 NEXT n
  70 LET r=5: LET c=16: LET h=1:
 LET v=1
  80 PRINT AT r,c;"A"
  90 PAUSE 100
 100 REM PRINT AT r,c;" "
 110 LET v=v+0.2
 120 LET r=r+v: LET c=c+h
 130 IF c<1 OR c>30 THEN LET h=-
h: BEEP 0.05,20
 140 IF r<1 OR r>20 THEN LET v=-
v: BEEP 0.05,40
 150 PRINT AT r,c;"A"
 160 PAUSE 5
 170 GO TO 100

0 OK, 0:1
```

In this program, the ball is a user-defined character. The gravity factor is added at line 110. The addition of 0.2 to v means that the change in r – the vertical position – is no longer constant. It increases on each loop, speeding the ball up. When the ball hits the bottom of the screen its direction is reversed (line 140) and therefore v becomes a negative number, repeatedly decreasing the row number. Moreover, the additive gravity factor at line 110 makes v less and less negative, slowing down the upward progress of the ball until its vertical motion ceases, v becomes positive again and the ball begins to move downwards once again.

This display shows how the ball moves with this program:

BOUNCING BALL DISPLAY



The ball bounces around as before, but as it does so, it doesn't reach the same height on each bounce. Its height is gradually decreasing, although its horizontal movement remains the same. The result of this is a rough example of a curve known as a "parabola". Eventually the ball will bounce along the bottom line of the screen, just as a real one would.

In just the same way as the vertical movement can be modified by a "force", you can alter the horizontal movement as well. This gives the impression of an object that is not only falling under gravity, but which is also being blown along by a strong wind.

The curve that the ball makes during this program is not very smooth. This is because the ball is a text character, and its movement is limited to the 32×22 character positions on the screen. If you want to produce smoother bouncing, you can experiment with the PLOT command instead. This will produce a single point at a graphics co-ordinate, allowing much smoother curving over the 256×176 graphics grid. To do this, however, you would have to modify the program so that the character positions in all the lines were converted to graphics co-ordinates. If you refer to the grid on page 59, you should find that this is not too difficult. Because a single point is not very easy to follow, it is simplest to leave all the points PLOTted on the screen to make up a series of gravity curves, tracing the path of the point as it falls to the ground.

WRITING GAMES 1

The next six pages will take you through writing a games program, showing you how to put all the phases together to build up a complete listing. Writing a games program requires some careful planning before you actually start writing lines. To begin with, you need to decide what sort of game you want. Many games combine your acquired skill with an element of chance (the roll of dice, the turn of a card, and so on), and many have a number of different phases of play, each of which confronts you with a different set of problems.

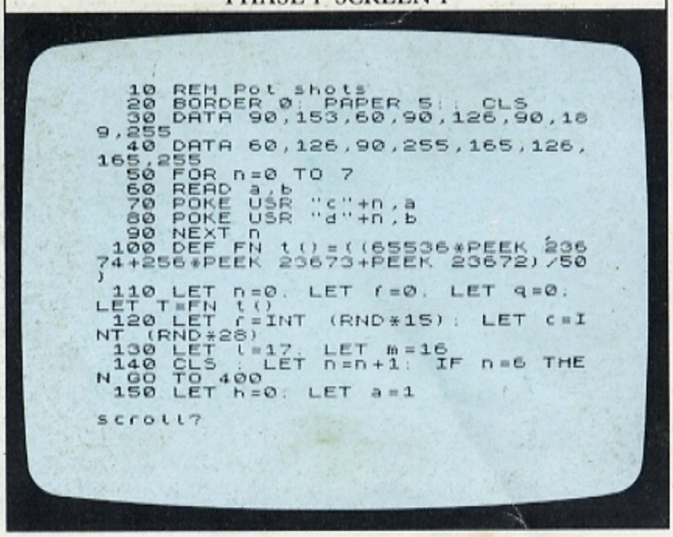
To plan a game, it is best to start by drawing a rough sketch of the screen display, marking the colours and the positions of any fixed characters or patterns. You'll want to refer back to this as you write your program.

Next, you can draw up a flowchart showing the program steps and the order in which they will appear in the program. It isn't necessary to draw a detailed chart – a list of the steps connected with arrows to show their order is sufficient. A complete games program will be more complicated than anything you've written so far, so it is worth designing the program before you key it in. It's easier to rub out a pencil arrow or a couple of lines on your plan than it is to start rearranging lines on the screen if, when you try to RUN it, you find that the program doesn't work.

Keying in phase 1

With the game on this page, the planning stage has been completed, and you can now key in the first part of the two-phase program. The listing that follows is for a practical game – one that anyone should be able to play without any prior knowledge of the program or the computer. Below is the first screen of the program. This phase of the game involves shooting at a moving spacecraft:

PHASE 1 SCREEN 1



```
  10 REM Pot shots
  20 BORDER 0: PAPER 5: CLS
  30 DATA 90,153,60,90,126,90,16
9,255
  40 DATA 60,126,90,255,165,126,
165,255
  50 FOR n=0 TO 7
  60 READ a,b
  70 POKE USR "c"+n,a
  80 POKE USR "d"+n,b
  90 NEXT n
 100 DEF FN t()=((65536*PEEK 236
74+256*PEEK 23673+PEEK 23672)/50
)
 110 LET n=0: LET f=0: LET q=0:
LET T=FN t()
 120 LET r=INT (RND*15): LET c=I
NT (RND*28)
 130 LET l=17: LET m=16
 140 CLS : LET n=n+1: IF n=6 THE
N GO TO 400
 150 LET h=0: LET a=1

scroll?
```

The program gives you a laser base which you can then move left or right. You can fire, but only straight up the screen. A number of spacecraft approach you one by one, and you must destroy them to carry on.

The DATA for the user-defined spacecraft and laser base are READ in by the loop at lines 50 to 90. The time taken to complete the game will be used later to calculate the score, so a time function is defined at line 100, using a technique mentioned on page 21. Some of the variables used in the program are initialized (set to their starting values) by line 110.

The program is impossible to decipher if you don't know what these variables represent. The following table outlines what each of them does.

PHASE 1 VARIABLES

The first phase of the game uses a total of fourteen different variables to control graphics and record strikes.

Variable(s)	Function
r,c	Fix row and column co-ordinates of spacecraft
l,m	Fix row and column co-ordinates of laser base
h	Records successful laser strike
f	Records total number of laser strikes on target
q	Records number of times laser fired
x,y	Fix starting point of laser beam
a	Sets change in position of spacecraft (=1 if left to right, otherwise=−1)
g	Sets column position of mine explosion
T	Records time taken to complete game when subtracted from FN t( )
n,p	General variables

Line 120 sets the random starting point for the spacecraft. Line 130 starts your laser base off in the middle of row 17. In line 140, n records the number of spacecraft attacks. When there have been five attacks, n will equal 6. When it does, the program jumps to line 400 and the scoring calculation.

The second screen of the program contains a number of lines which make decisions and then direct the program to later subroutines. You will notice as you go through the listing that the line numbers sometimes jump by more than 10. This is because it is simpler to give subroutines line numbers that are easily remembered – multiples of one hundred are convenient.

When you key in the second screen's lines, remember that D and C in lines 170 and 180 represent user-defined characters. This means that you will need to switch to the graphics cursor so that they are not PRINTed as letters (after the program has been RUN they can then be LISTed in the form in which they appear in the game). The second screen looks like this:

PHASE 1 SCREEN 2

```
160 PAUSE 100+RND*100
170 PRINT INK 1;AT l,m;" D "
180 PRINT INK 2;AT r,c;" C "
190 BEEP 0.05,25
200 IF INKEY$="Z" THEN LET m=m-
1
210 IF INKEY$="X" THEN LET m=m+
1
220 IF m<0 THEN LET m=0: IF m>3
1 THEN LET m=31
230 IF INKEY$="m" THEN GO SUB 5
00
240 IF h=1 THEN GO TO 120
250 LET c=c+a
260 IF c=29 OR c=0 THEN LET a=-
a
270 LET p=INT (RND*20+1)
280 IF p=3 THEN GO SUB 600
290 GO TO 170
400 LET s=(FN t()-T)/5+q*2-10*f
410 STOP

scroll?
```

After a random PAUSE (line 160), the laser base and spacecraft appear again. Lines 200 and 210 let you move your laser base to either side and line 220 stops it disappearing out of the side of the screen. If you press the M key, the program jumps to the "fire" routine at line 500. If that records a hit, the program jumps back to line 120 and begins a new attack.

Lines 250 and 260 control the movement of the spacecraft. Lines 270 and 280 make the program jump to the mining routine – which is positioned by a variable listed in the table on the opposite page – at line 600 once in every 20 spacecraft moves. This is done by picking a random number from 1 to 20; just one of these numbers will trigger the mining routine. Line 290 continues the same attack by jumping back to line 170.

Lines 400 and 410 calculate the score and end this part of the program. The score is based on the time taken to complete the program, the number of times the laser was fired and the number of direct hits. The score isn't actually used here, but it will appear again later as you develop the game. If you want to check that the scoring lines are working, RUN the program and then key in the direct command PRINT s, without a line number. Your score should then be displayed on the screen.

The subroutine section

Finally, here is the last part of the first program. It contains a pair of subroutines. Lines 500 to 550 DRAW and "unDRAW" the laser beam using DRAW and OVER 1. If m=c then the laser has hit its target. Lines 600 to 690 explode a mine on row 17. The mine's column position is random. If it lands on the laser base, the q value of your score will be affected.

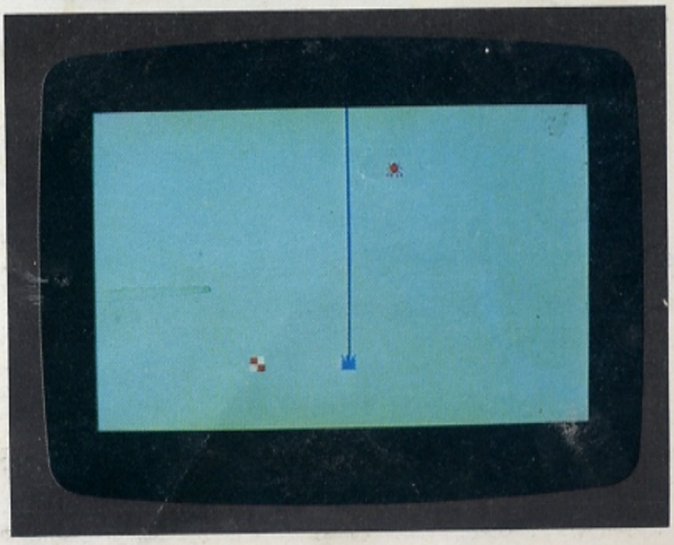
Once you have typed in the listing on the following screen and RUN this phase of the game, SAVE it on a tape so that it is ready to be combined with the next part of the program:

PHASE 1 SCREEN 3

```
500 LET x=m*8+12: LET y=40: LET
q=q+1
510 PLOT x,y: DRAW INK 0; OVER
1;0,135
520 BEEP 0.05,50
530 DRAW INK 0; OVER 1;0,-135
540 IF m=c THEN BEEP 0.2,0: LET
h=1: LET f=f+1
550 RETURN
600 LET g=INT (1+RND*31)
610 FOR p=1 TO 10
620 PRINT INK 2;AT 17,g;"▚"
630 BEEP 0.02,60
640 PRINT INK 7;AT 17,g;"▞"
650 BEEP 0.02,0
660 NEXT p
670 IF g=m THEN LET q=q+10
680 PRINT AT 17,g;" "
690 RETURN


0 OK, 0:1
```

Here is the program in action. In the first display, the laser is firing at the spacecraft, while in the second, a mine has appeared:

PHASE 1 DISPLAYS



WRITING GAMES 2

In the second phase of the program, the scene changes from the air to the sea as a ship tries to depth-charge a moving submarine. Again, the aim is to hit the enemy to produce the best score. The scoring instructions are still not used in this phase, but are ready to be brought into operation when the last phase of the game has been keyed in, linking up the first two parts.

As before the program uses a number of variables to control movement and subroutines. These variables need some explanation if you are to follow what is happening (in your own games you could use REM lines to remind you).

PHASE 2 VARIABLES

The second phase uses six variables to control the three objects animated by the program.

Variable(s)	Function
c	After line 1090, this fixes the column position of the ship
t,d	Fix row and column co-ordinates of submarine
f	Records when depth-charge has been dropped
u,e	Fix row and column co-ordinates of depth-charge

Setting the scene

The first section of the program produces the coloured screen, and selects some random numbers:

DISPLAY/ANIMATION SECTION

```
 100 DEF FN t()=((65536*PEEK 236
74+256*PEEK 23673+PEEK 23672)/50
): LET s=0
1010 REM Sub sinker
1020 LET n=0: LET f=0: LET T=FN
t()
1030 BORDER 2: PAPER 4: INK 1: C
LS
1040 IF n=5 THEN GO TO 1400
1050 FOR r=0 TO 4
1060 FOR c=0 TO 31
1070 PRINT AT r,c;"■"
1080 NEXT c
1090 NEXT r
1100 LET c=14
1110 LET t=10+INT (RND*11)
1120 LET d=INT (RND*11)
1130 IF INKEY$="z" THEN LET c=c-
1
1140 IF INKEY$="x" THEN LET c=c+
1
1150 IF c>25 THEN LET c=25

scroll?
```

For the moment, ignore line 100 – you will find out why it is included on the next page. Lines 1050 to 1090 PRINT a blue sky over the green sea, simply by laying down rows of blue INK squares on top of the green PAPER background. Line 1100 sets the column position for the ship and lines 1130 to 1160 control its movement across the screen.

The aim of this game is to hit the submarine. The M key controls the release of the ship's depth-charges. The ship is also manoeuverable. If you press the Z key the ship will move to the left, while pressing the X key will make it move to the right. All these functions are controlled by INKEY$ for a rapid response. The ship always starts off in the middle of the screen. The position of the enemy is less predictable. Lines 1110 and 1120 set the random starting point of the submarine. It may appear at almost any depth in the water and at any point across the screen.

Main program and subroutines

The second part of the program contains some of the main program, together with a number of subroutines which the main program calls. The subroutines control movement on the screen and detect whether or not your depth-charges have hit the target:

SUBROUTINE SECTION

```
1160 IF c<0 THEN LET c=0
1170 PRINT PAPER 1; INK 7;AT 3,c
;" ▙ "
1180 PRINT PAPER 1; INK 7;AT 4,c
;"▀▀▀▀▀ "
1190 IF f=1 THEN GO SUB 1310
1200 IF INKEY$="m" AND f=0 THEN
LET f=1: LET s=s+100: GO SUB 130
0
1210 LET d=d+1
1220 IF d>24 THEN PRINT AT t-1,d
+1;" ";AT t,d;" ": LET d=0
1230 PRINT AT t-1,d;" ▟▀▙ "
1240 PRINT AT t,d;" ▀▀▀ "
1250 GO TO 1130
1300 LET u=5: LET e=c+2
1310 PRINT AT u,e;" "
1320 LET u=u+1
1330 IF u=21 THEN LET f=0: RETUR
N
1340 PRINT AT u,e;"▄"
```

```
1350 IF u=t AND e>d AND e<d+6 TH
EN BEEP 0.2,0: LET n=n+1: LET f=
0: GO TO 1030
1360 RETURN
1400 LET s=s+FN t()-T
```

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

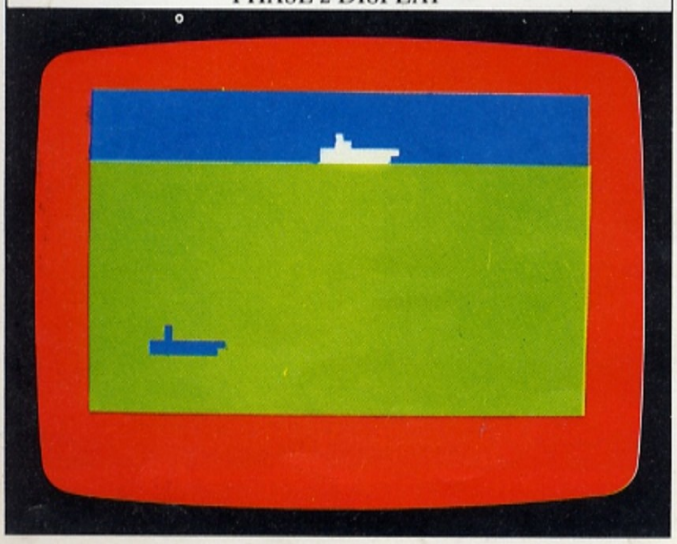
Line 1200 makes the program jump to the depth-charge routine at line 1300 if you have pressed M. The score, s, is also adjusted every time a depth-charge is dropped. The score is related to the time that has passed by using FN t() in the last line of the program. Lines 1210 to 1240 control the movement and appearance of the submarine. Line 1250 continues the program by returning it to line 1130 to check the keyboard for key-presses.

Lines 1300 to 1360 make a depth-charge travel down the screen. The charge is composed of the graphics character on key 3. If the charge reaches the bottom of the screen, line 1330 returns f to its original value (zero) and RETURNs to the main program. However, if the position of the depth-charge coincides with any position occupied by the submarine (line 1350), then that attack is terminated and a new one started. You will notice when you RUN the program that only one depth-charge can be released at a time. If f has been set to 1 by line 1200, when a depth-charge is dropped, line 1190 stops you from dropping another charge until f is once again equal to zero. This will be true either when the charge reaches the bottom of the screen (line 1330) or when it hits the submarine (line 1350).

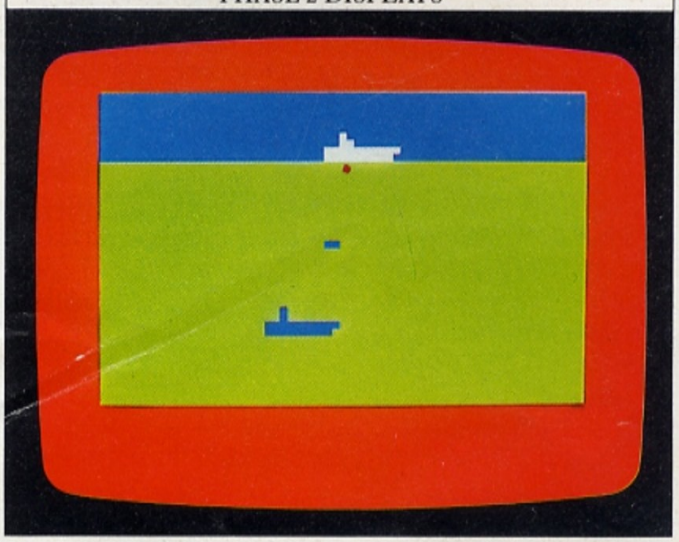
The program uses keyboard graphics to make up all the objects in this display, and as you can see from the following screens, the results show the block outlines of these characters. However, it is easy to improve the game by using user-defined characters instead. To make a ship or submarine, just PRINT a row of characters together. The greater resolution that you can then achieve will considerably improve the display, although more programming lines will be needed.

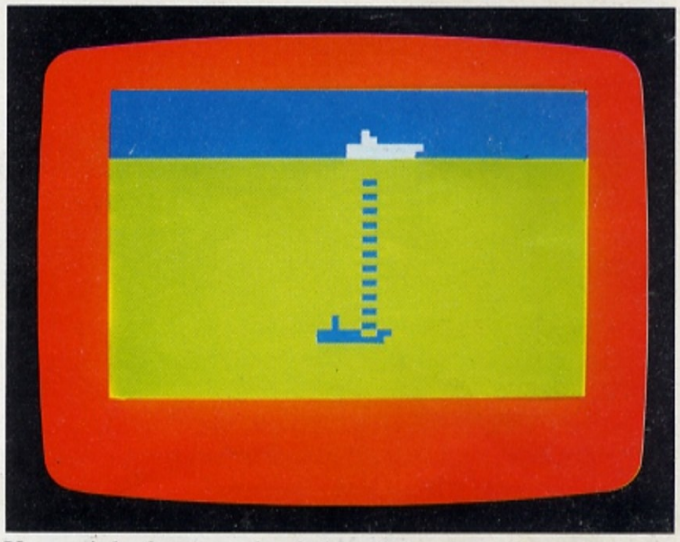
Here are some displays of the game in action. In the third screen, the path of a single depth-charge is shown by putting a REM command in line 1310, disabling the PRINT command which normally erases the charge every time it moves down one position:

PHASE 2 DISPLAY



PHASE 2 DISPLAYS





You might have noticed that there doesn't seem to be any means of erasing the old unwanted images of the ship and submarine before the new images are PRINTed. As both only move one character position to either side of their current position, they can be effectively and simply erased by including a blank square (a space) to either side of the graphics.

The scoring routine

The time is once again used to calculate the score at the end of the game. In the final version of this program, the time function will be defined in the first phase of the game, so a second definition is not necessary in this part. However, if you want to check that the program is working properly, you need to have the time function so that you can PRINT s. Line 100 is put in here so the scoring routine can be tested. The timing function will be taken out again when the final version of the game is written.

When you have checked that the program RUNs, SAVE it on cassette ready to be combined with phase 1.

WRITING GAMES 3

Now that you have keyed in and SAVEd the first two phases of the game, you are ready to add the game instructions and complete the part of the program which will produce the score. The first problem to overcome is that the two programs are SAVEd as two separate files on your cassette. To RUN as one program, they need to be combined. You cannot simply LOAD one program from cassette onto another which is already in the computer's memory. If you do this, you will find that the program in memory will simply disappear, just as if you had pressed NEW.

The Spectrum has a command to deal with this problem – MERGE. It adds together the contents of two files. To see how MERGE is used, imagine that you have SAVEd both programs from pages 38–41 on tape, but that the "sub sinker" program is already in the computer's memory. If you then type:

```
MERGE " "
```

– putting the first filename inside the quotation marks – and then press ENTER and play the tape, the first program will be LOADed into the computer. This time the "sub sinker" program will not be erased, as happens with LOAD. However, MERGE only combines programs properly if their line numbers do not overlap. The extra line (100) added to the "sub sinker" program so that you could test the scoring routine will be over-written by line 100 of the first program. This is why the "sub sinker" is numbered from line 1010 onwards. If it had been numbered from line 10, it would have been over-written by the first phase of the game wherever there were lines of the same number.

The two MERGEd phases are now a single program. As a safety precaution you could now SAVE the whole program on cassette. This is well worth the trouble if you are developing a long program, because accidental deletions can otherwise take a long time to put right.

Adding the game instructions

If you RUN the MERGEd program, you will find that although it is theoretically one program, it still behaves as two separate units. When you are writing games programs in phases like this, you will need to do a little tailoring to the final MERGEd program to make it RUN through properly.

Linking the two phases is easily done. Change line 410 to:

```
410 GOTO 1000
```

That's not a mistake, even though the "sub sinker" program begins at line 1010. It's to allow you some space to add game instructions starting from line 1000.

Now you can go right back to the beginning and start the program off with a title frame containing all the instructions the player will need. The keys that control the movement of objects on the screen need to be listed. You also need to tell the player how to start the game, bearing in mind that by the time the message appears, the program that contains the game will already be RUNning. Here are four lines that give the opening instructions for phase 1:

PHASE 1 INSTRUCTION LINES

```
  1 PRINT AT 2,11;"POT SHOTS";A
T 3,11;"*********";AT 8,5;"Five
alien spacecraft";AT 9,2;"are la
ying mines in your zone";AT 11,2
;"You must destroy the aliens"
  2 PRINT AT 13,2;"************
***************";AT 18,6;"LASER
BASE CONTROLS";AT 19,4;"z:left
x:right  m:fire";AT 21,5;"Press
any key to start"
  3 IF INKEY$<>"" THEN GO TO 3
  4 IF INKEY$="" THEN GO TO 4
```

Lines 1 and 2 PRINT the game title, and explain its controls. Line 3 stops the computer from accepting your RUN and ENTER key-presses as the trigger to start the game. The next time you press a key, the condition for repeating line 4 is broken, and the first phase of the game begins:

PHASE 1 INSTRUCTIONS

```
          POT SHOTS
          *********


     Five alien spacecraft
  are laying mines in your zone

  You must destroy the aliens

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


       LASER BASE CONTROLS
  z:left   x:right   m:fire

  Press any key to start
```

The instructions for the second phase of the game are inserted in a similar way:

PHASE 2 INSTRUCTION LINES

```
1000 PRINT AT 2,11;"SUB SINKER";
AT 3,11;"**********";AT 8,4;"Now
 five submarines have";AT 9,0;"i
nvaded your territorial waters";
AT 11,3;"You can depth charge th
em"
1001 PRINT AT 12,3;"************
**************";AT 18,5;"SURFACE
SHIP CONTROLS";AT 19,0;"z:left
x:right  m:drop charge";AT 21,5;
"Press any key to start"
1002 IF INKEY$<>"" THEN GO TO 10
02
1003 IF INKEY$="" THEN GO TO 100
3

0 OK, 0:1
```

```
SUB SINKER
**********

Now five submarines have
invaded your territorial waters

You can depth charge them
*************************

SURFACE SHIP CONTROLS
z:left  x:right  m:drop charge

Press any key to start
```

You can of course use any keys you want to specify movement as long as you change them throughout the program. Now neither phase of the game starts until the player is ready and presses a key to begin.

Completing the scoring routine

Finally, a scoring routine needs to be added to the end of the MERGEd program. The final score line of "sub sinker" is retained:

```
1400 LET s=s+FNt()-T
```

However, if you have played the two games independently and typed PRINT s afterwards, you will have noticed that the first phase of the game yields a result ranging from −20 or so to several hundred, but the second phase produces results of several hundreds to several thousands. The results of the two games need to be of the same order of magnitude. That can be achieved by multiplying the running score total in line 400 by 100. This makes the score compatible with that from phase 2:

```
400 LET s=100*(FNt()-T)/5+q↑2-(10*f)
```

This line is a good test of your understanding of the variables from pages 38–41! To make the presentation of the score more interesting, you can add a few more lines to turn this purely numerical score into a ranking:

SCORING ROUTINE

```
1410 BORDER 0: PAPER 7: INK 0: C
LS
1420 PRINT AT 6,2;"You have earn
ed the rank of"
1430 IF s<1000 THEN LET a$="COMM
ANDER"
1440 IF s>=1000 AND s<2000 THEN
LET a$="CAPTAIN"
1450 IF s>=2000 AND s<4000 THEN
LET a$="PILOT"
1460 IF s>=4000 AND s<6000 THEN
LET a$="CADET"
1470 IF s>=6000 THEN LET a$="ROO
KIE"
1480 PRINT AT 10,12;a$

0 OK, 0:1
```

```
You have earned the rank of

PILOT

0 OK, 1480:1
```

Lines 1410 to 1480 divide the scores up into bands, each of which is assigned to a rank. A series of IF ... THEN lines decides where your score comes in the ranking. You can change the cut-off scores for each band to make the games harder or easier.

You now have a complete two-phase game with instructions, action and a scoring routine. Although the two phases used on these pages are relatively simple, the way that they are combined can be used to build up games of your own that are much more complex. You can use MERGE to put together a number of sub-programs, each written and tested independently. The only restriction on this is the size of the computer's memory, but unless you are combining very long programs, this shouldn't be a problem.

IMPROVING SOUND

On the Spectrum the command used to produce sound, BEEP, is quite straightforward. A line like:

```
100 BEEP d,p
```

will produce a sound that is d seconds long at a pitch p. But one of the problems of using BEEP in programs is that it stops everything else while it is being carried out. This means that if you want to produce long sounds in conjunction with movement on the screen, you cannot use a single BEEP command, or the program will "freeze" until the BEEP has finished. To link sound and animation as well as possible, BEEP needs to be used briefly but frequently, so that the sounds produced add up to give the effect you want.

How to improve sound with animation

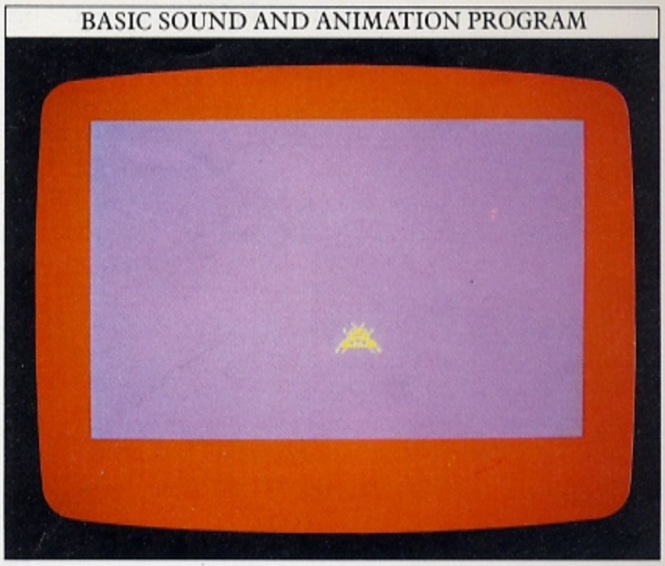
You can hear BEEP working badly with animation if you RUN the following program:

BASIC SOUND AND ANIMATION PROGRAM

```
  10 DATA 0,130,0,3,239,128,0,68
,0,15,69,224
  20 DATA 0,40,0,30,40,240,8,254
;32,63,125,248
  30 DATA 9,57,32,63,255,248,5,2
55,64,68,0,68
  40 DATA 3,131,128,200,0,36,2,2
54,128,144,0,18
  50 FOR n=0 TO 7
  60 READ u,v,w,x,y,z
  70 POKE USR "a"+n,u
  80 POKE USR "b"+n,v
  90 POKE USR "c"+n,w
 100 POKE USR "d"+n,x
 110 POKE USR "e"+n,y
 120 POKE USR "f"+n,z
 130 NEXT n
 140 LET r=10: LET c=14
 150 BORDER 2: PAPER 3: INK 6: C
LS
 160 PRINT AT r,c;"      "
 170 PRINT AT r+1,c;" ABC "

scroll?
```

```
 180 PRINT AT r+2,c;" DEF "
 190 PRINT AT r+3,c;"     "
 200 LET a=INT (RND*2)
 210 IF a=0 THEN LET a=-1
 220 LET b=INT (RND*2)
 230 IF b=0 THEN LET r=r+a
 240 IF b=1 THEN LET c=c+a
 250 IF c<0 OR c>27 OR r<0 OR r>
18 THEN GO TO 150
 260 BEEP 0.2,25
 270 GO TO 160

0 OK, 0:1
```

BASIC SOUND AND ANIMATION PROGRAM



The flying saucer is made up from six user-defined characters arranged in two rows – a,b,c on top and d,e,f underneath, with one space either side. In addition, a row of spaces is PRINTed above and below the saucer, so that, whichever way it moves, the spaces surrounding it erase the previous image. Line 200 produces a 0 or 1 at random. Line 210 turns that into either −1 or +1. This will be used to change the position of the saucer. Line 220 produces another 0 or 1 at random. If it's a 0, the saucer's row is changed by the variable a. If the result of line 220 is 1, the saucer's column is changed by a. This moves the saucer around on the screen in an unpredictable way. Line 250 tests to see if the saucer has reached the edge of the screen and if so brings it back to the centre. The sound is dealt with in line 260 in a single BEEP statement lasting a fifth of a second. It can be improved by adding these lines:

NEW LINES TO SPLIT SOUND

```
195 BEEP 0.005,25
205 BEEP 0.005,30
215 BEEP 0.005,35
225 BEEP 0.005,40
235 BEEP 0.005,45
245 BEEP 0.005,40
260 BEEP 0.005,35

0 OK, 0:1
```

The first version is a very poor example of animation. Effective animation relies on changing from one character display to the next as quickly as possible, but in that program the single BEEP interrupts the movement for too long, so the saucer moves in a slow, halting way and the sound effect really doesn't add much to it.

By adding the lines listed on the previous screen you will split the BEEP up into a number of shorter sounds, thus improving the quality of sound in the program. Each BEEP now lasts for only five hundredths of a second. The total duration of all the BEEP statements (0.035 seconds) is less than a quarter of the length of the single BEEP used previously, so this program RUNs approximately four times as quickly as the first version. Moreover, because the sound effect is split up into a number of separate statements, this gives you a chance to sound them at different pitches to produce a much more interesting warbling effect.

You could also write more BEEPs between the introductory lines (10 to 140) to announce the arrival of the saucer in advance of its appearance, but note that any BEEPs written into lines 50 to 130 will sound eight times because of the n loop connecting them.

Linking BEEP to screen positions

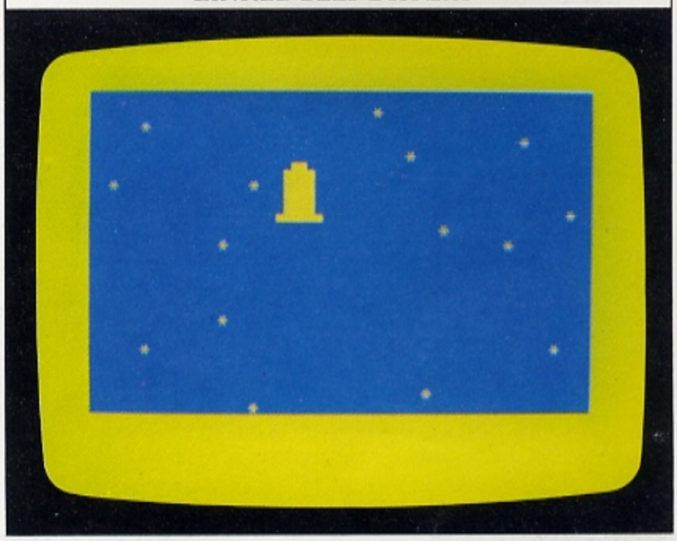
One very effective way of producing sound with animation is to link the BEEP pitch to a variable that controls a character's position on the screen. You will have already encountered this with the "bouncing" program on pages 8–9. Linking BEEP like this is fairly easy, and it cuts out quite a lot of programming lines. All you have to ensure is that the variable controlling the BEEP comes within the right range, and is used to change the duration or pitch of the BEEPs in the right direction. You can experiment with these techniques on the next program. Type in the following listing and RUN the program:

LINKING **BEEP** WITH POSITION

```
 10 BORDER 6: PAPER 1: INK 6: C
LS
 20 DATA 1,18,2,3,3,27,4,20,6,1
,6,10,8,30,9,22,10,8,10,26,15,6,
17,3,17,29,20,21,21,10
 30 FOR n=1 TO 15
 40 READ r,c
 50 PRINT AT r,c;"*"
 60 NEXT n
 70 FOR r=21 TO 4 STEP -1
 80 PRINT AT r-4,12;"  "
 90 PRINT AT r-3,12;" ▄ "
100 PRINT AT r-2,12;"▟▙"
110 PRINT AT r-1,12;"▟▙"
120 PRINT AT r,12;"  "
130 NEXT r


0 OK, 0:1
```

This program PRINTs a star field and then moves a keyboard graphics rocket upwards through it. Lines 10 to 60 PRINT the star field. Lines 70 to 130 PRINT and move the rocket. When you RUN the program, the following display should appear:

LINKED **BEEP** DISPLAY



Now you can add the sound effects to the above program. This time, their aim is to produce an unusual sound of changing pitch. What is needed is a BEEP statement containing a variable whose value changes every time the BEEP is carried out. The obvious thing to relate the pitch to is the changing row number. The problem is that the sound should increase in pitch as the rocket rises, but as it does so, the row number decreases. So if the pitch were simply made a multiple of or a fraction of the row number, it, too, would decrease as the rocket rose up through the star field.

However, there is a way of overcoming this problem. Try typing in this line:

```
125 BEEP 0.05,80-(10*r)/2
```

Now the smaller r is, the higher is the pitch number. If the range of pitch is too large, multiplying r by a smaller number will reduce it:

```
125 BEEP 0.05,80-(5*r)/2
```

If the pitches are too high overall, reducing the starting value (80) will bring all the BEEPs down in pitch:

```
125 BEEP 0.05,60-(5*r)/2
```

To make the sound more interesting, you can split it into two components:

```
 75 BEEP 0.025,60-(5*r)/2
125 BEEP 0.025,63-(5*r)/2
```

The second sound is slightly higher in pitch than the first, giving a progression in pitch between the two. Using a number of even shorter BEEPs in these lines will make the sound effect even more complex.

THE SPECTRUM SCREEN POINTER

In many Spectrum games programs, areas of INK colour on the screen represent objects that you have to avoid. If you hit them, the programs respond with a penalty of some kind. But how do you go about programming the computer to decide if a character you are moving on the screen has hit something? You could use IF ... THEN to check co-ordinates, but there is a way that is much easier and quicker.

The Spectrum BASIC command POINT lets the computer examine any point on the screen and test whether it is a PAPER or an INK colour. In the line:

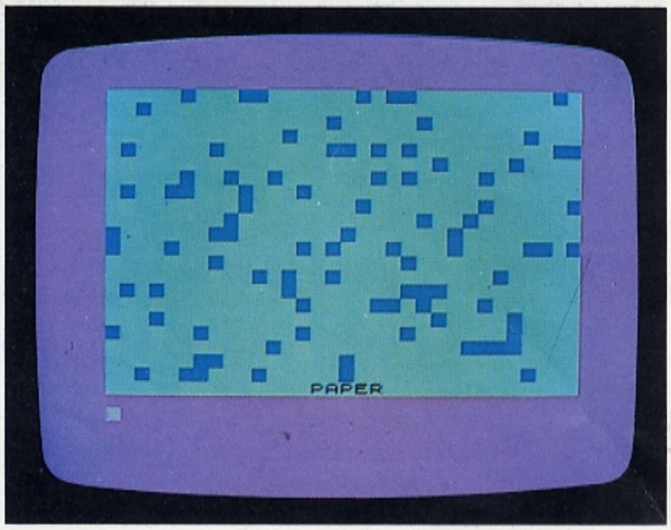
```
100 a=POINT (x,y)
```

a will be equal to zero if x,y is a PAPER colour, or it will be equal to 1 if x,y is an INK colour.

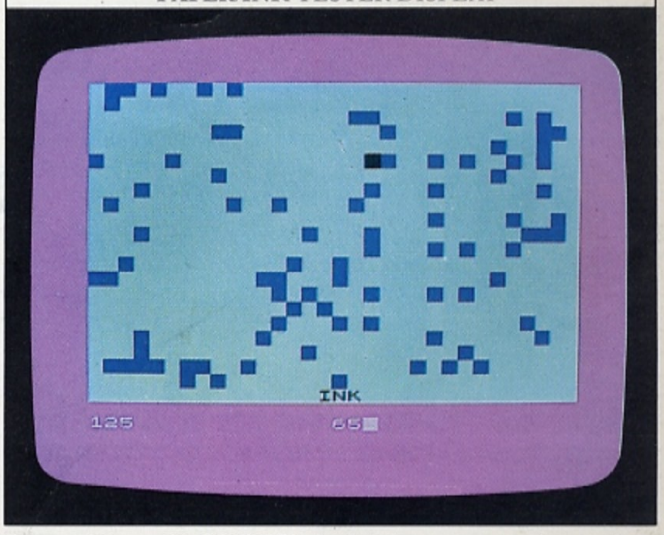
The next program produces INK squares at random, and then lets you test whether a particular pair of co-ordinates is covered by a PAPER or INK colour:

PAPER/INK TESTER

```
  10 BORDER 3: PAPER 5: INK 1: C
LS
  20 FOR n=1 TO 100
  30 LET r=INT (RND*21)
  40 LET c=INT (RND*32)
  50 PRINT AT r,c;"■"
  60 NEXT n
  70 INK 0
  80 INPUT x,y
  90 LET a=POINT (x,y)
 100 PLOT x,y
 110 IF a=0 THEN PRINT AT 21,14;
"PAPER"
 120 IF a=1 THEN PRINT AT 21,14;
" INK "
 130 GO TO 80


0 OK, 0:1
```



PAPER/INK TESTER DISPLAY



Lines 20 to 60 PRINT dark blue squares in random positions on a cyan background. The bottom line of the screen is left blank for use later. Line 70 switches to black INK. Line 80 waits for you to type in the x and y co-ordinates of any point on the screen.

Type in an x co-ordinate (0–255), press ENTER, and then type in the y co-ordinate (0–176) and press ENTER again. Line 90 will then test whether this point is PAPER or INK. Line 100 PLOTs a black point at x,y to mark it. Note that this program would not work if lines 90 and 100 were reversed, because x,y would always be an INK colour, coming immediately after PLOT x,y.

The result of line 90 is PRINTed on row 21 by lines 110 and 120. If x,y is a PAPER colour, the point ENTERed appears as a small black dot on the screen. If x,y is an INK colour, the whole character-sized square turns black because if you change the INK colour of a single point in any character position, the whole character position changes to the new INK colour.

Using POINT with animation

Now you know how POINT is used, you can try out a practical application of the command. You have probably come across a situation where you have several characters on the screen, but you don't want to go to the trouble of storing their positions and recalculating when any of the characters move in order to keep track of them during animation.

The next program gets around that problem. It uses POINT to test for the position of obstacles scattered over the screen as a character bounces between the screen edges. There are only a few obstacles in this display, but you can easily increase the number by altering the range of the loop between lines 20 and 50:

MINESWEEPER PROGRAM

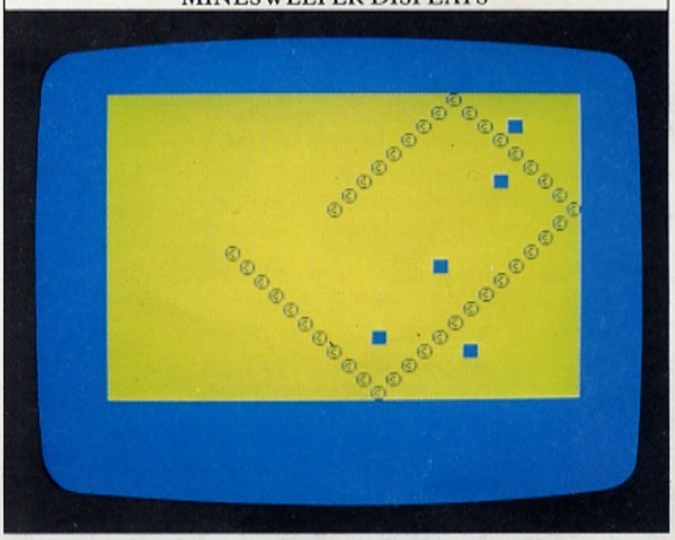
```
 10 BORDER 1: PAPER 6: INK 1: C
LS
 20 FOR n=1 TO 5
 30 LET s=INT (RND*22): LET d=I
NT (RND*32)
 40 PRINT AT s,d;"■"
 50 NEXT n
 60 LET r=10: LET c=7: LET v=1:
 LET h=1
 70 IF r=s AND c=d THEN GO TO 2
0
 80 PRINT AT r,c;" "
 90 IF r=0 OR r=21 THEN LET v=-
v: BEEP 0.1,5
 100 IF c=0 OR c=31 THEN LET h=-
h: BEEP 0.1,29
 110 LET r=r+v: LET c=c+h
 120 LET x=c*8+4: LET y=176-(r*8
+4)
 130 IF POINT (x,y)=1 THEN GO TO
 190

■
```

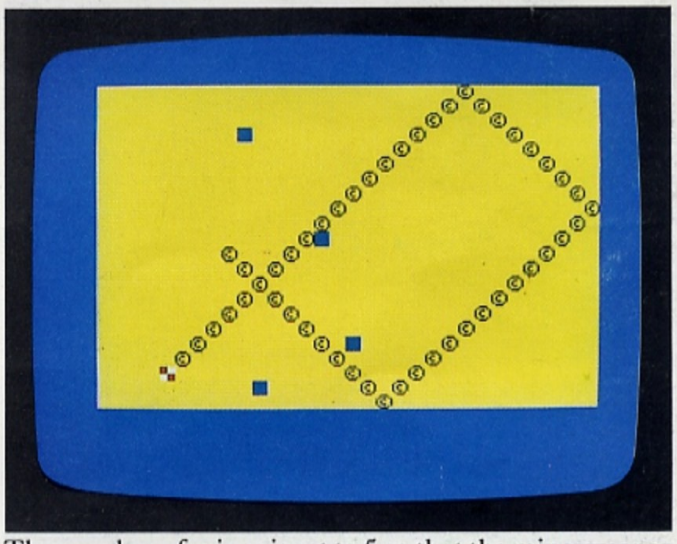
```
 140 PRINT INK 0;AT r,c;"©": PAU
SE 3
 150 GO TO 80
 160 FOR n=1 TO 10
 170 FOR a=1 TO 6
 180 PRINT INK a;AT r,c;"▚": BEE
P 0.05,10
 190 LET a=a+1
 200 PRINT INK a;AT r,c;"▞": BEE
P 0.05,30
 210 NEXT a
 220 NEXT n
 230 GO TO 10



0 OK, 0:1
```

The program uses POINT to locate INK characters on a PAPER screen. Lines 20 to 50 PRINT dark blue mines (squares) on a yellow PAPER screen. Line 60 sets the start position for the minesweeper (a copyright symbol). If this start position coincides with any of the random blue squares, a new set of random squares is PRINTed. Lines 80 to 150 form a routine that you have encountered several times now. It makes the minesweeper bounce around the screen from side to side and top to bottom. Every time the minesweeper reaches one of the screen's four edges, its direction is reversed and a BEEP sounded. The BEEP produced when the minesweeper reaches the sides of the screen is much higher in pitch than when it reaches the top or bottom edge.

For each position the minesweeper symbol moves into, line 120 converts its row and column co-ordinates into the x,y graphics co-ordinates of the mid-point of the position. Line 130 then checks whether this is a PAPER or INK colour. If it is a PAPER colour, the program jumps back to line 80, erases the minesweeper and rePRINTs it in a new position (calculated by line 110). If it's an INK colour, line 130 returns the value 1 and the program jumps to the explosion subroutine at line 190. This repeatedly PRINTs two graphics characters in a range of different colours, accompanied by sound effects. After every explosion, the program jumps back to line 10 and starts again:

MINESWEEPER DISPLAYS





The number of mines is set to 5 so that the minesweeper has enough space to go back and forth across the screen several times before it hits anything. You can provoke an explosion more quickly by increasing this number in line 20. Masking the PAUSE in line 140 will also speed up the program.

Using POINT in routines like this enables you to program the computer to make a complex decision with a simple series of commands. POINT is very useful in games where you are controlling an object that has to be steered around the screen and kept clear of walls or other obstacles. You can test your skill at computerized navigation by using POINT to direct the program into a crash or penalty subroutine when you steer into INK.

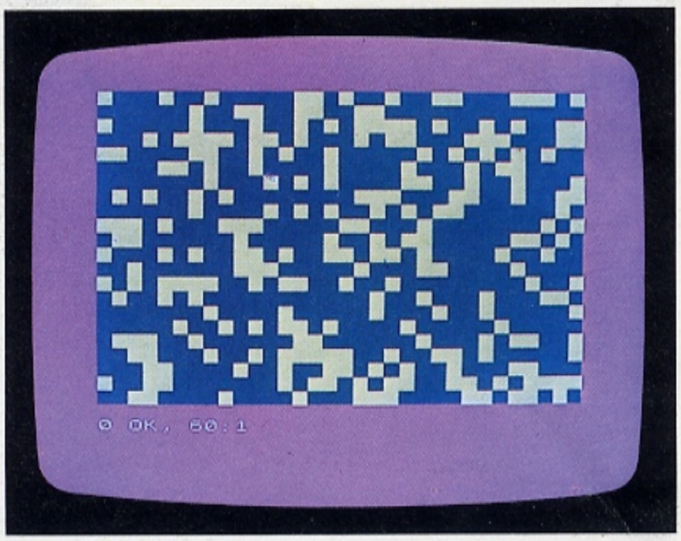
PATTERNS WITH SYMMETRY

If you make the Spectrum produce a pattern at random, the display produced will be entirely unpredictable. Using a simple technique, it is possible to generate a random display in a way that produces a symmetrical result. An unpredictable display makes a random pattern all over the screen, a technique that is useful for producing a background "galaxy" effect for a star wars game. But with a symmetrical display, the first part of the program produces a random pattern in one quarter of the screen, while subsequent parts of the program repeat this random pattern in each of the other three corners of the screen. The effect is like seeing part of the display reflected in a mirror.

To see how you do this, first you need a program that produces a random display. On the Spectrum this is very easy:

RANDOM DISPLAY PROGRAM

```
LS
  10 BORDER 3: PAPER 1: INK 7: C
  20 FOR n=1 TO 300
  30 LET c=INT (RND*32)
  40 LET r=INT (RND*22)
  50 PRINT AT r,c;"■"
  60 NEXT n


0 OK, 0:1
```



This short program repeatedly produces row and column co-ordinates for a random location on the screen. Although the column furthest to the right is numbered 31, the expression in line 30 that produces the column number includes 32, because INT rounds numbers down to the nearest integer, so the highest value that INT(RND*32) can have is 31. The graphics symbol on key 8 is used to produce the display. The program positions 300 squares at random. There are a total of 704 character positions, so, at most, about half the screen is covered. There's nothing to prevent lines 30 and 40 producing the same pair of co-ordinates on several occasions, so that often the INK squares will take up a smaller area than this.

Reflecting a random pattern

Now you can use the graphics symbol, but in a slightly different way, to turn this entirely random pattern into a random pattern with symmetry:

SYMMETRICAL DISPLAY PROGRAM

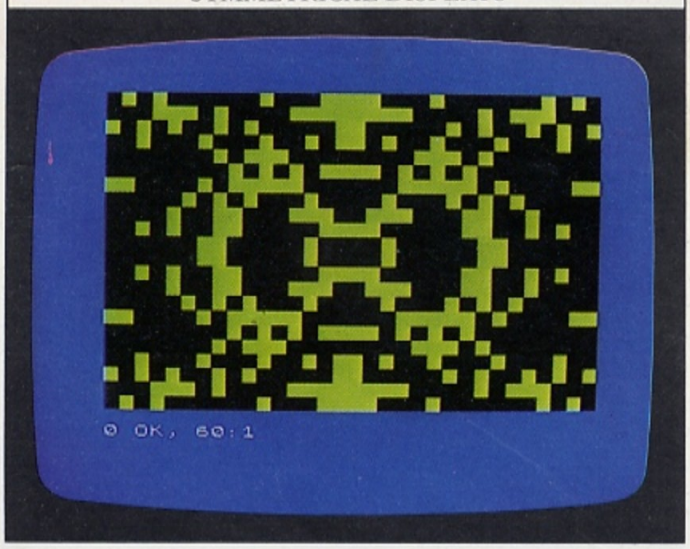
```
LS
  10 BORDER 1: PAPER 0: INK 4: C
  20 FOR n=1 TO 75
  30 LET c=INT (RND*16)
  40 LET r=INT (RND*11)
  50 PRINT AT r,c;"■";AT r,31-c;
"■";AT 21-r,c;"■";AT 21-r,31-c;"
■"
  60 NEXT n


0 OK, 0:1
```

This program divides the screen into four equal quarters. All of the co-ordinates that lines 30 and 40 produce lie in the top left-hand quarter. They are copied onto corresponding locations in the other three quarters. Because each pair of co-ordinates produced gives rise to four symbols PRINTed on the screen, the program only needs to carry out a quarter the number of loops as the previous program. The maximum value of n in line 20 is therefore reduced to 75, but this produces 300 images as before. The first of the next pair of screens shows a typical RUN of the program. The second screen shows the result of adapting the program by changing line 10 to:

```
10 BORDER 1:PAPER 0:INK 6:CLS
```

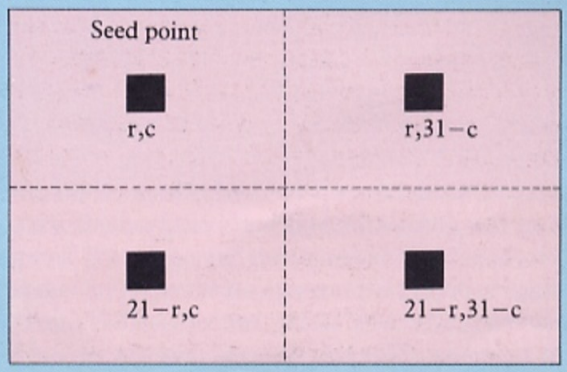
and then using a smaller graphics character (this time on the 3 key). The program will produce a different display every time it is RUN:

SYMMETRICAL DISPLAYS



The program works out the positions of the mirror images of each original "seed" symbol by performing some simple arithmetic on the co-ordinates r,c. The first mirror image is on the same row as the seed, but on the opposite side of the screen. Its co-ordinates are therefore r,31−c. Each point is reflected by two others in the bottom half of the screen.

USING SYMMETRICAL CO-ORDINATES



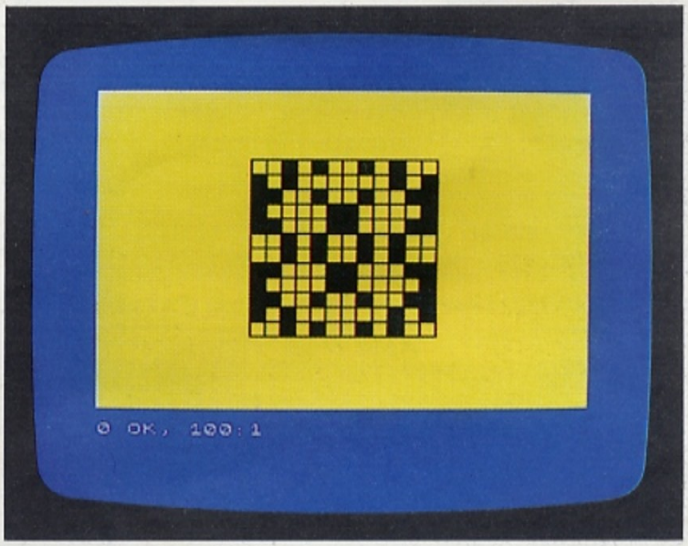
Each is as far up from the bottom of the screen as the seed point is down from the top, so their co-ordinates are the 21−r,c and 21−r, 31−c in line 50.

Programming a crossword grid

Most crosswords are constructed from a symmetrical grid, and by reflecting the black squares in one corner of the crossword grid, you can program the computer to PRINT the complete pattern. Again, "seed" squares are positioned at random:

CROSSWORD PROGRAM

```
 10 BORDER 1: PAPER 6: INK 0: C
LS
 20 FOR y=40 TO 136 STEP 8
 30 PLOT 80,y: DRAW 96,0
 40 PLOT y+40,40: DRAW 0,96
 50 NEXT y
 60 FOR n=1 TO 20
 70 LET c=10+INT (RND*12)
 80 LET r=5+INT (RND*12)
 90 PRINT AT r,c;"■";AT r,31-c;
"■";AT 21-r,c;"■";AT 21-r,31-c;"
■"
100 NEXT n

0 OK, 0:1
```



Lines 20 to 50 DRAW the grid. The grid is 12 squares across and 12 squares down. This line spacing is chosen so that each blank square that lies within the grid is the same height and width as a keyboard character (8 graphics pixels across and 8 down). The RND statements in lines 70 and 80 are chosen so that the random co-ordinates produced by them always lie within the crossword grid. Line 90 then PRINTs the seed point and its three mirror images as in the previous program. Wherever a graphic square is PRINTed, it fills a blank square in the grid.

TRACING ERRORS

The Spectrum is much more helpful than many microcomputers in tracing errors or "bugs" in programs. Some computers will let you type in almost anything; you only discover that your program does not make sense when you RUN it and get a whole succession of error reports. The Spectrum, on the other hand, checks each line you type in for errors before you can use it in a program.

Although you cannot spell a keyword wrongly, because the Spectrum's entry system works on single keys, it is easy to use punctuation incorrectly. If you miss out a semi-colon after an AT, or separate two commands with something other than a colon, for example, a flashing question mark appears on the line when you press the ENTER key. Its position indicates where the error is. All you have to do is move the cursor back down the line and correct the error. If the cause of the error isn't immediately apparent, check that you have used the commands in question correctly.

This system of entry-checking does not mean that you can't make a mistake when programming the Spectrum. You can easily write a program which the Spectrum will accept, but which is still riddled with errors. Here is a program that is full of errors. If you key it in (the computer will accept all the lines) you can then see how to go about a thorough "debug".

The program is the "hangman" game from page 27, but it has been written and ENTERed hurriedly, so that it will not work. Don't cheat by looking back at the correct program! Go through the listing and see if any bugs are obvious to you. See how many you can find, and try and work out how you would correct them. Then check your results against the bugs that are explained on these pages.

Key in the program as it is and then try to use it:

BUGGED "HANGMAN" PROGRAM

```
  10 BORDER 0: CLS : PRINT AT 1,
12;"HANGMAN"
  20 PRINT AT 10,2;"Ask a friend
 to type a word"
  30 PRINT AT 12,3;"or phrase fo
r you to guess"
  40 PRINT AT 18,4;"DON'T LOOK A
T THE SCREEN"
  50 PRINT AT 20,0;"Press ENTER
when you're finished"
  60 INPUT a$
  70 LET l=LEN a$
  80 FOR n=1 TO l
  90 IF a$(n)=" " THEN PRINT AT
11,(32-l)/2+n;"■"
 100 PRINT AT 11,(32-l)+n;"j": N
EXT l
 110 PRINT AT 2,12;"HANGMAN";AT
5,6;"-:letter    ■:space"
 120 PRINT AT 18,6;"     Try a le
tter     ";AT 19,0;"Press 1 to gu
ess the whole thing"

scroll?
```

```
 130 INPUT t$
 140 IF t$="2" THEN STOP
 150 IF t$="1" THEN GO TO 190
 160 PRINT AT 16,12;"score=";s
 170 FOR n=1 TO l
 180 IF t$=a$(n) THEN PRINT AT 1
1;(32-l)+n;t$
 190 NEXT n
 200 GO TO 130
 210 PRINT AT 18,6;"Try the whol
e thing": INPUT t$
 220 IF t$=a$ THEN PRINT AT 11,1
2;"CORRECT";AT 13,12;"score=";s+
1
 230 GO TO 120



0 OK, 0:1
```

When you try to RUN the program you will find that the title frame comes up and you are invited to ENTER the test string. But when your press ENTER, the error report "1 NEXT without FOR, 100:2" appears. This means that the program has stopped at the second statement in line 100. LIST the program. You will see that a loop containing n begins at line 80, but the NEXT statement in line 100 says NEXT 1. To make the loop work properly, change this to NEXT n. Now try the program again:

"CRASHED" PROGRAM DISPLAY

```
                HANGMAN


  Ask a friend to type a word
                          jjjjjjj
    or phrase for you to guess


       DON'T LOOK AT THE SCREEN
Press ENTER when you're finished
```

This time the title frame and test string entry work properly, but the title frame stays on when the next phase of the game starts. That's easily dealt with. Add :CLS to line 70. You may also have got a "B Integer out of range" error report on screen. The characters representing the test string are also PRINTed in the wrong position. They may have run out of the side of the screen as shown above.

The expression in line 100 should PRINT all the characters in the middle of the screen. It should read:

```
(32-1)/2+n
```

but in the program, the division by 2 has been missed out altogether.

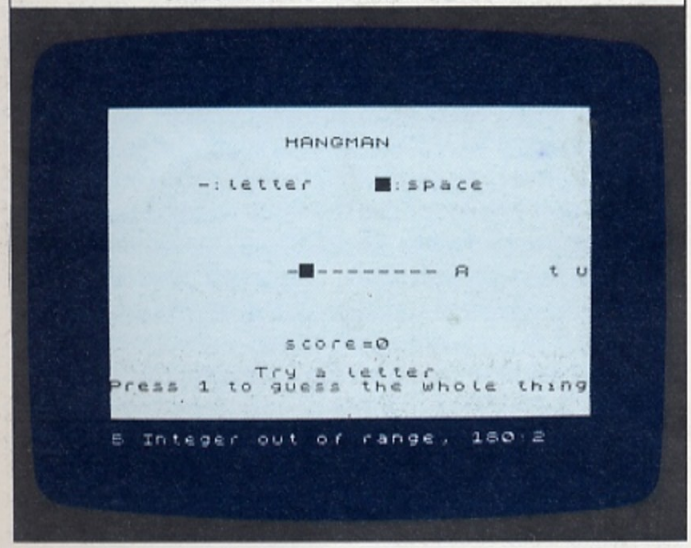
The display instructions tell you that a hyphen represents a letter, but a row of the letter j is PRINTed instead. Moreover, if you've ENTERed a test string containing any spaces, you'll have found that the black square graphics symbol has not appeared.

The j-line has appeared because the programmer forgot to press the shift key to get a hyphen. Correct that, then look at line 90 again. This line checks for a space in the test string and if necessary PRINTs a black square, but this is masked by whatever line 100 PRINTs. If you add: NEXT n to the end of line 90, then when a black square is PRINTed, the program will move on to the next value of n and the next character in the test string.

Further test RUNs

Now, when you RUN the program, as soon as you ENTER your first guess at a letter, you will get a "2 Variable not found, 160:1" error report. The only variable in line 160 is s, which represents the score. It seems straightforward – it PRINTs the score. This line is, in fact, correct. The error report is PRINTed because the computer doesn't have an initial value for s, which it can PRINT. So, add :LET s=0 to the end of line 70. When you have keyed in these corrections, RUN the program again:

"CRASHED" PROGRAM DISPLAY



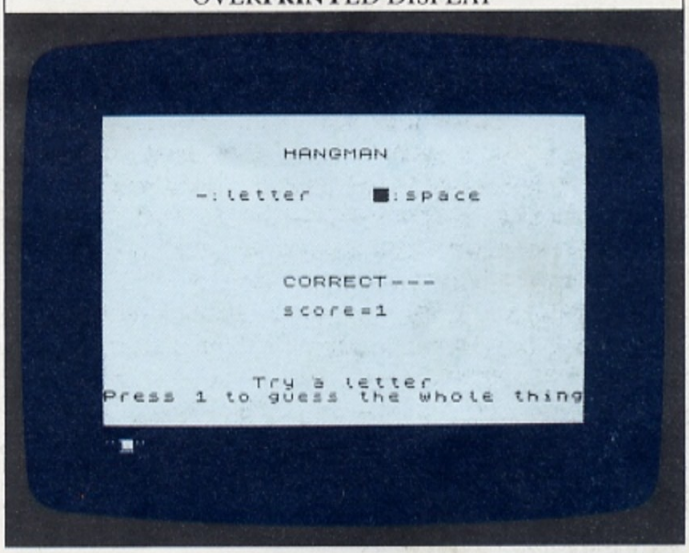
Now, when you type in a correct guess, it will probably appear to the right of the screen. Alternatively, if it represents a letter that appears towards the end of the test string, you may even get a "B Integer out of range, 180:2" error report, because the computer has tried to PRINT the character off the screen altogether. Once again, the division by 2 has been omitted from the column expression. It should read as follows:

```
(32-1)/2+n
```

The game now seems to work properly, but if you look at the score, you will see that it's not increasing. The score should go up by 1 after each guess and this has not been written into the program, so add LET s=s+1: to the beginning of line 160.

Although single-letter guesses will work properly now, the computer refuses to respond correctly when you press 1 to guess the whole string. Line 150 should make the program jump to the whole word guess routine, but it actually leads to line 190, which is simply NEXT n. This is a very common fault after you have renumbered any of the program's lines, and perhaps forgotten to renumber the GOTOs and GOSUBs. Change the GOTO statement in line 150 to GOTO 210. The following display should appear:

OVER**PRINT**ED DISPLAY



Try typing in a correct guess for the whole string. The "CORRECT" frame is PRINTed over the previous game frame. That's easily put right – change line 220 to:

```
220 IF t$=a$ THEN CLS:PRINT AT 11,12;
    "CORRECT"; AT 13,12;"score=";s+1
```

Finally, the program refuses to stop. When you've made a correct guess, the program immediately restarts. So, add :STOP to the end of line 220. The program should now RUN properly.

If you are developing a program, constant checking should prevent all but a few bugs from slipping into the final listing. When you're testing a program that you've written, put it through all the situations it will meet in use. If it's supposed to have safeguards to stop it "crashing" in some circumstances, test them too. Testing the end of a growing program with GOTO will ensure that each routine actually works before you move on to the next.

SPEEDING UP PROGRAMS

One of the features of BASIC is that, if you want to, you can write programs without much prior planning. With many other computer languages, a much more organized approach is needed. Although BASIC is easy to use, it doesn't encourage the best programming.

One area where good and bad programming show up is in RUNning speed. While it's not of great importance that a short program RUNs quickly, RUNning speed does become more significant as your programs become longer and more complex. Ideally, you will want them to RUN as quickly as possible. On these two pages you can see how a poorly written program compares with a version that has been streamlined.

A "slow" program

The following program is of the type used on page 23 to show how arrays can produce spreadsheets on the screen. Here is the program listing:

"SLOW" TAX TABLE PROGRAM

```
  10 REM Tax table
  20 DEF FN t()=(65536*PEEK 2367
4+256*PEEK 23673+PEEK 23672)/50
  30 LET T=FN t()
  40 DIM p(8): DIM n(8)
  50 DATA 1.20,2.30,1.10,2.00,1.
80,2.90,1.45,2.05
  60 DATA 14,16,24,17,15,11,14,2
0
  70 FOR a=1 TO 8
  80 READ p
  90 LET p(a)=p
 100 NEXT a
 110 FOR a=1 TO 8
 120 READ n
 130 LET n(a)=n
 140 NEXT a
 150 CLS
 160 FOR y=20 TO 164 STEP 16
 170 PLOT 28,y: DRAW 200,0
 180 NEXT y

scroll?
```

```
 190 PLOT 28,20: DRAW 0,144: PLO
T 68,20: DRAW 0,144: PLOT 92,20:
 DRAW 0,144
 200 PLOT 140,20: DRAW 0,144: PL
OT 180,20: DRAW 0,144
 210 PLOT 228,20: DRAW 0,144
 220 PRINT AT 2,6;"£";AT 2,9;"No
";AT 2,13;"SUB";AT 2,18;"TAX";AT
 2,23;"TOTAL"
 230 FOR a=1 TO 8
 240 PRINT AT 2*a+2,4;p(a)
 250 PRINT AT 2*a+2,9;n(a)
 260 PRINT AT 2*a+2,12;p(a)*n(a)
 270 PRINT AT 2*a+2,18;INT (15*p
(a)*n(a)+0.5)/100
 280 PRINT AT 2*a+2,23;p(a)*n(a)
+INT (15*p(a)*n(a)+0.5)/100
 290 NEXT a
 300 PRINT AT 21,4;"Time taken:
";FN t()-T


0 OK, 0:1
```

The program takes a list of unit prices, numbers of items and – given the current tax rate – calculates the total cost of the items. The information is presented as a table. To test the program's RUNning speed, lines 20 and 30 start timing the program as soon as it starts and line 300 PRINTs its total RUNning time on the bottom line of the screen:

"SLOW" TAX TABLE DISPLAY

£	No	SUB	TAX	TOTAL
1.2	14	16.8	2.52	19.32
2.3	16	36.8	5.52	42.32
1.1	24	26.4	3.96	30.36
2	17	34	5.1	39.1
1.8	15	27	4.05	31.05
2.9	11	31.9	4.79	36.69
1.45	14	20.3	3.04	23.34
2.05	20	41	6.15	47.15

Time taken: 2.2600002

0 OK, 300:1

The unit prices are stored in the DATA statement at line 50, and the numbers of each item in line 60. Both arrays are dimensioned by line 40. The loop at lines 70 to 100 READs the unit price DATA into the one-dimensional, eight-element array p. A similar loop at lines 110 to 140 READs the numbers DATA into the n array. Lines 150 to 220 clear the screen, DRAW the grid of intersecting lines and PRINT the column headings – "No" for number and "SUB" for subtotal. Lines 230 to 290 calculate and PRINT the figures to fill the grid.

On looking through the listing, you might have noticed that two loops use the same variable (a) over the same range of values (1 to 8). It would have been possible to combine these loops and save space and time. Also the row number given by 2*a+2 is newly calculated in each line. This wastes some time. Finally, the same expression (p(a)*n(a)) appears three times in lines 270 to 290, calculated again each time it is required. All this shows that the program has been badly thought out, and needs improvement. It takes about two and a quarter seconds to RUN. Now see how much you can shave off this by using more economical programming techniques.

Techniques for time-saving

Here is the program again, but in a modified form. It produces exactly the same display, but in places the program statements are quite different:

IMPROVED TAX TABLE PROGRAM

```
  10 REM Fast Tax table
  20 DEF FN t()=(65536*PEEK 2367
4+256*PEEK 23673+PEEK 23672)/50
  30 LET T=FN t()
  40 DIM p(8): DIM n(8)
  50 DATA 1.20,14,2.30,16,1.10,2
4,2.00,17,1.80,15,2.90,11,1.45,1
4,2.05,20
  60 FOR a=1 TO 8
  70 READ p,n
  80 LET p(a)=p: LET n(a)=n
  90 NEXT a
 100 CLS
 110 FOR y=20 TO 164 STEP 16
 120 PLOT 28,y: DRAW 200,0
 130 NEXT y
 140 PLOT 28,20: DRAW 0,144: PLO
T 68,20: DRAW 0,144: PLOT 92,20:
 DRAW 0,144: PLOT 140,20: DRAW 0
,144: PLOT 180,20: DRAW 0,144: P
LOT 228,20: DRAW 0,144

scroll?
```

```
 150 PRINT AT 2,6;"£";AT 2,9;"No
";AT 2,13;"SUB";AT 2,18;"TAX";AT
 2,23;"TOTAL"
 160 FOR a=1 TO 8
 170 LET r=2*a+2: LET b=p(a)*n(a
): LET d=INT (15*b+0.5)/100: LET
 e=b+d
 180 PRINT AT r,4;p(a);AT r,9;n(
a);AT r,12;b;AT r,18;d;AT r,23;e
 190 NEXT a
 200 PRINT AT 21,4;"Time taken:
";FN t()-T




0 OK, 0:1
```

The first thing you will notice is that ten lines have been lopped off the listing. To achieve this, a lot of unnecessary calculations have been removed and several similar statements have been neatly condensed into a single line.

The two listings are identical up to line 50. All of the DATA has been written into this line, but it's also been reorganized. This is necessary because the loops that load the DATA into the two arrays have been rewritten. There is now a single loop at lines 60 to 90, saving four lines. Both the price and number are READ by READ p,n in line 70. The DATA in line 50 must, therefore, be presented in this way – a price followed by a number, and so on.

Lines 110 to 150 are identical to lines 160 to 220 in the first program, except that several of the PLOT and DRAW statements have been written into a single statement instead of three statements taking up three lines. The computer can deal with multiple statements in a single line substantially faster than with several separate statements, each occupying a different line.

There is now a major change in the way the grid is filled with figures. At the beginning of each loop, once the value of a has been established by line 160, all the calculations necessary for that loop are carried out once and for all by line 170. The variables used work in the following way:

TAX TABLE VARIABLES

The program uses four variables to fix positions and carry numbers calculated for use in the table.

Variable(s)	Function
r	Fixes row number where DATA will be PRINTed
b	Stores product of unit price and number of items (SUB column)
d	Stores amount of tax at 15% rate
e	Stores total cost (subtotal + tax)

Line 180 is now a lot less complicated, because all the calculations have already been done. It's just a matter of PRINTing the correct variable in the correct position in the grid. If you RUN the program in its improved form, the display now shows the time saved by making these changes:

IMPROVED TAX TABLE DISPLAY

£	No	SUB	TAX	TOTAL
1.2	14	16.8	2.52	19.32
2.3	16	36.8	5.52	42.32
1.1	24	26.4	3.96	30.36
2	17	34	5.1	39.1
1.8	15	27	4.05	31.05
2.9	11	31.9	4.79	36.69
1.45	14	20.3	3.05	23.35
2.05	20	41	6.15	47.15

```
Time taken:  1.9799995
0 OK, 200:1
```

The changes have taken more than a quarter of a second off this short program, a saving of more than 12 per cent. You can imagine the saving in a longer program dealing with much more complicated numerical DATA or making lots of calculations for example.

When you are writing a long program, watch out for any repetition in the lines. The chances are that with some planning you could save both RUNning time and memory. Any large routine that is repeated a number of times will be worth making into a subroutine with GOSUB, while any calculations that crop up frequently may be carried out with FN. These devices, plus the various loops, can all be used to speed up a program. Because the speed with which your programs are RUN is slowed down by conversion into machine code, savings in BASIC are always worth considering.

HINTS AND TIPS

How to SAVE screen displays

Sometimes you may write a program which produces a display which you would like to be able to see again later. With the Spectrum you don't necessarily have to RUN the original program to do this. Instead, the direct commands:

```
SAVE "display" SCREEN$
```

can be used with a cassette recorder to store a picture called "display" – or any other name – on tape. The computer will then transmit to the tape the information about every pixel on the screen. When you want to see the image again, just play back the tape as usual, but with these commands:

```
LOAD "picture" SCREEN$
```

The computer will gradually scan across the television screen, PRINTing the pixels in exactly the same way as in the original display. By using this command, you can produce graphics directly without using program lines, but then store the result on tape cassette in exactly the same way that you would store a program.

Problems with functions

Two of the Spectrum's built-in functions, SQR and ↑, may sometimes cause problems in your programs. Because the square of any number is always positive, the function SQR cannot be used with a minus number, so if you are using SQR, make sure that this will not happen. The problem with the exponent function is less obvious. If you are mathematically-minded, you might have tried the following way of producing a number that is either +1 or −1 (this can be used in a program to produce lines or characters over the screen in an unpredictable way):

RANDOM GRAPHICS WITH EXPONENTS

```
10 PLOT 128,88
20 LET x=(RND*88)*-1↑(RND*2)
30 LET y=(RND*88)*-1↑(RND*2)
40 DRAW x,y
50 GO TO 10


0 OK, 0:1
```

The exponent function should produce a positive or a negative number, either +1 or −1 each time. In fact, it won't work. This is not because the maths is faulty, but because, like SQR, the Spectrum's exponent function will not work with a minus number. The answer will always come out positive – not much help for up-or-down movement. The best way to produce the effect is to set up a decimal figure from 0 to 0.99999999 with RND, and then use an IF ... THEN line to produce +1 or −1 like this:

```
200 LET a=RND
210 IF a>0.5 THEN LET a=1: IF a<=0.5 THEN
    LET a=-1
```

SQR used with a minus number will produce an error report. The exponent function with a minus number will not, so you may not realize at first why a program is RUNning oddly.

Setting boundaries with animation

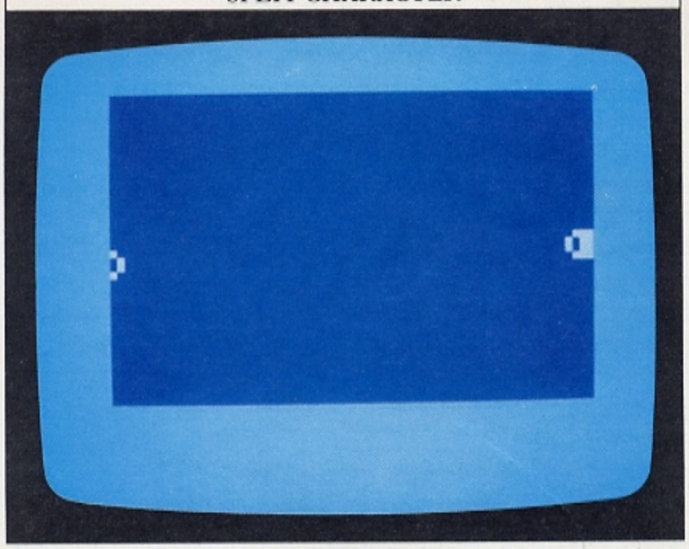
When you start animating graphics, you may find that they don't behave as you wish when they reach the edge of the screen. Here is a program that demonstrates this problem:

ANIMATION PROGRAM

```
  10 BORDER 5: PAPER 1: INK 7: C
LS
  20 LET a$="   "
  30 LET b$="   "
  40 LET r=10: LET c=0: LET a=1
  50 PRINT AT r,c;a$
  60 PRINT AT r+1,c;b$
  70 BEEP 0.02,5
  80 LET c=c+a
  90 BEEP 0.02,7
 100 IF c=31 OR c=0 THEN LET a=-
a
 110 BEEP 0.02,9
 120 GO TO 50


0 OK, 0:1
```

The program animates a small flying saucer symbol built up from two rows each of three characters with a space either side. The space erases the old image of the saucer as the new image is drawn one space to the left or right of the last position. Line 40 sets the starting position and sets a – the change in position of the saucer – to 1. Line 80 sets the position for the next PRINT). Line 100 checks whether the saucer has reached the edge of the screen (c=0 or c=31) and if so, it reverses the saucer's direction. Then back to line 50 to PRINT the new saucer:

SPLIT CHARACTER



But, as you can see, it doesn't work properly. The saucer appears to wrap round onto the next row down and then retreat back up. It's a common problem with animation programs.

Line 100 deals with how the saucer behaves at the screen's edges. Remember that c represents the column number of the first or furthest left character of the saucer. When the last saucer character (the furthest right) reaches the right edge of the screen, c has a value of 27, not 31. To rectify this, change line 100:

```
100 IF c=27 OR c=0 THEN LET a=-a
```

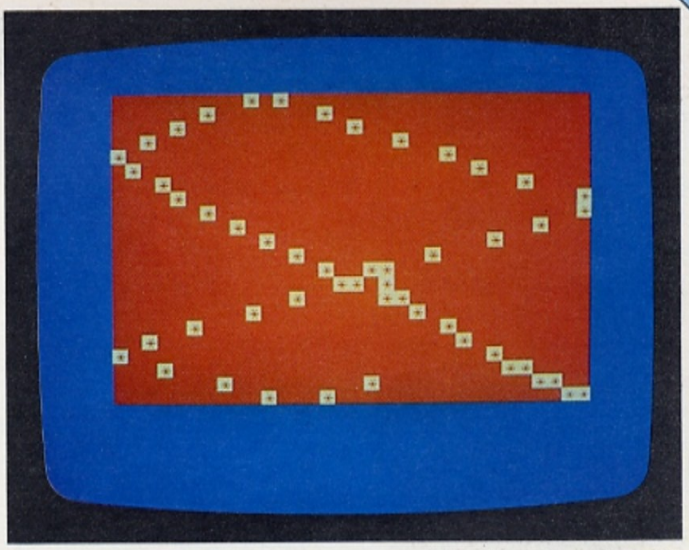
The program now takes into account the size of the object that is being animated.

Testing for the screen edge

The next program is similar to the previous one in that it moves an object around the screen, but this time the object is composed of only one character and it moves in both horizontal and vertical directions:

EDGE-TESTER PROGRAM

```
  10 BORDER 1: PAPER 2: INK 7: C
LS
  20 LET r=11: LET c=16: LET v=1
: LET h=1
  30 PRINT AT r,c;" "
  40 IF INKEY$="v" AND v>0 THEN
LET v=v+0.1
  50 IF INKEY$="h" AND h>0 THEN
LET h=h+0.1
  60 LET r=r+v: LET c=c+h
  70 IF c=0 THEN LET c=0: LET h=
-h
  80 IF c=31 THEN LET c=31: LET
h=-h
  90 IF r=0 THEN LET r=0: LET v=
-v
 100 IF r=21 THEN LET r=21: LET
v=-v
 110 PRINT AT r,c;"■"
 120 BEEP 0.02,60-5*r
 130 GO TO 30

0 OK, 0:1
```



Each time the character is erased by line 30, line 60 changes the row and column values by v and h respectively. These are initially set to 1. Because of this and the effect of lines 70 and 80, the character bounces around the screen just like a ball bouncing around inside an empty box.

Lines 40 and 50 allow you to change the ball's speed. If you press the V key and the ball is travelling downwards, the variable v is increased by 0.1, increasing the ball's downward speed. In the same way, if you press the H key and the ball is travelling to the right, the ball's horizontal speed increases. But try it. When you press either V or H, the ball speeds towards the edge of the screen, and then the program throws up an error message – either "B Integer out of range, 110:1" or "5 Out of screen, 110:1". If you don't touch the keyboard, the program RUNs correctly, proving that the animation lines have been written properly, but for some reason will not work with INKEY$.

Consider for a moment what would happen if the ball was at position 20,10 and travelling downwards and to the right. At the same instant, you press the V key. Line 40 increases v from 1 to 1.1. Line 60 changes r to 21.1 and c to 11. Line 100 checks whether r equals 21. It doesn't. So, line 110 tries to PRINT the ball AT 21.1,11, producing the "Out of screen" error message.

The problem is that you can no longer predict the exact value of r or c when the ball reaches a screen edge. So, change some lines to cope with the situation and conditions you do know about – that is, neither r nor c can have values less than zero, r cannot be greater than 21 and c cannot be greater than 31. So the lines are now:

```
70 IF c<0 THEN LET c=0:LET h=-h
80 IF c>31 THEN LET c=31:LET h=-h
90 IF r<0 THEN LET r=0:LET v=-v
100 IF r>21 THEN LET r=21:LET v=-v
```

Once you have made these changes, the program will correctly test for the edge of the screen.

CONVERTING PROGRAMS

One of the problems with BASIC is that it exists in many different forms or "dialects". The Spectrum uses a form of BASIC which includes some commands – BORDER, PAPER and INK for example – that other machines do not have. Similarly, other micros may produce effects that can be achieved on the Spectrum, but by completely different techniques; nowhere is this more evident than in graphics programs. All this makes "program mobility" – trying to use the same program on different computers – very difficult. You can see this in software retailing where many programs are only available for one type of machine.

If you do see a program in a magazine or book, or a friend shows you a program RUNning on another computer, you may decide that it's worth converting to RUN on your Spectrum. To do this you must know not only how the machines concerned differ, but also you must be able to understand why and how a program does what it does. Then you can break the program down into blocks or subroutines and finally look at it line by line. But in the same way as you can rarely translate a message into a foreign language word for word, you cannot simply translate each "foreign" program statement directly into the equivalent statement for your computer. It may be more economical in time and more efficient to completely rewrite a section of program using the best commands available on your machine.

Points to watch for when converting

One of the most variable aspects of BASIC is punctuation and spacing. The Spectrum's error-checking system will ensure that you do not ENTER any lines that have incorrect spelling or punctuation, but you don't want to have to keep experimenting until you get things right. Full stops, colons and semi-colons are used in different ways by many machines, and you will often need to make punctuation changes before you can use lines on your Spectrum.

Remember that any text or graphics co-ordinates are likely to need changing. Because different computers often have different display resolutions, the co-ordinates used to produce displays can rarely be incorporated without alterations in converted programs. Making a note of the other micro's text and graphics grid limits will help you here.

When converting programs, you must always be on the lookout for commands that relate directly to a machine's operating system, because they will have no meaning for the Spectrum. Similarly, commands like PEEK and POKE on the Spectrum cannot be used in a program converted for other machines, as they refer to the Spectrum's memory addresses. Watch out too for commands that use a computer clock: you will need to completely rewrite routines to make use of the Spectrum's timing system.

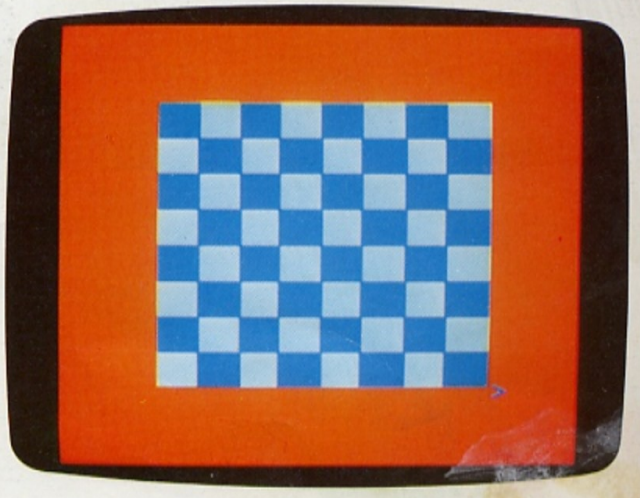
The list of these problems is quite a long one, but once you have tried some program conversion, you will soon learn what is safe "standard" BASIC, and what is "dialect". It is a good idea when converting programs to have a look through the other computer's manual, so you can pick out any commands which look unfamiliar. If you know what these do in advance, you will find the process of translating a lot easier.

Converted and original listings

The following listing is an example of a program written specifically for another computer, the Acorn BBC Micro. It sets up a games board – a routine that you might want to use on your Spectrum. But it won't RUN on the Spectrum in its present form:

BBC MICRO GAMES BOARD PROGRAM

```
>LIST
  10 MODE2:VDU5
  20 PROCSCREEN
  30 GCOL0,4:Y=832
  40 FOR ROW=1 TO 7 STEP 2
  50 LEFT=240:RIGHT=840
  60 PROCBOARD
  70 LEFT=340:RIGHT=940:Y=Y-80
  80 PROCBOARD
  90 Y=Y-80
 100 NEXT ROW
 110 END
 120 DEF PROCSCREEN
 130 GCOL0,129:CLG:GCOL0,6
 140 MOVE 240,832:MOVE 1040,832
 150 PLOT 85,240,192:PLOT 85,1040,192
 160 ENDPROC
 170 DEF PROCBOARD
 180 FOR X=LEFT TO RIGHT STEP 200
 190 MOVE X,Y:MOVE X+100,Y
 200 PLOT 85,X,Y-80
 210 PLOT 85,X+100,Y-80
 220 NEXT X
 230 ENDPROC
>_
```



Almost everthing in this program is peculiar to the BBC Micro and cannot be used by the Spectrum. The BBC Micro's colour commands and the way in which it calls special kinds of subroutines, called procedures, are entirely different. That's why it is necessary to understand how the program works before you can start converting it. You will find that it is a great help if you can see the program RUNning rather than working only from a printed listing.

The most straightforward way to convert or translate this program is to take each block, identify its function, and then write a routine that will carry out the same function in Spectrum BASIC. Lines 10 to 110 form the main program, which jumps to two subroutines called PROCSCREEN and PROCBOARD. These are called by name instead of line number. PROCSCREEN draws a large cyan square on a red background. The main program draws a row of alternately coloured squares across the screen (lines 50 and 60) then moves down (line 70) and draws a second row of squares. The second row is displaced to the right of the first. Four such pairs of rows are produced to form a chessboard. The BBC Micro builds each square from a pair of triangles, a facility not available on the Spectrum.

You can now rewrite the program to RUN on the Spectrum. Because so much of the program is in BBC BASIC, there is nothing to be gained by trying to translate it line by line. It's quicker and easier to examine the program and then work out how the Spectrum could best produce the same result:

SPECTRUM GAMES BOARD PROGRAM

```
  10 PAPER 2: BORDER 2: INK 5: C
LS
  20 FOR r=3 TO 18
  30 FOR c=8 TO 23
  40 PRINT AT r,c;"■"
  50 NEXT c
  60 NEXT r
  70 INK 1
  80 FOR r=3 TO 15 STEP 4
  90 FOR c=8 TO 20 STEP 4
 100 PRINT AT r,c;"▀▀"
 110 PRINT AT r+1,c;"▀▀"
 120 PRINT AT r+2,c+2;"▀▀"
 130 PRINT AT r+3,c+2;"▀▀"
 140 NEXT c
 150 NEXT r



K
```

More BASIC differences

The next program, also written in BBC BASIC, lists ten names on the screen. Each name is accompanied by a number – it might be a list of the best scores for a game. At first sight the method looks quite familiar in that it uses arrays, and should be easily adaptable for the Spectrum:

BBC MICRO ARRAY PROGRAM

```
>LIST
  10  DIM NAME$(10):DIM SCORE(10)
  20  DATA JOHN,400,ELLEN,246,FRED,115,
GEORGE,341,KATE,692,DEBBIE,944,JAMES,443
,TINA,672,JUDITH,195,TONY,733
  30 FOR N=1 TO 10
  40 READ NAME$,SCORE
  50 NAME$(N)=NAME$:SCORE(N)=SCORE
  60 NEXT N
  70 CLS
  80 PRINT TAB(10,1)"NAME","SCORE"
  90 PRINT TAB(10,2)"**************"
 100 FOR N=1 TO 10
 110 PRINT TAB(10,2*N+2)NAME$(N),;SCORE
(N)
 120 NEXT N
>_
```

The Spectrum can produce this, but watch out for a typical hidden problem. The Spectrum dimensions arrays in a different way. The BBC's name array is dimensioned by:

```
10 DIM NAME$(10)
```

but the Spectrum requires the length of the strings to be specified as well:

```
20 DIM N$(10,6)
```

Also the strings in the Spectrum's DATA must be enclosed in quotation marks. Otherwise the programs are very similar:

SPECTRUM ARRAY PROGRAM

```
  10 BORDER 0: CLS
  20 DIM N$(10,6): DIM s(10)
  30 DATA "John",400,"Ellen",246
,"Fred",115,"George",341,"Kate",
692,"Debbie",944,"James",443,"Ti
na",672,"Judith",195,"Tony",733
  40 FOR n=1 TO 10
  50 READ N$(n),s(n)
  60 NEXT n
  70 CLS
  80 PRINT AT 5,8;"NAME","SCORE"
  90 PRINT AT 6,8;"**************
"
 100 FOR n=1 TO 10
 110 PRINT AT n+7,8;N$(n),s(n)
 120 NEXT n




0 OK, 0:1
```

When you are trying to convert programs that seem to use Spectrum-compatible commands, be careful that you do not fall into the trap of assuming that they operate in exactly the same way. Standard words like DIM, INKEY$, TAB and so on, can make converting tricky if you do not realize that many computers use the same commands quite differently.

USING A PRINTER

Although you can write, edit and RUN programs on screen and store them on cassette, a paper print-out is still the best way of examining a listing closely. A print-out can also let you keep the product of a program RUN, which is very useful if you want to look over the results again. So, sooner or later, you will want to add a printer to your system.

The Sinclair printer is of the thermal/electrostatic type. Sparks jump from a pair of moving metal needles onto the special aluminium-coated paper, burning off the aluminium in places to reveal a black ink layer underneath. The spark pattern, and therefore the character printed, is controlled by the computer.

Printer enabling commands

The printer is connected up to the computer by the edge connector on the Spectrum's back panel. When the printer is plugged in, it won't print anything unless you tell it to. PRINT statements in a program send characters to the television screen; the printer ignores them completely.

There are several ways of getting information out to the printer. The program opposite enables you to see all of them at work, offering you a choice of the three printer commands. The commands are designed to allow you to use the printer selectively. You can either produce a printed version of what the program will PRINT, or produce a printed listing, or see a copy of the screen display produced on paper.

First of all, the printer equivalent of PRINT is LPRINT. It is used just like PRINT but with LPRINT nothing appears on the screen. If you ENTER 1 with the program, the results of a program routine are directed to the printer instead. In this case, the printer will produce a copy of the Spectrum's character set.

The printer's equivalent of LIST is LLIST, selected in the program by ENTERing 2. Again, this only works on the print-out, and will not produce a listing on the screen.

The final method of using the printer is controlled by the COPY command, selected in the program by ENTERing 3. This makes whatever is displayed on the screen appear on the print-out. This is the command used to copy graphics displays, charts and so on.

PRINT-OUT PROGRAM

```
 10 PRINT AT 4,8;"Enter Choice"
 20 PRINT AT 6,8;"1  To LPRINT"
 30 PRINT AT 8,8;"2  To LLIST"
 40 PRINT AT 10,8;"3  To COPY"
 50 INPUT a
 60 GO TO 100*a
100 LPRINT "This will print the
Spectrum character set";
110 FOR f=32 TO 255
120 LPRINT CHR$ f;
130 NEXT f
140 STOP
200 LLIST
210 STOP
300 CLS
310 PLOT 93,15
320 DRAW 160,88
330 DRAW -176,0
340 DRAW 160,-88
350 DRAW -72,136
360 DRAW -72,-136
370 COPY

0 OK, 0:1
```

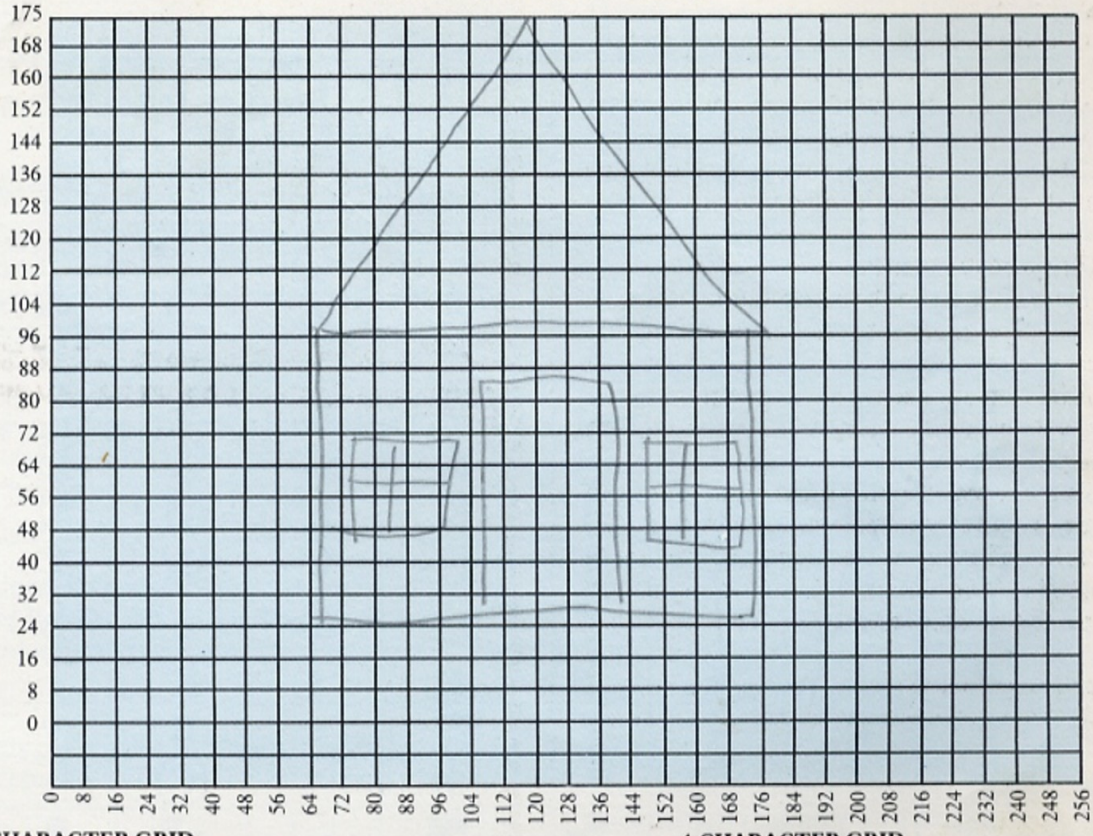
```
Enter Choice
1  To LPRINT
2  To LLIST
3  To COPY
```

The ZX Printer The printer used with the Spectrum is controlled by three keywords that can be used in programs.

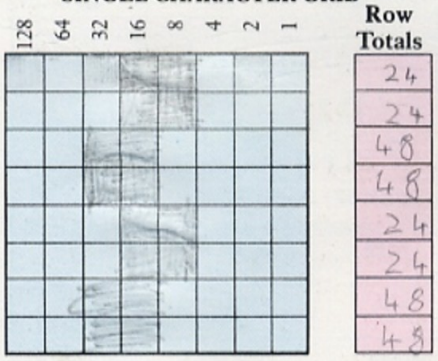
GRAPHICS AND CHARACTER GRIDS

The grid below shows the co-ordinates of the screen display when graphics commands are used. A point on the screen is identified by two co-ordinates x,y. The first co-ordinate sets the horizontal position which is measured along from the left-hand side of the screen. The second co-ordinate sets the vertical position measured from the bottom of the screen. A character PRINTed on the screen occupies an area that is 8 graphics units wide and 8 graphic units high. You cannot PRINT on the bottom two lines of the screen.

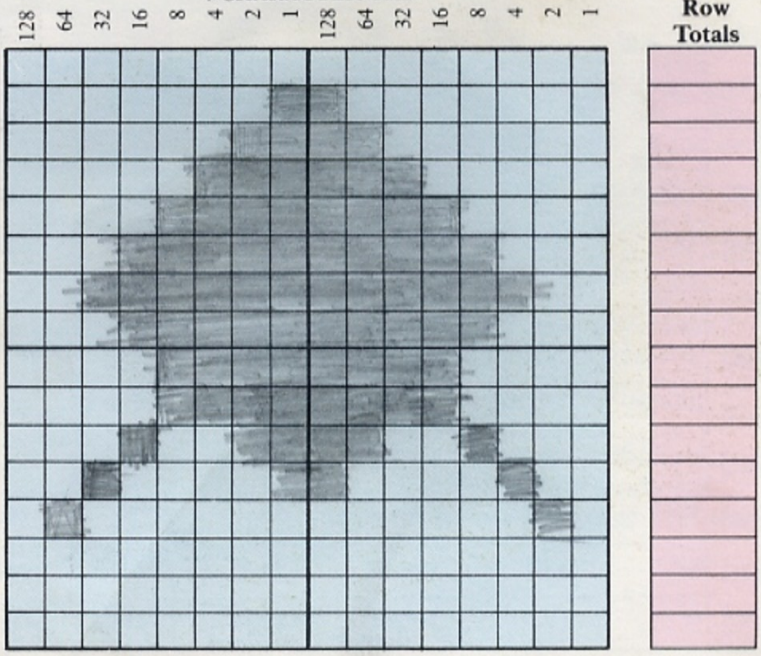
GRAPHICS CO-ORDINATES GRID



SINGLE CHARACTER GRID



4-CHARACTER GRID



Character grids These grids can be used to design either a single character *(above)* or a symbol made from up to four characters *(right)*. You can pencil in your design on the grids and then use the blue columns to list the row totals. These are used in a program with the commands POKE USR. Keys A to U are free for programming user-defined characters.

THE SPECTRUM CHARACTER SET

Each symbol or keyword that the Spectrum uses is represented by a code number. There are 256 code numbers altogether (0–255), each of which can be converted into a single byte of eight binary digits. The letter S for example is specified by CHR$ 83, or CHR$ BIN 1010011. The computer recognizes the binary forms of the code numbers as instructions or information needed to carry out programs. Codes 33 to 126 are allocated to characters specified by the ASCII standard which is used by most microcomputers. The Spectrum's keywords and graphics symbols are specified by the "spare" codes outside this range.

	0	1	2	3	4	5	6	7	8	9
0							PRINT comma	EDIT	cursor left	cursor right
10	cursor down	cursor up	DELETE	ENTER	number		INK control	PAPER control	FLASH control	BRIGHT control
20	INVERSE control	OVER control	AT control	TAB control						
30			space	!	"	#	$	%	&	'
40	(	)	*	+	,	–	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	/	]	↑	_	£	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	©		
130										
140					(a)	(b)	(c)	(d)	(e)	(f)
150	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	(p)
160	(q)	(r)	(s)	(t)	(u)	RND	INKEY$	PI	FN	POINT
170	SCREEN$	ATTR	AT	TAB	VAL$	CODE	VAL	LEN	SIN	COS
180	TAN	ASN	ACS	ATN	LN	EXP	INT	SQR	SGN	ABS
190	PEEK	IN	USR	STR$	CHR$	NOT	BIN	OR	AND	<=
200	>=	<>	LINE	THEN	TO	STEP	DEF FN	CAT	FORMAT	MOVE
210	ERASE	OPEN #	CLOSE #	MERGE	VERIFY	BEEP	CIRCLE	INK	PAPER	FLASH
220	BRIGHT	INVERSE	OVER	OUT	LPRINT	LLIST	STOP	READ	DATA	RESTORE
230	NEW	BORDER	CONTINUE	DIM	REM	FOR	GO TO	GO SUB	INPUT	LOAD
240	LIST	LET	PAUSE	NEXT	POKE	PRINT	PLOT	RUN	SAVE	RANDOMIZE
250	IF	CLS	DRAW	CLEAR	RETURN	COPY				

GLOSSARY

Entries in **bold type** are BASIC keywords.

AND
Allows a program to take a particular course of action only if two conditions are met. An extension of **IF ... THEN**.

Array
A collection of **DATA** organized so that each item is labelled and can be handled separately.

AT
Used with **PRINT** to place characters on the screen.

Bar chart
A type of graph where numerical data is represented by columns.

BASIC
Beginners' All-purpose Symbolic Instruction Code; the most commonly used high-level programming language.

BEEP
Makes the computer sound a short tone or beep, whose duration and pitch are determined by the numbers following the command.

BIN
Converts a number written in binary into the equivalent number in decimal.

Binary
A counting system used by computers based on only two numbers – 0 and 1.

Bisection search
A method of searching **DATA** for one particular item. The **DATA** is continually bisected, or divided into two, until the item in question is found.

Bit
A binary digit – 0 or 1.

BORDER
Changes the colour of the screen's border area.

BRIGHT
Turns specified characters to a brighter shade of their **INK** colour.

Byte
A group of eight bits.

Chip
A single package containing a complete electronic circuit. Also called an integrated circuit (IC).

CHR$
Translates the number following from a character code into the equivalent character.

CIRCLE
Draws a circle of a specified size with its centre at a specified point on the screen.

CLS
Clears the text area of the screen.

COPY
Instructs a printer to print out a copy of the screen display.

COS
The trigonometric cosine function.

CPU
Central Processing Unit. Normally contained in a single chip called a microprocessor, this carries out the computer's arithmetic and controls operations in the rest of the computer.

Cursor
A flashing symbol on the screen, showing where the next character will appear.

DATA
The computer treats whatever follows **DATA** as information that may be needed later in the program. Used in conjunction with **READ**.

Debugging
The process of ridding a program of errors or bugs.

DEF FN
Defines a function used elsewhere in a program by the command **FN**.

DIM
Informs the computer about the dimensions of an array so that the computer knows how many items the array contains.

DRAW
Draws a line in the current **INK** colour from the graphics origin at 0,0 or from the last point visited to a point specified.

FLASH
Makes characters flash on the screen.

Flowchart
A diagrammatic representation of the steps necessary to solve a problem.

FN
Indicates that the variable following is being used as the name of a function. The function must be defined by a **DEF FN** statement.

FOR ... NEXT
A loop which repeats a sequence of program statements a specified number of times.

GOSUB
Makes the program jump to a subroutine beginning at the line number following the command. The subroutine must always be terminated by **RETURN**.

GOTO
Makes a program jump to the line number following the command.

Hardware
The physical machinery of a computer system, as distinct from the programs (software) that make it do useful work.

IF ...THEN
Prompts the computer to take a particular course of action only if the condition specified is detected.

INK
Changes the colour of text and graphics that appear on the screen.

INKEY$
Monitors the keyboard to see if any key has been pressed, and if so returns the character.

INPUT
Instructs the computer to wait for some data from the keyboard which is then used in a program.

INT
Converts a number with decimals into a whole number.

Interface
The hardware and software connection between a computer and another piece of equipment.

INVERSE
Switches the **PAPER** colour for the **INK** colour and vice versa.

K
Abbreviation of kilobyte (1024 bytes).

LEN
Counts the total number of characters in a string that follows it.

LET
Assigns a value to a variable.

LIST
Makes the computer display the program currently in its memory.

LLIST
Instructs a printer to print out a listing of the program currently in memory.

LOAD
Transfers a program from a cassette tape into the computer's memory.

Loop
A sequence of program statements which is executed repeatedly or until a specified condition is satisfied.

LPRINT
Sends whatever follows the command to a printer instead of to the screen, so a paper copy only is produced.

MERGE
Allows a second program to be **LOAD**ed into the computer from a tape cassette without erasing the program currently in memory, as long as the line numbers do not coincide.

NEW
Removes a program from the computer's memory.

OR
Allows a program to take a particular course of action if either of two specified conditions are met. An extension of **IF ... THEN**.

OVER
Allows new characters to be **PRINT**ed on top of existing characters.

PAPER
Changes the screen's background colour.

PAUSE
Halts a program for a period set by a number measured in fiftieths of a second.

PEEK
Reports the number stored in a specified location in the memory.

Peripheral
An extra piece of equipment which can be added to the basic computer system – a printer, for example.

Pie chart
A graphic display of numerical data in the form of a divided circle. The size of each slice in the pie reflects the size of the number it represents.

PLOT
Makes a dot appear on the screen at the point specified.

POINT
Reports whether the point at the co-ordinates following is displayed as an **INK** colour or **PAPER** colour.

POKE USR
Stores a number that reprograms a key to produce a user-defined character. On its own, **POKE** puts a number into a specified memory location.

PRINT
Makes whatever follows appear on the screen.

RAM
Random Access Memory (volatile memory). A memory whose contents are erased when the power is switched off. (See also ROM.)

RANDOMIZE
Sets the point at which the fixed **RND** sequence will begin.

READ
Instructs the computer to take information from a **DATA** statement.

REM
Enables the programmer to add remarks to a program. The computer ignores whatever follows the commands.

RESTORE
Resets the point from which **DATA** items are **READ**, so that items can be used more than once in a program.

RETURN
Terminates a subroutine. (See also **GOSUB**.)

RND
Produces numbers between 0 and 1 at random.

ROM
Read Only Memory (non-volatile memory). A memory which is programmed permanently by the manufacturer and whose contents can only be read by the user's computer.

SAVE
Records a program currently in the computer's memory onto a tape. The program is identified by a filename.

SCREEN$
Holds the details of a screen display so that they can be **SAVE**d or **LOAD**ed.

SIN
The trigonometric sine function.

Software
Computer programs.

SQR
Produces the square root of the number that follows.

Statement
An instruction in a program. There may be more than one statement in each program line.

STEP
Sets the step size in a **FOR ... NEXT** loop.

STOP
Halts a program and **PRINT**s out the line number in which it appears.

String
A sequence of characters treated as a single item – someone's name, for instance.

Subroutine
A part of a program that can be called when necessary, to produce a particular display or carry out a number of calculations repeatedly, for example.

Syntax
Rules governing the way statements must be put together in computer language.

TAB
Used with **PRINT** to specify how far along a line characters are to appear.

VAL
Evaluates a number-string, and produces a number.

Variable
A labelled slot in the computer's memory in which information can be stored and retrieved later in a program.

VERIFY
Checks that a program that is currently in memory has been recorded correctly on a tape cassette using **SAVE**.

INDEX

Main entries are in **bold type**

DK

ScreenShot

PROGRAMMING SERIES

The original and exciting new teach-yourself programming course for ZX Spectrum owners.

Over 150 unique screen-shot photographs of program listings *and* programs in action – showing on the page exactly what appears on the screen.

Packed full of programming tips and techniques, reference charts and tables, and advice on how to get the most out of your ZX Spectrum.

CONTENTS INCLUDE

Question-and-answer conversations • Multiple-choice displays • 'Natural' graphics • Sorting and fact-finding • Curves and circles • Pies and slices • Bars and graphs • A guide to writing games • Multiple characters and animation techniques

Further volumes in the ScreenShot series include

Step-by-Step Programming for the **ZX Spectrum** BOOK ONE

PLUS

Step-by-Step Programming for the **BBC Micro, Commodore 64, Acorn Electron,** and **Apple II**

DORLING KINDERSLEY

ISBN 0-86318-031-0

00595

£5.95

9 780863 180316