Programming your ZX SPECTRUM

Tim Hartnell and Dilwyn Jones

To Richard + Nicholas
Love from
Mummy + Daddy
20 June, 1987

Programming your ZX SPECTRUM

Tim Hartnell and Dilwyn Jones

Foreword:

Your first hours with your ZX Spectrum can be bewildering. Once you've run through the sample programs given in the manual, you're likely to think: "Yes, but now what?" This book is intended to answer that question. It will take you through programming the Spectrum from first principles, right up to quite sophisticated programming techniques.

And while you're entering the programs, zapping aliens and asteroids all over the place, you'll discover that you're actually learning a lot about programming, and about computers in general — all without any effort at all.

This book is intended to be a tool, to be worked through with your computer turned on beside you. Its value will be greatly diminished if you simply try to read through the programs. It'd be best if you enter each program as you come to it, leaving no alien unzapped. That way, you'll gain the maximum benefit from the book, and you'll be a programming whiz before you know where you are.

Time to get underway. Plug in your Spectrum, turn on your TV, and let's get going.

Tim Hartnell, London
Dilwyn Jones, Bangor, Gwynedd

First published in the UK by:
Interface Publications,
9-11 Kensington High Street,
London W8 5NP.

First printing July 1982.
Second printing, amended version Nov. 1982.
Third printing July 1984.
ISBN 0 907563 19 8

Books published by Interface Publications are distributed in the UK by WHS Distributors, St John's House, East Street, Leicester LE1 6NE (0533 – 551196) and in Australia and New Zealand by PITMAN PUBLISHING. Any queries regarding the contents of this volume should be directed by mail to Interface Publications, 9-11 Kensington High Street, London W8 5NP.

This book is dedicated to Iris

Typeset and Printed in England by Commercial Colour Press, London E7.

Using the keyboard

Although, at first sight, the Spectrum keyboard can look bewildering, it is not very difficult to master, so long as you proceed carefully in the early stages.

The two most important keys are the shift keys CAPS SHIFT (bottom left-hand corner) and SYMBOL SHIFT (second key from the right-hand end of the bottom row of keys). Find them now.

The colours of these keys indicate one of their uses. Turn on your Spectrum, hold down the CAPS SHIFT key then press any of the keys with letters of the alphabet on them. You'll see that you get the capital letter version of that key. CAPS SHIFT also triggers the words in white above keys 1 to 4. Hold down CAPS SHIFT, then press the 2 key. Now, press any of the alphabet keys, and you'll see they come out as capitals. The use of TRUE VIDEO and INVERSE VIDEO is discussed in the section on graphics.

Moving away from the white SHIFT key, let's look now at the red one, SYMBOL SHIFT. Hold it down, and press any key at all (except ENTER, BREAK SPACE or CAPS SHIFT). You'll see you get the little red symbols (like the upward arrow on the H key) from the key.

Next, press any alphabet key, without holding down a SHIFT key. You'll see you get the white word (IF, NEXT, DIM and the like). These are *Keywords* and are the first words in a line of program, which we will discuss shortly.

You get the green words above the keys by pressing down both SHIFT keys at once, then letting them go and touching the key. For example, hold down both SHIFT keys, let them go, and touch the D key. The word DATA (the word above the key) will appear.

The red words below each key are obtained by pressing both SHIFT keys at once, then letting go of the white one, but continuing to press on the red one, touch a key. This will get

you the word BRIGHT from the B key, ATTR from the L key, and VERIFY from the R key.

This covers everything except for EDIT, ENTER, GRAPHICS and DELETE.

EDIT — You use this to modify a line of program. Moving the cursor (a greater than, > , sign) then pressing the 1/EDIT key while holding down the CAPS SHIFT key, will bring the line to be edited to the bottom of the screen. The 5 and 8 keys will then move you along the line in the direction of the arrow heads shown above those keys.

ENTER — You press this key after typing a line of program at the bottom of the screen, to get it to move up into the main body of program at the top of the screen. It is also used after a word like RUN has been entered, to get the computer to execute the command.

GRAPHICS — If you hold down the CAPS SHIFT, then press the 9 key, you will see the cursor turn into a G. Now, press the number keys, and see what appears on the screen. Hold down CAPS SHIFT, and press them again, and you'll see you get the 'opposite' of the graphic you got without using CAPS SHIFT. The second use of the GRAPHICS mode, for user-defining keys, is described in a later section in the book.

DELETE — This is, as its name implies, a 'rub out' key. Hold down the CAPS SHIFT key, then press the **0**, and the line you are working on will be rubbed out, element by element. You can take your finger off the CAPS SHIFT, once the DELETE is auto-repeating, and it will continue to rub out for you.

Do not worry if this description — which we've made as simple and clear as we can — does not immediately open up the mysteries of the keyboard to you. Using your Spectrum, and finding each thing on the keyboard as you need it, will eventually lead you to the point where using the keyboard will become second nature.

The PRINT statement

PRINT is probably the most-used command in BASIC. It is the command which allows the computer to communicate with you. Type the following line into your Spectrum, and then press ENTER:

PRINT 5

You'll see that the computer obediently prints the number five. You can use the PRINT command to make your computer act as a calculator. Enter the following, and then press ENTER:

PRINT 5−3

When you press ENTER, you'll see it prints up the correct result. This 'direct calculation mode' can work out problems as complex as you wish. Try the following, remembering to press ENTER after you've done so to make the computer act on what you've typed in:

PRINT SQR (8 + 1)

This asks the computer to PRINT the square root (that's what SQR means) of the sum of the numbers in brackets, that is, the square root of nine. If your computer is functioning correctly, you should — of course — have got an answer of three.

So you can see that PRINT can be used in the direct mode to print out numbers, and the results of calculations. It can also print out words. Engage CAPS LOCK by holding down the white 'CAPS SHIFT' key, then pressing the 2 key. This will make the computer print in capital letters. Try the following, then press ENTER:

PRINT HI THERE

Instead of happily printing HI THERE, the computer comes up with what is called an error message. In this case, the error message reads "2 variable not found". If you want the computer to print out words, the words must be enclosed

within quote marks. Enter and run (that is, press ENTER after typing it in) the following:

```
PRINT "HI THERE"
```

You'll see the words HI THERE appear at the top of the screen.

To recap quickly. Simply used as a command, typing PRINT 2+3 will tell the computer to print out the result of that addition. Entering PRINT "WORDS" will get the computer to print out everything which is within the quote marks.

Computers use programs, and it is now time to write our first, simple program. Enter and run this program. When you RUN this, which you do by pressing the R key, then pressing ENTER, you should see a print out similar to that which is below the program listing.

```
  10 REM PROGRAM ONE
  20 PRINT "THIS IS A DEMONSTRAT
ION"
  30 PRINT 1
  40 PRINT 2
  50 PRINT "THIS IS THE END"

THIS IS A DEMONSTRATION
1
2
THIS IS THE END
```

While we have this program in the computer, let's learn a little more about programs. Enter the word LIST (which you do by pressing the K key), then press ENTER. You'll see the program listing comes back. Notice that every line starts with a line number. The first line, in this case numbered 10, starts with the word REM. REM is computer talk for 'remark', and is used in a program when you want to explain what is going on within that program, or what a program is (as in this case), so that when you return to it later, you'll know what is going on. The computer ignores REM statements when it comes to them.

A REM statement is made up of a line number, then the word REM, and some text. The message which follows the word REM can be made up from anything you like — letters, numbers, punctuation marks, graphics or spaces — although it is best to keep the messages as brief and clear as you can. Although anything typed after the word REM is ignored by the computer when it is running a program, a REM line still uses up memory.

REM statements are often like the following:

```
10  REM THIS WORKS OUT THE SCORE
10  REM FIND THE ANGLE
```

There is no reason why there should be just one REM statement, but if the commentary you wish to add to a particular line of a program is one which may take up more than one line of text, it is important to place the word REM at the beginning of each new line. For example:

```
60  REM THE MULTIPLICATION ROUTINE IN
    WHICH
70  REM THE TWO VARIABLES A AND B
80  REM ARE MULTIPLIED TOGETHER
```

So long as each REMark line starts with the word REM, the computer will ignore the text that follows on that line (although the complete program listing, REMs and all, will be printed on the screen if a LIST is requested).

Now, let's have a look at editing. Type in the number 10, then press ENTER. Line 10 has disappeared. It is very easy to get rid of lines you don't want in a computer, just by typing in the relevant line number, then pressing ENTER.

```
  20 PRINT "THIS IS A DEMONSTRAT
ION"
  30 PRINT 1
  40 PRINT 2
  50 PRINT "THIS IS THE END"
```

Add 10 REM, then press ENTER.

You'll recall, from the times you've pressed LIST while working through this section, that LIST is the BASIC command which we use to get the computer to print out the whole of the program it is currently holding. All the lines in the program are LISTed in numerical order, rather than the order in which they were entered into the computer. That is, the computer automatically sorts its lines into order. Enter the following, and then press ENTER.

```
15 PRINT "THIS IS A NEWLINE"
```

You'll see, in the next program, that the new line (15) automatically moves into its correct position within the listing.

```
 10 REM
 15 PRINT "THIS IS A NEWLINE"
 20 PRINT "THIS IS A DEMONSTRAT
ION"
 30 PRINT 1
 40 PRINT 2
 50 PRINT "THIS IS THE END"
```

As you've no doubt realised, the RUN command is used to start the computer operating on a program which you have entered into the computer, either by typing it in, or by loading a program in from cassette. The computer executes all the lines stored in its memory, starting from the lowest number, and working through in order. Various commands can make the computer loop back on itself, but in essence, the computer works through a program in line number order, unless told to do otherwise.

If you want the program to stop at a particular point, you can use — naturally enough — a command called STOP. Enter 25 STOP (from the A key, after holding down the red SHIFT), then press ENTER, then run the program. It will print out:

```
THIS IS A NEWLINE
THIS IS A DEMONSTRATION
```

Then, at the bottom of the screen, will be the message 9 STOP statement, 25:1 which means a STOP was executed, the first command in line 25.

We'll return to look at PRINT in a little more detail in a moment, but there is one more command I'd like to introduce at this moment. The command NEW will erase any program from the computer's memory, and should always be used to remove anything from the memory before you start writing a new program. If you don't do this, and you use different line numbers for the second program, you'll find the lines may well be interwoven with the lines from the old program. The NEW command is brutal, and final, causing the computer to dramatically forget everything you had typed in, or loaded in from tape.

Try it now on your computer. Type in NEW (from the A key), press ENTER, then press LIST and press ENTER again. You'll find, not unexpectedly, that no listing appears. Try LIST 10, and you'll get the same nothing result.

PRINT formatting and TAB

To continue our exploration of the PRINT command, engage CAPS LOCK as before, then enter and run the next program.

```
 10 REM PRINT FORMATS
 20 PRINT
 30 PRINT
 40 PRINT
 50 PRINT
 60 PRINT "HI ";60
 75 PRINT "HI ";70
 80 PRINT 12
 90 PRINT 1,2
100 PRINT ,1
110 PRINT ,1,2
```

Follow this explanation carefully, and you should learn a lot about the way the computer formats its print output. You can then use what you've learned to arrange output of your own programs as you wish. I'll go through the program line by line:

10 title, REM statement

20–50 Each of these words PRINT, with nothing following, prints a blank line, moving the next print position down a line. This explains the gap at the top of the screen, when you run the program, before anything is printed.

60 This prints the word HI and then, leaving a space, prints the number 60, so you know which line it comes from.

70 The comma (red SHIFT the comma, near the bottom right hand corner of the keyboard), as you can see, moves the start of the line halfway across the screen.

80 This allows the numbers 1 and 2 to be printed close together. Note that even if there is a space between the numbers in the program (as in PRINT 1 2), the computer will still print them as 12.

90 This line uses commas between the numbers to ensure that they will be printed in separate halves of the screen.

100 The comma at the beginning of the line moves the 1 halfway across the screen, just as the word HI was moved in line 70.

110 The semicolon between the numbers ensures that they are printed hard up against each other, just as they were in line 80.

You can use the comma and semicolon within PRINT statements to control the output to produce the screen display you need. Clear the program with NEW, then enter and run the next series of programs, to produce a number of effects.

The first program, called PRINT TWO, simply prints the numbers one to 10 down the side of the screen. The next one (PRINT TWO–B) prints them hard up against each other. PRINT TWO–C prints them in neat little columns, and PRINT TWO–D prints out the numbers, again from one to 10, with a singles space between them.


```
10 REM print two
20 FOR j=1 TO 10
30 PRINT j
40 NEXT j
```

```
10 REM print two - b
20 FOR j=1 TO 10
30 PRINT j;
40 NEXT j
```

```
10 REM print two - c
20 FOR j=1 TO 10
30 PRINT j,
40 NEXT j
```

```
10 REM print two - d
20 FOR j=1 TO 10
30 PRINT j;" ";
40 NEXT j
```

The use of TAB

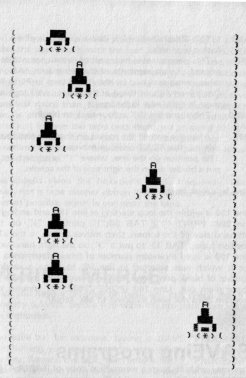
TAB (for tabulate) is a command which can usefully be combined with PRINT. It moves the PRINT position across the number of spaces specified following the number. Enter programs PRINT TWO–E and PRINT TWO–F and see the effect of the TAB command in these.

```
10 REM print two - e
20 FOR j=1 TO 10
30 PRINT TAB j;j
40 NEXT j
```

```
10 REM print two - f
20 FOR j=1 TO 10
30 PRINT TAB 3*j;j
40 NEXT j
```

The next program TABULATOR ROCKET RANGE, shows how effectively the TAB command can be used. Enter, and RUN it, then return to this book for a discussion on the important lines within it.

```
 10 REM Tabulator rocket range
 20 DIM a$(5,5)
 25 BORDER 0
 30 FOR j=10 TO 1 STEP -1
 40 PRINT TAB 3*j; INK 2;j
 50 FOR a=1 TO j
 52 BEEP .02,5*j
 60 NEXT a
 70 NEXT j
 71 FOR b=1 TO 5
 72 READ q$
 73 LET a$(b)=q$
 74 NEXT b
 80 REM **main program**
 90 LET q=INT (RND*25)+1
100 FOR r=1 TO 5
110 BEEP .02,10*r
120 PRINT "(";TAB q; INK r;a$(r
); INK 0;TAB 30;")"
130 NEXT r
140 POKE 23692,-1
150 LET space=q/3
160 FOR p=1 TO space
190 PRINT " (";TAB 30;")"
200 NEXT p
210 GO TO 90
220 DATA "  A","  ■","  ■■","  ■
■■","(*)("
```

The most useful lines for this discussion are 120 and 190, as these make use of TAB in printing. Line 120 behaves as follows:

"(" This prints a left-hand bracket, hard up against the left-hand edge of the screen.

TAB q q is a number between one and 25 (chosen in line 90) which determines how many spaces across the PRINT position will move.

INK r — This determines which colour each pair of the rocket is printed.

A$(r) — This determines which part of the rocket will be printed. It uses elements of the string array, A$, which were assigned by the READ loop in 71 to 74. Don't worry about these at this stage, we'll look at them in detail later in the book.

INK 0 — This turns the INK colour back to black

TAB (30) — After the part of the rocket on that line has been printed, the PRINT position moves across to the 31st position on the line, where "|" is printed, to put a border down the right-side of the screen.

Now let's look at line 190:

Line 190 is within the loop starting at line 180 and ending at line 200. PRINT "|"; TAB 30;"|" prints a "|" on the left-hand side of the screen, then moves across to the 31st position (using TAB 30) to put a "|" on the right-hand side. Line 190 is used a random number of times (determined by the q which was selected in line 90) to place a random number of blank print lines between successive 'rockets' to space them out. Line 140 (POKE 23692,−1) ensures the program does not keep stopping to ask 'scroll?'.

SAVEing programs

You may wish to keep a permanent copy of TABULATOR ROCKET RANGE. You can SAVE programs by typing in the program, connecting up your cassette recorder as shown in the manual, then typing in SAVE followed by the name of the program within quote marks. In this case, I suggest you use the name ROCKET, so you would type in SAVE "ROCKET": Turn your cassette recorder on to record, after connecting it up as shown in the manual, and then press the ENTER key.

We suggest you make a habit of saving each program three times in a row on C−12 or C−15 (i.e. computer) cassette,

and that you only put one program on each side of a tape. Label the tape clearly with the loan name (i.e. with ROCKET in this case). Although it may seem wasteful to use up the whole side of a cassette with just one program, recorded three times, the frustration you will save yourself by not having to search through tape after tape for a program you want will more than compensate for using more cassettes than is strictly necessary. The program is recorded three times just in case the tape gets damaged at some point, or you accidentally erase part of the program, or — as sometimes happens — one recording of the program refuses to load properly.

You should clean the recorder's heads frequently, using liquid, not a tape cleaner ribbon in a cassette, to ensure the clearest possible signal is put onto the tape.

VERIFY, MERGE

Once you have a program on tape, you can check that it is correctly SAVEd, before you wipe the program from within the computer.

You save by, for example, typing in SAVE "PROGRAM", then turning your tape recorder onto record, then pressing any key. Rewind the tape, then enter VERIFY "PROGRAM", press ENTER and rerun the tape. If all is well, the name of the program will appear, and the 0 OK message will appear on the screen. You know then that the program has been successfully SAVEd.

It is possible to load a new program into the computer while an old one is still there. The two programs will combine. If the two programs have any identical line numbers, the line from the newest program will overide that of the old. Here's an example. We entered this program, SAVEd it, then NEWed the computer.

```
10 REM TEST PROGRAM ONE
20 REM LINE 20
30 REM LINE 30
40 REM LINE 40
```

Then, this program was written. MERGE "ONE" was typed in ("ONE" was the name under which we have saved the first program) and then the recorder was played.

```
 5 REM PROGRAM TWO
15 REM LINE 15
25 REM LINE 25
35 REM LINE 35
```

Within a few seconds, we had this program in the computer.

```
 5 REM PROGRAM TWO
10 REM TEST PROGRAM ONE
15 REM LINE 15
20 REM LINE 20
25 REM LINE 25
30 REM LINE 30
35 REM LINE 35
40 REM LINE 40
```

MERGE is a very useful way to take advantage of special routines, such as RENUMBER, which you can load in after writing another program if you like. It is worth ensuring that your routines have high line numbers (say above **9000**) and the lower ones are used for the program. This ensures you will not have problems with overwritten lines.

Getting programs back into the computer can prove difficult at times. If you have LOAD problems then try the following tips:

(i) Disconnect the lead not in use from both the computer and the cassette recorder.
(ii) Try operating the cassette recorder from batteries.
(iii) Try moving the computer and the cassette recorder further apart, as well as the T.V. if you can.
(iv) Change the volume setting on the cassette recorder since some cassettes may have a higher output than others. Try changing the tone control settings, in particular turn up the treble or turn down the bass.
(v) Make sure your leads have not broken or cracked, or a solder joint come loose.
(vi) This sounds silly, but make sure your plugs are in the correct hole! You may find it useful to stick labels on top of the computer above the sockets to tell you which one is which so that you don't have to peer round the back to look every time.

Now, let's return to TAB

We can only use TAB with a single number after the word. Remember, TAB A will move the start of the PRINT position A+1 spaces across a line. You can have the word PRINT followed by AT, and two numbers, such as PRINT AT **10**, **6**; which will move the PRINT position seven spaces across, and 11 down. The top left-hand corner of the screen is zero, zero, so PRINT AT **0**, **0**; indicates that the printing will begin in the top left-hand corner of the screen. The left-hand side of the screen is numbered **0**, while the right-hand side is 31. The screen is 32 characters wide, so the position furthest to the right is numbered 31.

PRINT AT

The following program, SQUASH, uses PRINT AT Y, X to position a ball (line 620) and a bat (line 150). You use the "Z" and "M" keys to move the slide (the bat) at the bottom of the screen right and left respectively. The program keeps track of how long you keep the ball in flight, and gives you a score at the end based on this time. Pressing any key at the end of the game will give you a new game.

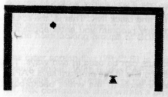
This listing may well look pretty horrifying at the moment. Once you've finished working through this book, you may wish to come back to programs like this, and try and work out

what each section of the program is doing. You'll be surprised to see how much of it you can decipher.

```
 10 REM SQUASH
 15 REM AFTER PROGRAM BY
       JEREMY RUSTON
 20 GO SUB 690
 25 REM MOVE BAT WITH Z AND M
          KEYS
 27 LET MOVEBALL=550
 30 LET SETUP=300
 35 LET MOVEBAT=460
 40 BRIGHT 1: BORDER 4
 45 PAPER 7: CLS
 50 LET SCORE=0
 60 GO SUB SETUP
 70 REM ************
 80 LET SCORE=SCORE+INCREMENT
 110 LET A$=INKEY$
 130 IF A$="Z" OR A$="M" THEN GO
SUB MOVEBAT
 140 GO SUB MOVEBALL
 150 PRINT AT 19,B+11; INK 2;B$
 160 GO TO 80
 290 REM ***********
 300 REM ** SET UP **
 310 LET X=1
 320 PRINT AT 10,10; INK 1;"[redacted]
[redacted]"
 330 FOR T=0 TO 10
 340 PRINT AT T+10,10; INK 1;"■"
AT T+10,30;"■"
 350 NEXT T
 360 LET B$=" "+B$+" "
 380 LET Y=1
 385 LET L=1
 390 LET A=1
 400 LET B=10
 410 PRINT AT 19,11+B;B$
 420 LET INCREMENT=2*7+INT (RND*
100)
 430 RETURN
 450 REM ******************
 460 REM ** MOVE BAT **
 480 IF A$="M" AND B=16 THEN RET
URN
 490 IF A$="Z" AND B=0 THEN RETU
RN
 510 IF A$="M" THEN LET B=B+1
 520 IF A$="Z" THEN LET B=B-1
 530 RETURN
 540 REM ******************
 550 REM *** MOVE BALL ***
 570 PRINT AT 11+Y,11+X;" "
 580 IF L+X>18 OR L+X<0 THEN LET
L=-L: BEEP .01,50
```



```
 590 IF M+Y>8 OR M+Y<0 THEN LET
M=-M: BEEP .01,20
 600 LET X=X+L
 610 LET Y=Y+M
 620 PRINT AT 11+Y,11+X; INK 4;D
$
 622 IF Y<>8 THEN RETURN
 625 PRINT AT 6,7; INK 6; PAPER
2;" SCORE IS "+STR$ SCORE+" "
 630 IF Y=8 AND ABS (B-X)<=2 THE
N RETURN
 640 PRINT AT 2,2; INK 7; PAPER
3;" END OF GAME "
 650 FOR G=1 TO 50
 660 BEEP .01,G: BEEP .01,60-G
 670 NEXT G
 680 RUN
 690 REM LETTER IN QUOTE MARKS
        IN LINE 700 IS AN M,
        ENTERED AFTER GETTING
        INTO GRAPHICS MODE
 700 LET D$="◆"
 710 FOR H=0 TO 7
 715 BEEP .01,H*H
 720 READ N
 730 POKE USR "M"+H,N
 740 NEXT H
 750 REM NEXT LETTER IS
        GRAPHICS B
 760 LET B$="▲"
 770 FOR H=0 TO 7
 775 BEEP .01,50/(H+1)
 780 READ N
 790 POKE USR "B"+H,N
 800 NEXT H
 890 RETURN
1000 DATA BIN 00000000,BIN 00011
000,BIN 00111100,BIN 00111110,B
IN 00111110,BIN 00111110,BIN 0
0111100,BIN 00001000
1010 DATA BIN 11111111,BIN 00111
100,BIN 00011000,BIN 00111100,BI
N 01111100,BIN 01111110,BIN 01111
110,BIN 11111111

SCORE IS 5152
```



Colours and graphics

The Spectrum is equipped with powerful graphics commands which you can use to greatly enhance your programs. The commands are simple to use, and capable of producing a wide range of effects.

There are three things you can control with the colour command: the border, around the main display area (accessed by the command BORDER), the main display area itself (known as the PAPER) and the colour in which printing is carried out (the INK).

There are eight colours available (if you count black and white as colours) and these are numbered from zero to seven. The colours, and their corresponding numbers, are:

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

The lower the number, the darker the colour. On a black and white set, the lower numbers are closer to black, the higher numbers to white.

When you first turn your Spectrum on, you'll have white PAPER, a white BORDER and black INK. That is, the screen is completely white, and any program you enter appears in black.

The INK and PAPER colours can be used in two ways. The first is 'globally'. That is, if a line in the program says PAPER 6, followed by CLS (clear screen), the entire background

(that is, the area within the border) will turn yellow. Similarly, the program line INK 2 will ensure that all printing from that point on appears in RED.

The colours can also be used 'locally'. If you enter PRINT INK 1; PAPER 7; "HI THERE", the Spectrum will print the words HI THERE in white on a little blue strip. The same local control is possible within INPUT statements. If you want a string input, you could enter INPUT (INK 2; PAPER 6; "What is your name"); a$ and the question would be printed in red on a little yellow strip.

Let's try out the colour commands now, by entering the following program:

```
 20 REM COLOUR DEMONSTRATION
 30 FOR B=0 TO 7
 40 FOR P=0 TO 7
 50 FOR I=0 TO 7
 60 BORDER B
 70 PAPER P: CLS
 80 INK I
 90 PRINT AT 10,10;"BORDER ";B;
TAB 10;"PAPER ";P;TAB 10;"INK ";
I
100 FOR U=1 TO 60: NEXT U: BEEP
.15,P*I+2*B
110 NEXT I
120 NEXT P
130 NEXT B
```

This goes through all the combinations of BORDER, PAPER and INK. As you'll see, it takes quite a long time to run (there are 8*8*8 possible combinations), although several of the possible combinations are not very effective (white INK on white PAPER with a white BORDER is not particularly easy to read!) Certain other groups are just unattractive. Other groups of colour which you'll see as the program runs are, however, very effective indeed, and it is worth keeping a pen and paper nearby when running this program to take a note of the best-looking groups of BORDER, INK and PAPER.

The clear screen line (70 CLS) is needed to make the paper colour 'global'. With it, the PAPER only changes underneath the words being printed. Try the program again, without line 46. INK commands used within a program and automatically

global if on a separate line followed by CLS, are automatically local if coupled directly with a PRINT or INPUT statement. A global INK command (such as INK 2 to get all red printing) is not changed by a local INK command (such as PRINT INK 1; "test") as the INK colour reverts to the one which was globally defined as soon as a PRINT statement without an INK parameter appears in the program.

Run the next program now, which shows how effectively the colours can mix when they are chosen randomly.

```
  6 REM Pyramid
  7 REM @ Hughes, Hartnell
 10 BORDER 7
 15 CLS
 20 LET b=16
 30 LET t=0
 60 LET s=0
 70 LET l=20
 80 LET t=t+b
 90 FOR n=s TO s+b*2-2
100 PRINT AT l,n; INK INT (RND*
6)+1;"■"
105 BORDER INT (RND*6)+1
110 NEXT n
120 LET l=l-1
130 LET b=b-1
140 LET s=s+1
150 IF b>0 THEN GO TO 80
155 BORDER 1
160 GO TO 160
```

The program draws a pyramid of little coloured blocks. The BORDER flashes alarmingly all through the program, and finally (lined 155) turns blue. Line 160, which just calls itself, is designed to suppress the 'OK' report code which would otherwise spoil the display. You get out of the program by pressing BREAK.

The little black square at the end of line 100 is available directly from the keyboard by getting into the GRAPHICS mode (white SHIFT key, then press key 9) and then pressing the 8 key, still holding down the white SHIFT key. Inverses of other characters are available by simply pressing the INV. VIDEO (white SHIFT key, then the 4 key). You revert to what is called TRUE VIDEO, by pressing the white SHIFT key, and the 3 key. The black background behind inverse letters turns into the INK colour, and the letters themselves turn into the

PAPER colour, which can look most effective, as the next program demonstrates:

```
 15 PAPER 5
 17 CLS
 20 FOR g=1 TO 100
 30 INK RND*7
 40 PAPER RND*7
 50 BORDER RND*7
 70 PRINT AT RND*21,RND*9;"THIS
 IS A DEMONSTRATION"
 80 NEXT g
```

Using the program colours directly in a program can produce good results as this program — COLOUR CODE — shows. This is a variation of Mastermind but, as you'll see by running it, the program expects you to guess a code of four colours, not four numbers or letters as in most computer versions of the game. Enter and run the game, then return to the book for an explanation of the colour and graphics commands which are used in it.

```
 10 REM COLOUR CODE
 20 POKE 23609,100
 30 DIM C(4)
 40 DIM G(4)
 50 DIM H(4)
 60 INK 0: BORDER 0: PAPER 7: C
LS
 70 PRINT '''TAB 3;"I AM THINKI
NG OF A 4-COLOUR"
 80 PRINT '"CODE. YOU HAVE 10 G
UES TO GUESS"
 90 PRINT '"IT. I CHOOSE FROM T
HESE COLOURS";''
100 FOR C=1 TO 6
110 PRINT TAB 4+C; INK 0;C;;" >
";" INK C;"■"
120 NEXT C
130 PRINT '''" ALL 4 COLOURS ARE
DIFFERENT."
140 PRINT '''"   PRESS ANY KEY TO
BEGIN..."
150 PAUSE 4E4
160 CLS
170 PRINT AT 1,5;
180 FOR C=1 TO 6
190 PRINT INK 0;C;">"; INK C;"■
■";
200 NEXT C
210 PRINT
220 LET C(1)=INT (RND*6)+1
230 LET Z=1
```

```
240 LET Z=Z+1
250 LET C(Z)=INT (RND*6)+1
260 LET J=0
270 LET J=J+1
280 IF C(J)=C(Z) THEN GO TO 230
290 IF J<Z-1 THEN GO TO 270
300 IF Z<4 THEN GO TO 240
310 FOR D=1 TO 10: POKE 23692,-
1
320 PRINT INK 0;"ENTER GUESS NU
MBER "\D
330 INPUT A
335 FOR D=1 TO 32: PRINT CHR$ 8
; NEXT D
340 PRINT "
"
350 FOR Z=1 TO 4
360 LET G(Z)=A-10*INT (A/10)
370 LET H(Z)=G(Z)
380 LET A=INT (A/10)
390 NEXT Z
400 LET B=0: LET W=0
410 FOR Z=1 TO 4
420 IF C(Z)<>G(Z) THEN GO TO 45
0
430 LET B=B+1: BEEP .2,B*15
440 LET G(Z)=0
450 NEXT Z
460 FOR Z=1 TO 4
470 IF G(Z)=0 THEN GO TO 520
480 FOR W=1 TO 4
490 IF C(Z)<>G(W) THEN GO TO 51
0
500 LET W=W+1: BEEP .2,60-B*15
510 NEXT W
520 NEXT Z
530 FOR T=4 TO 1 STEP -1
540 PRINT INK H(T);"■ ";
550 NEXT T
560 PRINT INK 0;"";B;" BLACK";
570 IF B<>1 THEN PRINT "S";
580 PRINT " AND ";W;" WHITE";
590 IF W<>1 THEN PRINT "S"
600 IF W=1 THEN PRINT
610 IF B=4 THEN PRINT "YOU GOT
IT. IN ";D;" GUESS";
620 IF B>1 AND B=4 THEN PRINT "
ES"
630 IF B<>4 THEN NEXT D
640 PRINT '''"THE CODE WAS ";
650 FOR H=1 TO 100: NEXT H
660 FOR T=4 TO 1 STEP -1
670 FOR H=1 TO 50: NEXT H
680 BEEP .2,T*10: PRINT INK C(T
);"■ ";
690 NEXT T
700 FOR H=1 TO 60: BEEP .01,H:
NEXT H
```



```
710 POKE 23609,-1
720 PRINT '''"DO YOU WANT ANOTHE
R GAME?"
735 PRINT TAB 8;"ENTER Y OR N"
740 LET A$=INKEY$: IF INKEY$=""
THEN GO TO 740
750 IF CODE A$<>CODE "N" THEN A
UN
760 CLS
770 PRINT '' INK RND*6;TAB RND*
15;"OK, BYE FOR NOW)"
780 POKE 23692,-1
790 FOR H=1 TO 25
800 NEXT H
810 GO TO 770
```

```
   I AM THINKING OF A 4-COLOUR
CODE. YOU HAVE 10 GOES TO GUESS
IT. I CHOOSE FROM THESE COLOURS

     1 >>■
       2 >>■
         3 >>■
           4 >>■
             5 >>■

 ALL 4 COLOURS ARE DIFFERENT.

   PRESS ANY KEY TO BEGIN...
```

Line **20** (POKE 23609, **100**) changes the rate of 'click' when you press a key into a beep, to act as positive feedback when you press a key. We tend to use this all the time, and find it very useful when programming. Line **60** sets the INK and BORDER black (**0**) and the PAPER white (7). The routine from lines **100** to **120** print out the six colours (printing a blob of each colour) in a diagonal line, with the numbers next to the colours they refer to. Line **150** waits until any key is pressed before continuing.

The routine from **220** to **300** picks the colours, making sure that all four are different. Line **210**, meanwhile, has moved the print position down one (using the apostraphe from the 7 key, accessed with the red SHIFT key), and lines **180** to **200** have printed the six colours across the top of the screen, together with the numbers which refer to them.

Line 310 starts the loop to give 10 guesses. The second half of line 310 (POKE 23692, −1) ensures that if the screen is ever filled, it will automatically scroll, without requiring a response to the question "scroll?" which you often otherwise get at the bottom of the screen. Along with the key press beep, we also use this 'automatic scroll' POKE frequently.

Line 320 asks for the guess to be entered, and once it has (line 330), uses the backspace (CHR$ 8) 32 times to back over the line requesting the entry of the guess. Line 340 overprints this with blanks. This means that the line ENTER GUESS 2 is erased, but previous guesses (and the colour code at the top) are not, so you can look at previous guesses to help you work out your answer. You enter your guess, by the way, by entering a four-digit number, using the colour code given at the top of the screen. That is, to enter BLUE, just press 1.

The routine from lines 350 to 390 strip the number you have entered down to four separate digits, the variables for blacks (B) and whites (W) are set to zero in line 400, and then the guess is compared with the four-digit code the computer has thought of, giving little beeps for 'whites' or 'blacks' as it finds them. If you are right, the program tells you. If you are not, and you have not used up your 10 guesses, you are told of the digits of the right colour in the right position (blacks) and of the right colour in the wrong position (whites) and given another guess.

You will know that you can use PRINT AT 3, 6; "TEST" to print the word TEST four lines down, and starting seven spaces across. The control character CHR$ 22 behaves like PRINT AT, but with a difference. To get the same result as PRINT AT 3, 6; "TEST" you need to enter PRINT CHR$ 22 + CHR$ 3 + CHR$ 6; "TEST". However, because the Spectrum allows concatenation (the adding together of strings), you can add all these CHR$'s to equal one string. This can be quite useful, if you wish to specify a particular PRINT AT location several times in a program. Run the next program, and you'll see this working.


```
  10 LET a$=CHR$ 22+CHR$ 4+CHR$
7
  20 PRINT a$;"TEST"
```

TAB can be emulated by preceding CHR$ n, where n is the number of spaces (plus one) you wish to start printing on a line, with CHR$ 23. Run the next program to see this in action. However, as CHR$ 23 really expects to be followed by two numbers (n and m, which has the same effect as PRINT TAB n + 256*m), you can precede the information within the quote marks with a space, or a dummy letter (X in our example), which will not be printed. Run the next program, and you'll see that instead of printing XTEST right down the screen, it will simply print TEST.

```
10 LET a$=CHR$ 23+CHR$ 4
20 PRINT a$;"XTEST"
30 GO TO 20
```

At the start of this section, we discussed the eight colours, and looked at how these could be used for the information which is printed (INK), the background (PAPER) or the border (BORDER). The information printed can be modified by the use of two additional commands — BRIGHT and FLASH. The following routine shows these in action. Enter and run it, then return to the book for a brief discussion on these two new statements.

```
  40 PRINT INK 4;"NORMAL ■■■■"
 45 PRINT BRIGHT 1; INK 4;"BRIG
HT ■■■■"
 50 PRINT INK 4; PAPER 2;"NORMA
L ■■■■"
 55 PRINT BRIGHT 1; INK 4; PAPE
R 2;"BRIGHT ■■■■"
 60 PRINT FLASH 1; INK 4;"FLASH
ING ■■■■"
 65 PRINT BRIGHT 1; FLASH 1; IN
K 4;"BRIGHT ■■■■"
 70 PRINT FLASH 1; PAPER 2; INK
 4;"FLASHING ■■■■"
 75 PRINT BRIGHT 1; FLASH 1; PA
PER 2; INK 4;"BRIGHT ■■■■"
```

Although the effect of FLASHING is impossible to miss, you may need to look a little more closely to see the effect of BRIGHT. Once you have run this program, look at the word

BRIGHT, just under NORMAL near the top of the screen. You'll see this is a different shade of green. The white on green (the sixth line down on the screen) shows the effect of BRIGHT more clearly. Compare the 'lightness' of the word BRIGHT here with the word FLASHING just above it. With the non-FLASHING words printed in green on red (a pretty awful combination), you'll see that the 'bright' word is somewhat easier to read than is the 'normal' one.

Although the numbers zero to seven have been explained for INK, PAPER and BORDER, other numbers can be used. Using 8 (as in PAPER 8) means that no matter which is printed at this point, the colour will remain unchanged. This is not particularly useful in ordinary programming, but the number 9 can be quite effective.

'9' means contrast, and ensures that if you are printing on a light background, will print the words in black, and in white on a dark background, somewhat like the way the colour of an INPUT statement changes, depending on the BORDER colour. The next program shows this in action, printing randomly-generated letters of the alphabet, in random positions on the screen, against a randomly chosen PAPER colour. Run the program for a while to see this, and then return to the book for our next useful graphics command.

```
 50 PAPER RND*8
 70 CLS
 90 INK 9
100 FOR G=1 TO 32
110 PRINT AT RND*20,RND*30;CHR$
(65+INT (RND*26));
120 NEXT G
130 GO TO 50
140 PRINT AT RND*20,RND*30; OVE
R 1;CHR$ (65+INT (RND*26));
```

The word OVER is very useful, and can produce some very odd effects. You will have noticed an apparently useless line at the end of the previous program. Using the edit control, put line 140 in place of line 110, and change the 32 at the end of line 100 into 300. You'll notice, from time to time, that letters are printed on top of a letter which had previously been printed in that position. The OVER command means

that the new letter does not wipe out the one below it, but simply compliments it, 'subtracting' one from the other to form a new shape. This allows us to build up some characters of our own. Enter and run the next program to create some of your own. It is very hard to predict the effect of 'adding' various letters in this way. For example, a small "o" and a small "w" combine to produce what appears to be a capital "T".

```
 10 OVER 1
 20 FOR G=1 TO 16
 30 INPUT "ENTER A LETTER";A$
 40 INPUT "ENTER ANOTHER LETTER
";B$
 50 PRINT AT G,G;A$;CHR$ 8;B$
 60 NEXT G
```

You'll remember we discussed the way CHR$ 22 and CHR$ 23 could be used to replace PRINT AT and TAB, and the way these can be added together (concatenation) so that the whole command can be held in a single string. The same can be done with the other commands. The control characters, and the commands they replace, are: CHR$ 16 - INK; CHR$ 17 - PAPER; CHR$ 18 - FLASH; CHR$ 19 - BRIGHT; CHR$ 20 - INVERSE; CHR$ 21 - OVER. These are followed by the character which corresponds to the colour required. These can, as we said, be added, as the next program shows.

```
 20 INPUT PAPER 6; INK 1;"ENTER
A COLOUR FOR INK": INK
 30 INPUT INK 2;"ENTER A COLOUR
FOR PAPER":PAPER
 40 INPUT FLASH 1; BRIGHT 1; IN
K 4; PAPER 2;"ENTER A WORD";A$
 50 LET A$=CHR$ 16+CHR$ INK+CHR
$ 17+CHR$ PAPER+A$
 60 PRINT AT 10,10;A$
```

Line 60 in this program could also, of course, be added into the string, A$. Perhaps you might like to try to do this as an exercise.

Additional control characters are explained in the manual, where there is a table giving a complete description of the various effects available from the top row of the keyboard.

If you want to see how effective the colour can be, even from a simple program, enter and run the next routine. If the beeps drive you mad, delete lines 90 and 100. If you want the picture to build up more quickly, change the 7 at the end of line 40 into a 6, so that white blobs are not printed.

```
 10 PAPER 7: BORDER 0: CLS
 20 LET A=RND*10
 30 LET B=RND*16
 40 LET Z=RND*7
 50 PRINT AT A,B; INK Z;"■"
 60 PRINT AT 21-A,B; INK Z;"■"
 70 PRINT AT 21-A,31-B; INK Z;"
■"
 80 PRINT AT A,31-B; INK Z;"■"
 90 IF RND>RND THEN GO TO 20
100 BEEP RND/30,RND*60-RND*60
110 GO TO 20
```

When you've run this for a while, modify it to read as follows.

```
 10 PAPER 7: BORDER 0: CLS
 20 LET A=RND*10
 25 LET F=RND
 30 LET B=RND*16
 40 LET Z=RND*6
 50 PRINT AT A,B; FLASH F; BRIG
HT 1; INK Z;"■"
 60 PRINT AT 21-A,B; FLASH F; B
RIGHT 1; INK Z;"■"
 70 PRINT AT 21-A,31-B; FLASH F
; BRIGHT 1; INK Z;"■"
 80 PRINT AT A,31-B; FLASH F; B
RIGHT 1; INK Z;"■"
 90 IF RND>RND THEN GO TO 20
100 BEEP RND/30,RND*60-RND*60
110 GO TO 20
```

You'll see this has BRIGHTened each blob, and added a random FLASH to each circuit of the program. BRIGHT and FLASH understand 1 as on (so FLASH 1 turns it on) and 0 as off (so FLASH 0 turns it off). FLASH and BRIGHT, like various other commands, do not INT a random number, but round it up or down to the nearest whole number (where the INT of a positive number is always the nearest whole number *below* the number plus fraction), so the effect of line 25 in program nine b is to turn the FLASH on for some loops of the program, and off for others. You can see this is so by changing the RND in line 25 to a 1, then running it for a while,

then a 0 and running it for a while. Finally, you may like to modify the program to become the next program, 'Greek alphabet soup', a name you will understand once you've seen the program running. This final version recaps many of the points we've discussed so far in this section of the book.

```
  7 REM GREEK ALPHABET SOUP
 10 PAPER 7: BORDER 0: CLS
 20 LET A=RND*10
 25 OVER 1
 30 LET B=RND*16
 40 LET Z=RND*7
 45 LET A$=CHR$ (65+RND*26)
 50 PRINT AT A,B; BRIGHT 1; INK
Z;A$
 60 PRINT AT 21-A,B;; BRIGHT 1;
INK Z;A$
 70 PRINT AT 21-A,31-B; BRIGHT
1; INK Z;A$
 80 PRINT AT A,31-B; BRIGHT 1;
INK Z;A$
 90 GO TO 20
```

PLOT

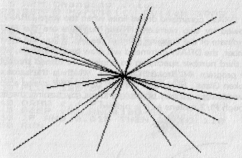
The PLOT commands allow very high resolution graphics, as can be seen by running the programs 'Galaxy' and 'Solid Sine'. Once you've run 'Solid Sine', you'll notice that while the dot resolution is 256 × 192, the colour resolution is only 32 × 22. In effect, the colour is mapped onto the PLOTted screen. Despite this lack of resolution in the colour, very effective high resolution designs can still be created.

```
 10 REM Galaxy
 20 PAPER 0: BORDER 0: CLS
 30 LET c=255: LET d=175
 40 INK RND*7
 50 LET a=c*RND
 60 LET b=d*RND
 70 PLOT a,b: PLOT a,d-b
 80 PLOT c-a,b: PLOT c-a,d-b
 90 IF RND>.5 THEN GO TO 60
 95 INK RND*7
100 GO TO 50
```

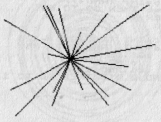
```
 10 REM Solid Sine
 20 REM © Colin Hughes
       Marinell    1982
 30 BORDER 2: CLS
 40 FOR x=0 TO 60 STEP .5
 50 LET y=20*SIN (x/30*PI)
 60 IF y=0 THEN GO TO 100
 70 FOR n=0 TO y STEP SGN y/4
 80 PLOT INK RND*6,x*3+30,3*(n+
30)
 90 NEXT n
100 BEEP .1,x: NEXT x
```

DRAW

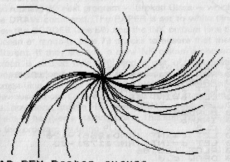
You can prove how effective the graphics can be by entering and running the next program — Broken Glass — which uses the DRAW command. The PAPER is set to white (line **30**), then the BORDER (line **50**) and the INK colours (line **60**) are chosen at random. Line **70** checks to ensure that these are different. If they are not, a new INK colour is chosen. The screen is cleared (line **100**) and a pair of coordinates are chosen randomly. A point is plotted in the centre of the screen (line **130**) and a line is DRAWn from this point to the previously chosen coordinates. The DRAW statement works out how long its line has to be, and at what angle, but it needs to be given a starting point. This starting point is given in this program by using PLOT.



```
 10 REM Broken glass
 20 REM © Marinell, Ruston 1982
 30 PAPER 7
 50 LET a=INT (RND*8)
 60 LET b=INT (RND*7)
 70 IF a=b THEN GO TO 60
 80 BORDER a
 90 INK b
100 CLS
110 LET c=INT (RND*256)-128
120 LET d=INT (RND*172)-85
130 PLOT 128,86
140 DRAW c,d
145 BEEP .01,RND*100-50
150 IF RND>0.02 THEN GO TO 110
160 RUN
```



The DRAW command draws lines when the word DRAW is followed by two numbers. These numbers are the PLOT coordinate of the finishing point of the line. If you add a third number, the DRAW command will draw part of a circle, with the third number specifying an angle to be turned through. The program — 'Broken Curves' — which is the same as 'Broken Glass', except for the end of line 140, draws a sort of windswept version of Broken Glass, by turning the line through PI/2 radians as it is plotted.



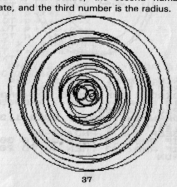
```
10 REM Broken curves
20 REM © Harinell, Ruston 1982
30 PAPER 7
50 LET a=INT (RND*8)
60 LET b=INT (RND*7)
70 IF a=b THEN GO TO 60
80 BORDER a
90 INK b
100 CLS
110 LET c=INT (RND*256)-128
120 LET d=INT (RND*172)-85
130 PLOT 128,86
140 DRAW c,d,PI/2
145 BEEP .01,RND*100-50
150 IF RND>0.015 THEN GO TO 110
160 RUN
```



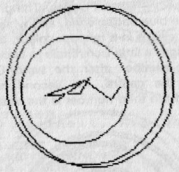
```
10 REM Broken curves
15 REM Changing colour
20 REM © Harinell, Ruston 1982
30 PAPER 7
50 LET a=INT (RND*8)
60 LET b=INT (RND*7)
70 IF a=b THEN GO TO 60
80 BORDER a
90 INK b
100 CLS
110 LET c=INT (RND*256)-128
120 LET d=INT (RND*172)-85
130 PLOT 128,86
135 IF RND>0.5 THEN INK RND*6
140 DRAW c,d,PI/2
145 BEEP .01,RND*100-50
150 IF RND>0.015 THEN GO TO 110
160 RUN
```

CIRCLE

There is still another graphics command, CIRCLE, which — as you might expect — draws circles. The program 'Tunnel Vision' sets a pale blue background, and white PAPER, then draws a series of circles in a random colour, around a centre point which changes a little from circle to circle, of a random radius. The first number after the word CIRCLE is the x-coordinate of the centre, the second number is the y-coordinate, and the third number is the radius.



```
 10 REM Tunnel vision
 15 BORDER 5: PAPER 7: CLS
 20 CIRCLE INK RND*6;128+RND*10
-RND*10,86+RND*7-RND-7,RND*65
 30 IF RND>.92 THEN CLS
 40 BEEP RND/3,RND*100-30
 50 GO TO 20
```



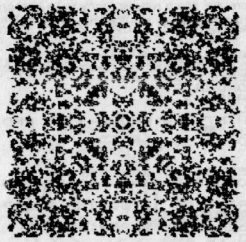
```
 10 REM Tunnel vision
 15 REM with random walk
 20 CIRCLE INK RND*6;128+RND*10
-RND*10,86+RND*7-RND-7,RND*65
 25 PLOT 128,86
 30 DRAW INK RND*6; OVER 1,32-R
ND*64,21-RND*42
 40 BEEP RND/4,RND*50-30
 50 IF RND>.33 THEN GO TO 30
 60 IF RND<.93 THEN GO TO 20
 70 RUN
```


Finally, here are some more programs to demonstrate how effective the graphics of the Spectrum can be.

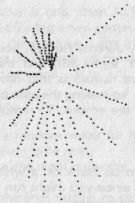
```
  5 PAPER RND*7
  7 REM Random coloured blobs
 10 REM screen demo
 20 BORDER 2
 25 PAPER 7
 27 CLS
 30 PRINT INK RND*5+1;AT RND*21
,RND*31;"■"
 40 GO TO 30
```

```
 10 REM Sound and vision
 20 BORDER RND*7: PAPER RND*7:
CLS
 25 PRINT AT 10,8; INK RND*7;"S
OUND and VISION"
 30 BEEP RND*50/200,RND*50
 40 GO TO 20
```

```
 10 REM Tesseract
 12 REM © Gourlay, Hartnell
 15 BORDER 1+RND: PAPER 7: CLS
 20 RANDOMIZE
 25 LET p=85: LET s=45
 30 LET x=RND*p: LET y=RND*p
 35 INK RND*6
 37 FOR g=1 TO RND*30
 40 PLOT p+x+s,p+y
 50 PLOT p+y+s,p+x
 60 PLOT p-x+s,p+y
 70 PLOT p+y+s,p-x
 80 PLOT p-x+s,p-y
 90 PLOT p-y+s,p-x
100 PLOT p+x+s,p-y
110 PLOT p-y+s,p+x
120 LET x=x+RND+RND-1
130 LET y=y+RND+RND-1
135 NEXT g
140 IF RND<.2 OR ABS x>p OR y>p
THEN GO TO 40
150 IF RND<.01 THEN GO TO 25
160 GO TO 30
190 RUN
```



```
  10 REM Whirlpool
  20 PAPER 7: CLS
  30 INK 0
  40 BORDER 7
  50 FOR a=1 TO 10
  60 FOR j=1 TO 7 STEP .1
  70 PLOT a*j*COS (j)+110,a*j*SI
N (j)+100
  80 NEXT j
  90 INK a/2
 100 NEXT a
```

```
  10 REM Spokesman
  20 PAPER 7: CLS
  30 INK 0
  40 BORDER 7
  50 FOR a=5 TO 19
  60 FOR j=1 TO 7 STEP .3
  70 PLOT a*J*COS (j)/2+110,a*j*
SIN (j)+100
  80 NEXT j
 100 NEXT a
```

```
  10 REM Forcefield
  15 REM (very slow)
  20 PAPER 7: CLS
  30 INK 0
  40 BORDER 7
  50 FOR a=2 TO 19
  60 FOR j=1 TO 7 STEP .01
  70 PLOT aa(j+4*SIN (j))*COS (j
)+110,a*(j+2*SIN (j))*SIN (j)+10
0
  80 NEXT j
  90 INK a/3.5
 100 NEXT a
```

String art

The program, written by Jeremy Ruston, bounces two balls around the screen (X,Y and L,M), and then draws lines between them. An extra feature is that new velocities are chosen every so often — so making the balls bounce at points other than the sides of the screen. To remove this feature, delete line 116.

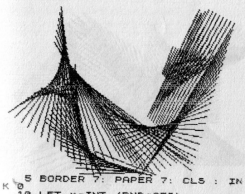
Line Description

5 Sets the colours to be black on a white background. This line is needed since you might run the program with different colours in effect.
10 Chooses a random x-coordinate for the first ball.
20 Chooses a random y-coordinate for the first ball.
30 Chooses a random x-coordinate for the second ball.
40 Chooses a random y-coordinate for the second ball.
50 Initializes the variables required for choosing velocities for the balls.
60 Defines a decent random number generator.
100 Calls the subroutine at line **1000**. This subroutine chooses random velocities for the balls.
110 Just showing off!
115 The variable 'num' holds the number of steps that will be made before new velocities are chosen.
116 If 'num' reaches zero, it is time to choose new velocities. The routine at **1000** also chooses a new value of 'num'.
120 Moves to x,y.
130 Joins x,y to L,M.
140 If adding the x-coordinate of the first ball to its velocity would take it off any edge of the screen, reverse the direction of movement, by negating the velocity.
150 See **140**.
160 See **140**.
170 See **140**.
180 Adds the velocities of the first ball to its coordinates.
190 And the same for the second ball.
200 Showing off again . . .
210 Generate a random number between **0** and 199 inclusive.

220 If the number is one, then RUN, so clearing the screen and creating a new pattern.
230 Otherwise, draw the next line in sequence.
1000 Makes the X velocity of the first ball be a random number in the range −V to +V.
1010 See **1000**.
1020 See **1000**.
1030 See **1000**.
1040 Chooses a random value for 'num'. Notice that num is not allowed to be zero.
1050 Returns to the main program.

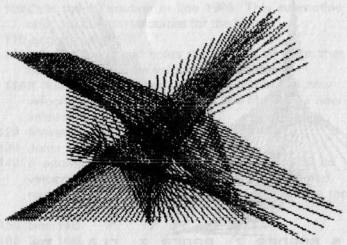
You can make the lines farther or closer from each other by altering the values of U and V in line **50**.

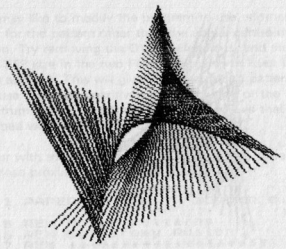
It is interesting to alter the DRAW statement in line **130** to draw a curve — but beware of drawing off the screen, and creating an error!

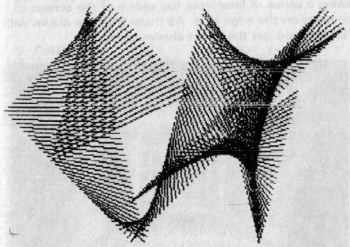


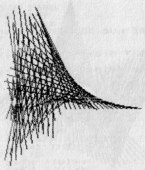
```
K, 0
  5 BORDER 7: PAPER 7: CLS : IN
  10 LET X=INT (RND*256)
  20 LET Y=INT (RND*176)
  30 LET L=INT (RND*256)
  40 LET M=INT (RND*176)
  50 LET U=15: LET V=7
  60 DEF FN r(x)=INT (RND*x)
```

```
  100 GO SUB 1000
  110 REM REPEAT
  115 LET num=num-1
  116 IF num=0 THEN GO SUB 1000
  120 PLOT x,y
  130 DRAW l-x,m-y
  140 IF x+a>255 OR x+a<0 THEN LE
T a=-a
  150 IF y+b>175 OR y+b<0 THEN LE
T b=-b
  160 IF l+c>255 OR l+c<0 THEN LE
T c=-c
  170 IF m+d>175 OR m+d<0 THEN LE
T d=-d
  180 LET x=x+a: LET y=y+b
  190 LET l=l+c: LET m=m+d
  200 REM UNTIL FALSE
  210 LET cont=FN r(200)
  220 IF cont=1 THEN RUN
  230 GO TO 110
1000 LET a=FN r(u)-v
1010 LET b=FN r(u)-v
1020 LET c=FN r(u)-v
1030 LET d=FN r(u)-v
1040 LET num=FN r(20)+10
1050 RETURN
```



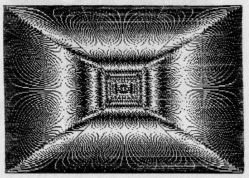






Moire patterns

This program draws Moire patterns. The program works by drawing a series of lines from the centre of the screen to each point on the edge of it. As these lines are drawn with OVER set, you get the effect shown below.

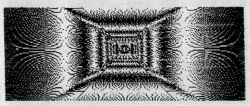


You may like to modify the program to use another central point for the pattern other than the actual centre of the screen. Try removing the OVER references, and increasing the STEP size in the two FOR statements in lines 10 and 50 to about 4. This will give you truer Moire patterns, but because of the compartively low resolution of the Spectrum screen, the effect is not so good as that obtained with, say, the BBC computer.

Colour with this program does not bode well, because of the close proximity of the dots drawn.

```
  1 PAPER 0: INK 7: BORDER 0: C
LS
  5 REM Moire patterns
  6 REM By Jeremy Ruston
  7 REM **********************
 10 FOR x=0 TO 255
 20 PLOT x,0
 30 DRAW OVER 1;255-x*2,175
 40 NEXT x
 50 FOR y=0 TO 175
 60 PLOT 0,y
 70 DRAW OVER 1,255,175-y*2
 80 NEXT y


  5 PAPER 0: CLS : BORDER 0
 10 FOR X=0 TO 255
 20 PLOT X,0
 30 DRAW OVER 1;255-X*2,175
 40 NEXT X
 50 FOR Y=0 TO 175
 60 PLOT 0,Y
 70 DRAW OVER 1;255,175-Y*2
 80 NEXT Y: REM © J. Ruston
 85 LET S=RND*5
 90 FOR X=255 TO 0 STEP -S
100 PLOT X,0
110 DRAW OVER 1,255-X*2,175
120 NEXT X
130 FOR Y=0 TO 175 STEP S
140 PLOT 0,Y
150 DRAW OVER 1,255,175-Y*2
160 NEXT Y
165 PAUSE 200
170 INK RND*7: PAPER 9: CLS
180 GO TO 85
```

```
   5 REM Spinning Triangles
   6 REM © Jeremy Ruston
   8 REM **********************
   9 INK 6: PAPER 0: BORDER 0: C
LS
  10 LET L=RND*256
  20 LET M=RND*176
  30 LET N=RND*256
  40 LET O=RND*176
  50 LET P=RND*256
  60 LET Q=RND*176
  70 LET A=RND*5
  80 LET B=RND*5
  90 LET C=RND*5
 100 LET D=RND*5
 110 LET E=RND*5
 120 LET F=RND*5
 130 REM REPEAT
 140 PLOT L,M
 150 DRAW N-L,O-M
 160 DRAW P-N,Q-O
 170 DRAW L-P,M-Q
 171 FOR X=1 TO 50: NEXT X
 180 IF L+A>255 OR L+A<0 THEN LE
T A=-A
 190 IF M+B>175 OR M+B<0 THEN LE
T B=-B
 200 IF N+C>255 OR N+C<0 THEN LE
T C=-C
 210 IF O+D>175 OR O+D<0 THEN LE
T D=-D
 220 IF P+E>255 OR P+E<0 THEN LE
T E=-E
 230 IF Q+F>175 OR Q+F<0 THEN LE
T F=-F
 240 CLS
 250 LET L=L+A: LET M=M+B
 260 LET N=N+C: LET O=O+D
 270 LET P=P+E: LET Q=Q+F
 280 REM UNTIL FALSE
 290 GO TO 130
```



POINT

POINT behaves to PLOT as SCREEN$ does to PRINT AT. That is, you can use POINT to determine the presence or otherwise of a PLOTed dot at a specific location.

Here is a simple program to demonstrate this, which scatters some PLOTs about at random, then checks them at random. Each time POINT (see line 70) finds a PLOT at the position it is checking, it BEEPs to let you know it has found one. You should get a 'success rate' of around 1%.

```
  10 REM PLOT..POINT
  20 FOR A=1 TO 100
  30 PLOT RND*100,RND*100
  40 NEXT A
  50 LET COUNT=0
  60 FOR G=1 TO 1000
  70 IF POINT (RND*100,RND*100)=
1 THEN BEEP .1,1: LET COUNT=COUNT
+1: PRINT AT 1,1;"SUCCESS RATE I
S ";COUNT/G*100;" %    ";AT 3,1;
"NUMBER OF TRIALS: ";G
  80 NEXT G
```

If POINT (x,y) equals one then there is a PLOTted point at that position. If it does not, POINT (x,y) equals zero. Try running the above program with the A loop running to **1000**, and the G loop to **10000**. Your success rate should be around **10** times higher than the first run (i.e. around **10%**). Why is that so?

The printer

There are three commands which are used in connection with the printer — LLIST, LPRINT and COPY.

LLIST — This dumps the entire program listing on to the printer.

LPRINT — This command also behaves like PRINT, writing to the printer rather than to the screen. It can be used in the direct mode, such as LPRINT "HI THERE", or within a program, such as 10 LPRINT "THE ANSWER IS "; A

COPY — This copies the entire contents of the screen, after a program has been run, to the printer. Because the printer cannot represent colours, or dark PAPER colours and lighter INKs, the copy on the printout may be considerably less impressive than the picture you have on your screen. Experiment with use of the INVERSE command on a local basis to enhance the printouts.

Random numbers

Random numbers are very useful for games playing. Let's examine the production of random numbers, and use them in a few simple programs.

The computer allows you to generate two floating point random numbers beween zero and one.

Enter and run the following to see a range of numbers between zero and one:

```
10 PRINT RND
20 GO TO 10
```

You'll get a list of numbers something like this:

```
.0011291504   0.69433594
.08581543     .075531006
0.43719482    0.6658783
0.79025269    0.94125366
0.2691803     0.59408569
0.18934631    0.55688477
0.20188904    0.75685096
0.14257813    0.51483154
              0.61291504
```


You'll find that random integers are often of far more use than are these numbers between zero and one. To do this, enter a statement like INT(RND *30) +1. Run this program. You're likely to get a series of numbers such as those following the program.

```
10 REM random integers
20 LET A=INT (RND*100)+1
30 PRINT TAB 8;A
40 GO TO 20
```

```
        14
        83
        19
        15
        51
        9
        41
        13
        72
        70
        86
        28
        44
        66
        15
        60
        96
        38
        36
        26
        85
        1
```

The computer takes the number in brackets (known as the *argument* of the function) and selects numbers at random between one and that number. To get negative random numbers, just put a minus sign in front of the word INT. Try that, and run it again, to get a result like this:

```
        -97        -88
        -68        -78
        -40        -50
        -88        -80
        -46        -6
        -91        -61
        -68        -9
        -42        -15
        -86        -46
        -52        -87
        -9         -25
```

You can use the random number generator for any application where you need to emulate a random activity in the real world, like the distribution of weeds in a garden, the spread of clouds in the sky, or the result of rolling dice. The next program emulates the roll of a six-sided die. Enter and run it a few times.

```
  10 REM *DICE ROLLER*
  20 PRINT "HOW MANY TIMES WILL"
  30 PRINT "I ROLL THE DIE?"
  40 INPUT A
  50 CLS
  60 PRINT "RESULT OF ROLLING","
THE DIE ";A;" TIMES"
  70 FOR B=1 TO A
  80 LET C=INT (RND*6)+1
  90 PRINT ,C
 100 NEXT B

RESULT OF ROLLING
THE DIE 6 TIMES
                 6
                 5
                 6
                 6
                 2
                 3
```

Bull fight

Here's a very simple game which shows the random number generator in action. The game is not really much of a game, but entering and running it is well worthwhile. Once you've played a few rounds of the game return to this book for a discussion of the program. You should be pleasantly surprised at how much you have already learned.

You are a matador. The bull will charge you 10 times. You select a number between one and three, and the bull does the same. So long as the numbers are different, you survive that move. If the bull picks the same number, the game is over. You are given a score at the end.


```
  10 REM BULLFIGHT
  20 LET SCORE=0
  30 FOR G=1 TO 10
  40 PRINT AT 4,4; INK 2;"THE BU
LL IS CHARGING"
  50 PRINT ' TAB 4; INK 1;"WHICH
MOVEMENT (1 TO 3)?"
  70 INPUT A
  80 IF A<1 OR A>3 THEN GO TO 70
  90 LET B=INT (RND*3)+1
 100 IF A=B THEN GO TO 220
 120 PRINT 'TAB 4; INK 4;"YOU AR
E SAFE IN MOVE "; INK 2;G
 130 BORDER RND*7
 140 PRINT ,,, INK 2;"THE BULL P
ICKED ";B
 150 PRINT ' INK 4;"YOU PICKED "
;A
 160 FOR H=1 TO 10
 170 BEEP .1,RND*50-RND*50
 180 NEXT H
 190 CLS
 200 NEXT G
 210 GO TO 230
 220 PRINT ' INK 2;"YOU HAVE FAI
LED AS" '"A MATADOR"
 230 FOR T=1 TO 50: BEEP .05,RND
*50: NEXT T
 240 PRINT ' INK 2;"YOU SCORED "
;100*(G-1)
```

```
   THE BULL IS CHARGING

   WHICH MOVEMENT (1 TO 3)?

YOU HAVE FAILED AS
A MATADOR
```

Note how the apostraphe (') in lines 120, 140, 150, 220 and 240 (available on the 7 key) is used to move the PRINT line down.

Let's go through the program line by line:

10 REM statement title.

20 Sets the variable SCORE to equal zero. We'll be discussing variables shortly.

30 Starts the FOR/NEXT loop to count the 10 goes. FOR/NEXT loops are discussed a little later in the book.
40 Prints out that the bull is charging.
60 Asks the player to enter a number between one and three.
70 Accepts the number from the player.
80 Checks to see if the number lies between one and three, and if it does not, goes back to line 70 to accept another input from the player.
90 Sets B equal to the bull's number, a number chosen at random between one and three.
100 compares the player's number (A) with the bull's number (B) and if they are the same, sends action to line 220 to tell you you have failed as a matador.
120 Tells the player he or she has survived that move.
130 Changes BORDER colour.
140 Tells the player the bull's number.
150 Reminds player of his or her number.
160–180 Puts in a short delay, with music, before next round.
190 Clears the screen.
200 Goes back for the next round.
210 If the player has survived 10 rounds, goes to print out the score.
220 This is the failure message, if A and B were found to be equal in line 100.
230 Delay loop, with music.
240 Prints out the score.

Reading through this explanation a couple of times, and looking carefully at the line or lines it refers to, should teach you quite a bit more about programming. There are a number of specific commands which we will look at in more detail, but you're probably starting to pick up quite a bit at this stage.

Variables

You will have noticed in the previous program that letters were used to represent numbers. The letter A was assigned (in line 70) to a number between one and three and B was assigned in the same way in line 90. The letters A and B in this program are called variables.

There are two types of variables: numeric and string (alphanumeric).

Almost any combination of letters and numbers can be used as a variable, so long as it begins with a letter, and there are no punctuation marks or symbols within the name. So SMUDGEPOT and D17 are valid variable names, while 2SMUDGE and 1D7 are not.Numeric variables, letters or combinations of letters and numbers beginning with a letter, are simple to use. You can assign a variable of this type to any number within the computer's numerical range. The Spectrum ignores spaces within variable names, and does not distinguish between small and capital letters (so a$ is the same as A$).

By the way, as you probably know, the computer uses scientific notation to display large numbers, with the number as a single digit and up to eight decimal places, followed by the letter E (for exponentiation) and the power of ten to which the number is to be multiplied. Enter and run the following demonstration which shows the variable A in use, being assigned to a number which is being multiplied repeatedly by 10, and then printed.

```
10 REM SCIENTIFIC NOTATION
20 LET A=1234
30 PRINT A
40 LET A=10*A
50 GO TO 30
```

```
1234
12340
123400
1234000
12340000
1.234E+8
```

```
1.234E+9
1.234E+10
1.234E+11
1.234E+12
1.234E+13
1.234E+14
1.234E+15
1.234E+16
1.234E+17
1.234E+18
1.234E+19
1.234E+20
1.234E+21
1.234E+22
1.234E+23
1.234E+24
```

Note that after the number has eight digits (12340000) it is printed as a number, a decimal point, more numbers after the decimal point, the letter E and a power of 10. Try and predict how long this program will run until it exceeds the maximum number possible on the computer, then run it until it crashes to see if you were right.

Looking at the listing tells us another couple of things about variables. The variable is assigned by just entering the name of the variable (in this case, A), preceded by the word LET, and followed by an equals sign, and the value which we want assigned to the variable. If we said LET A = 99, then following this with PRINT A would produce 99. Line 40 looks a little odd. The asterisk (*) stands for multiply in BASIC. Line 40 seems to be saying that A is equal to 10 times itself, which — in terms of standard arithmetic — is not true. This is, however, the way the assignment (LET) statement is used in BASIC.

String variables

String variables are a letter, followed by a dollar sign. Enter LET A$ = "HELLO", press ENTER then PRINT A$, ENTER, will give you HELLO. You can put anything, including numbers, symbols, punctuation marks and letters within the quote marks, to be assigned to a string variable. A series of letters and whatever, within quote marks in this way, is known as a string.

Crickets

There is, strange to say, a correlation between the temperature and the number of times a cricket chirps each minute. The following program which shows long variable names in action, converts the number of chirps per minute into the temperature, in degrees Fahrenheit. Enter and run it a few times. Note that the variable chirp is set equal initially to 80 in line 20. This is converted into the variable temperature in line 30, and this latter variable is used in the PRINT statement in line 40. The variable chirp is incremeneted by a random number between one and seven in line 60, there is a short delay (lines 70 and 80) and then the program returns to line 30 to go through the whole process again. It will run for a long, long time (until you exceed the highest possible number the computer can cope with) if you do not interrupt its running with the BREAK key.

```
THE TEMPERATURE IS 62
WHEN THERE ARE 87 CHIRPS

THE TEMPERATURE IS 63
WHEN THERE ARE 90 CHIRPS

THE TEMPERATURE IS 63
WHEN THERE ARE 92 CHIRPS

THE TEMPERATURE IS 64
WHEN THERE ARE 96 CHIRPS

THE TEMPERATURE IS 65
WHEN THERE ARE 98 CHIRPS
```

```
 10 REM CHIRP CONVERTER
 20 LET CHIRP=80
 30 LET TEMPERATURE=INT ((CHIRP
/4)+40.5)
 40 PRINT '"THE TEMPERATURE IS
"; INK 2;TEMPERATURE
 50 PRINT "WHEN THERE ARE "; IN
K 2;CHIRP; INK 0;" CHIRPS"
 60 LET CHIRP=CHIRP+INT (RND*7)
 70 FOR J=1 TO 100
 80 NEXT J
 90 POKE 23692,-1
100 GO TO 30
```

Although it takes a little longer to type in long variable names, these have a clear advantage over use of names like A, B and C2. You know, without having to refer back, what each variable represents. Here is another program which uses two variable names to help make it clear what is going on. Enter and run this.

```
 10 REM ** VARIABLES **
 20 LET U$="THE NUMBER IS "
 30 LET NUMBER=3
 50 PRINT U$;NUMBER
 60 PRINT '"THE SQUARE OF";NUMB
ER
 70 PRINT TAB 5;"IS ";NUMBER*NU
MBER
 80 PRINT '"AND THE SQUARE ROOT
"
 90 PRINT "IS ";SQR (NUMBER)
```

To summarise:

- Numeric variable — This can have any name, so long as it starts with a letter and does not contain punctuation or symbols..
- String variable — This is a letter followed by a dollar sign, which is assigned to anything within quote marks. All variables are assigned by use of a LET statement, followed by the name of the variable, an equals sign, and then the value to be assigned to the variable.

INPUT

The INPUT statement is used to get information from a user while a program is actually running. The computer stops when it comes to an INPUT statement and waits for an entry of some kind from the keyboard before it will continue with the execution of the program.

Enter and run the following, which shows numeric INPUTS in action. The program will wait for you to enter one number, then press ENTER, then wait for another number. After you have pressed ENTER again, it will print the sum of the two numbers.

```
10 REM ** input **
20 INPUT x
30 PRINT ;x
40 INPUT y
50 PRINT ;y
60 LET z=x+y
70 PRINT ;"="
80 PRINT ;z
90 PRINT ;
```

```
459
247
=
716
```

This is OK so far as it goes, but you would not have known what to do when you ran the program unless you had read it in this book. There is a simple way to rectify this, by programming in user prompts. The preceding program can easily be rewritten so that the user has no doubt as to what he or she is meant to do.

```
10 REM ** input **
20 INPUT "Give me a number ";x
30 PRINT ;x
40 INPUT "And another ";y
50 PRINT ;y
60 LET z=x+y
70 PRINT ;"="
80 PRINT ;z
90 PRINT ;
```

Running this shows that the computer prints up the words within the quote marks, then waits for the input.

Note that many of the commands used to control PRINT output can be used with INPUT, as can be seen in the next program.

Combat

```
  5 REM ** Combat **
  6 POKE 23609,100
 10 LET score=0
 15 FOR j=1 TO 20
 20 PRINT AT 1,8; INK j/3;"Go n
umber ";j
 30 INPUT INK j/3;"Enter a numb
er from 1 to 10 ";a
 40 IF a<1 OR a>10 THEN GO TO 3
0
 50 PRINT AT 10,0; INK 3;"Your
number is ";a;AT 12,6; INK 1;"Sc
ore is ";score
 60 FOR g=1 TO 4
 70 LET b=INT (RND*10)+1
 80 PRINT AT 3,3; INK 2;b;" "
 85 FOR m=1 TO 10: BEEP .01,3.5
*m: NEXT m
 90 IF b=a THEN GO TO 110
100 NEXT g
110 IF a=b THEN LET score=score
+1: PRINT AT 14,6; INK 4;"Well d
one"
120 IF a<>b THEN PRINT AT 14,8;
INK 0; FLASH 1;"Bad luck"
150 PRINT AT 12,6; INK 2;"The s
core is ";score
160 IF score=5 THEN GO TO 250
170 FOR t=1 TO 20
180 BEEP .01,2*t: BEEP .01,20-t
: NEXT t
190 CLS
200 NEXT j
210 PRINT BRIGHT 1; INK 2;"The
game is over"
220 PRINT FLASH 1; BRIGHT 1; IN
K 2;"And you only scored ";score
230 PRINT ' INK 1;"Your rating
is ";score/.05;" Percent"
```



```
240 STOP
250 PRINT INK RND*6; FLASH 1;TA
B 4;"You did it "
260 FOR m=1 TO 4: BEEP .01,RND*
50: PAUSE 3: NEXT m
270 PRINT INK RND*6; FLASH 1;TA
B 8;"YOU win!!"
280 POKE 23692,-1
290 GO TO 250
```

```
      Go number 1
 5


Your number is 3
    The score is 0
      Bad luck
```

In COMBAT, you select a number between one and ten. The computer selects up to four numbers between one and ten. If any of them is the same as yours, your score is increased by one. If you get a score of five, within your 20 goes, you win. If not, you fail and get a percentage rating. Once you've run the program, come back to the book to go through it line by line. Although the program is fairly trivial, running it, then following through the explanation will increase your knowledge of several aspects of BASIC, and — of course—shows INPUT in action.

5 Title.

6 Puts a 'pip' into key presses.

10 Sets the variable score to zero.

15 Starts the master FOR/NEXT loop to count your goes.

20 Prints the go number.

30	Accepts the input for variable A, using J to set the colour.
40	Checks that the input is legal.
50	Prints out the number chosen, and the score. Note that PRINT statements may be 'chained' in this way, with semi-colons and the use of AT or TAB.
60–100	Generates up to four numbers. After each number is generated (line 70), it is printed (line 80), there is some sound (line 85), and checked against the player's number (line 90).
110	Score is increased by one if the guess was correct and the program prints WELL DONE.
140	Prints BAD LUCK if the guess is incorrect
150	Reprints the score.
170–180	Short delay with BEEPs before next move.
190	Clears the screen.
200	The end of the master FOR/NEXT loop.
210–240	End of game, if you lose.
250–290	End of game, if you win.

Compound Interest

This next program to show the INPUT statement in action again, also shows the use of explicit names for variables, which make it easier to understand what is going on. You may well want to save this program on cassette, as it has a degree of practical application.

```
 10 REM SIMPLE AND COMPOUND
      INTEREST
 20 REM
 30 INPUT INK 1;TAB 5;"PRINCIPA
L? ";PRINCIPAL
 40 INPUT INK 2;TAB 5;"INTEREST
? ";INTEREST
 50 INPUT INK 4;TAB 4;"FOR HOU
MANY YEARS? ";YEARS
 60 PRINT INK 3;"YEAR";TAB 7;"S
IMPLE";TAB 17;"COMPOUND";TAB 27;
"DIFF."
```



```
 90 POKE 23692,-1
 100 PRINT INK 1;"
 110 FOR M=1 TO YEARS
 120 LET SIMPLE=PRINCIPAL+M*PRIN
CIPAL*(INTEREST/100)
 130 LET COMPOUND=INT (100*PRINC
IPAL*(1+INTEREST/100)↑M)/100
 140 LET DIFF=INT (100*(COMPOUND
-SIMPLE+.005))/100
 160 PRINT M;TAB 5;SIMPLE;TAB 17
;COMPOUND;TAB 27;DIFF
 170 NEXT M
```

```
YEAR    SIMPLE      COMPOUND    DIFF.
1       108.25      108.25      0
2       116.5       117.16      0.66
3       124.75      126.84      2.09
4       133         137.31      4.31
5       141.25      148.64      7.39
6       149.5       160.9       11.4
7       157.75      174.17      16.42
8       166         188.54      22.54
9       174.25      204.1       29.85
10      182.5       220.94      38.44
11      190.75      239.17      48.42
12      199         258.9       59.9
```

This program works out compound and simple interest, for a principal and interest rate you determine, over the number of years you decide. The example uses a principal of $100, at 8.25% over 12 years.

To stop a program during a string INPUT (BREAK does not operate during INPUTs), use cursor left (SHIFT 5) or RUBOUT (SHIFT 0) to get the cursor out of the quotes, then type in STOP (SHIFT A) followed by NEWLINE. If you are in a numeric INPUT without quotes, just type STOP (SHIFT A) followed by ENTER. In both cases the program stops with report 9.

It is useful to be able to reject invalid INPUTS, *before* they cause a program to crash.

If you invite a user to have another go, analyse his or her reply as follows:

```
 555 INPUT "DO YOU WANT ANOTHER
GO? ";R$
 556 IF R$(1)="Y" THEN RUN
```

There is a law somewhere that says the user will respond by pressing only ENTER — leaving you with a null INPUT. So there's no such thing as R$ (1) — it does not exist, as the computer will very quickly tell you in the form of an error report!

Here is one method of preventing this:

```
 550 DIM R$(1)
 555 INPUT "DO YOU WANT ANOTHER
GO? ";R$
 556 IF R$="Y" THEN RUN
```

Because R$ has previously been DIMensioned, it will have to consist of one character, no matter what is entered. If only ENTER is pressed then R$ will be a space since that is what is placed in R$ after DIM and a null INPUT will not change it. If the INPUT is several characters long, then there is only room in R$ for the first character. If this character is "Y" then the program will RUN for another go. This method has the advantage that if the user enters a very long reply such as "YES PLEASE NICE KIND COMPUTER, I WOULD LIKE VERY MUCH TO HAVE ANOTHER GO AT YOUR GREAT GAME PROGRAM" (very unlikely!), there is no need to store it all in memory. It is also very useful if you GO TO or do nothing that would CLEAR the variables, thus storing the entire reply unnecessarily. The second method is more conventional and uses one program line less than the previous routine, although it does place the entire reply unnecessarily in memory:

```
 550 DIM R$(1)
 555 INPUT "DO YOU WANT ANOTHER
GO? ";R$
 556 IF CODE R$=CODE "Y" THEN RU
N
```

The program explains itself really — if the first character of the reply has a CODE that is the same as the CODE of Y (i.e. it *is* Y) then the program RUNs again. Null INPUTs are rejected

as meaning the user does not want to play again, since merely pressing NEWLINE gives the empty string and the CODE of the empty string is 0 like a space.

Checking the first letter of a user's INPUT is fairly easy as you've just seen. It becomes a bit more difficult when you want to check an entire INPUT, e.g. to see if the user has entered any punctuation marks or has included letters in a numeric INPUT. Let us look at alphabetic INPUTs first. The relational operators <,>,>=,<=,><> are very useful in this case. Take the case of an INPUT where a word is strictly required and nothing else must be entered.

```
  10 INPUT A$
  15 IF A$="" THEN GO TO 10
  20 FOR A=1 TO LEN A$
  30 IF A$(A)<"A" OR A$(A)>"Z" T
HEN GO TO 10
  40 NEXT A
```

Line 15 ensures that null INPUTs (i.e. just ENTER pressed) are rejected. The loop starting at line 20 scans the entire INPUT string character by character and if a character is found which is not a letter then you are instructed to enter the string once again because the program jumps back to line 10. As it stands, the program will not allow spaces between words.

Change line 30 like this to allow spaces:

```
 30 IF (A$(A)<"A" OR A$(A)>"Z")
AND A$(A)<>" " THEN GO TO 10
```

You can easily extend this idea to allow punctuation marks, letters and spaces if you like (i.e. numbers, keywords, symbols etc. are not allowed) by extending the idea in line 30. Only slightly more difficult is detecting a given word in an INPUT, e.g. if you had a line at the tail end of a program inviting the user to have another go at the program and if the user replied "YES" then the program re-ran. It is a fairly simple thing to put the INPUT in a loop and slide the word along like this:

```
7010 INPUT "ANOTHER GO? ";A$
7020 FOR A=1 TO LEN A$-2
7030 IF A$(A TO A+2)="YES" THEN
RUN
7040 NEXT A
7050 STOP
```

If you entered "YES" or "YES PLEASE" the program will re-run as required. If a word whose length is less than the length of the search word (except the empty string) is INPUTed then this will cause an error because of line **7030** which expects the INPUT to be at least equal to the search word. The empty string is alright because then LEN A$ is **0**, making line **7020** FOR A = 1 TO **0**, so the string is totally bypassed, and the problem does not arise. Try also entering "YESTERDAY" — the routine reruns because it has detected the three letters "YES". What is needed is a routine that detects if the character on either side of those three letters is anything other than a letter. We need to be careful doing this because we cannot examine the character before and after the three letters "YES" if they occur at the beginning or at the end of an INPUT because they do not exist and to attempt to examine them would cause a subscript error. There follows a routine which makes allowances for this, by adding dummy characters at the start and end of A$.

```
7010 INPUT "ANOTHER GO? ";A$
7015 LET A$=" "+A$+" "
7020 FOR A=2 TO LEN A$-3
7030 IF A$(A TO A+2)="YES" AND (
A$(A-1)<"A" OR A$(A-1)>"Z") AND
(A$(A+3)<"A" OR A$(A+3)="Z") THE
N RUN
7040 NEXT A
7050 STOP
```

The routine allows all lengths of INPUT up to the maximum length that a string may be. If you want to change the search word in a program then it may be worth assigning it to a variable or having an INPUT somewhere in the program for the search word. You will have to make the following modifications to the routine to use a different search word:

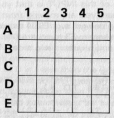
```
7010 INPUT "ENTER SEARCH WORD";S
$
7020 INPUT "ENTER SENTENCE";A$
7040 LET A$=" "+A$+" "
```



```
7050 LET LS=LEN S$
7060 LET LA=LEN A$
7070 FOR A=2 TO LA-LS
7080 IF A$(A TO A+LS-1)=S$ AND (
A$(A-1)<"A" OR A$(A-1)>"Z") AND
(A$(A+LS)<"A" OR A$(A+LS)>"Z") T
HEN RUN
7090 NEXT A
```

If the routine is a bit too long then provided you are using the same search word every time then you can avoid using S$ and LS and spell out the search word in full every time it is used, and replace all references to LS with the length of the search word. See the example using "YES" above.

Let us now look at another type of INPUT that is commonly used in games — both grid type games and board games — that is an INPUT involving coordinates as you would find on some maps. For instance you might have a board laid out like this:



The coordinates are usually entered in the form of a letter followed by a number e.g. C3 if you are referring to one square as in a Hunt the Hurkle type of game or C3B4 if you are using from-to coordintes in a board game such as draughts. If you have decided that the coordinates are to be entered in the form of a letter followed by a number then the chances are that sooner or later someone will — whether deliberately or accidentally — enter the coordinates in the wrong order and foul up the program. This routine will automatically detect if the two characters of a coordinate have been entered in the wrong order and sort them out. It

applies to the board layout above and to modify it for other ranges of characters, simply change the characters in quotes in lines 30 and 40. Line 50 has merely been included so that you can see the effect of the routine if any.

```
  10 INPUT A$
  20 IF LEN A$<2 THEN GO TO 10
  25 LET A$=A$( TO 2)
  30 IF A$(1)>="1" AND A$(1)<="5
" AND A$(2)>="A" AND A$(2)<="E"
THEN LET A$=A$(2)+A$(1)
  40 IF A$(1)<"A" OR A$(1)>"E" O
R A$(2)<"1" OR A$(2)>"5" THEN GO
 TO 10
  50 PRINT A$
```

The routine is very quick to RUN. It is a very difficult routine to crash but I'm sure some clever reader will find a way. If you do find a way of beating the routine then modify the routine to prevent that error happening again.

The routine for a four character coordinate is somewhat more complex. The idea of this INPUT is that you can enter the number of the square you are moving *from* and the square you are moving *to* in one go, e.g. E3D4 would mean that you moved a piece from square E3 to square D4. Let us first of all arrange the letters and numbers into order.

```
  10 INPUT A$
  20 IF LEN A$<4 THEN GO TO 10
  30 LET A$=A$( TO 4)
  40 IF A$(1)>="1" AND A$(1)<="5
" AND A$(2)>="A" AND A$(2)<="E"
THEN LET A$( TO 2)=A$(2)+A$(1)
  50 IF A$(3)="1" AND A$(3)<="5"
 AND A$(4)>="A" AND A$(4)<="E" T
HEN LET A$(3 TO )=A$(4)+A$(3)
  60 IF A$(1)<"A" OR A$(1)>"E" O
R A$(2)<"1" OR A$(2)>"5" OR A$(3
)<"A" OR A$(3)>"E" OR A$(4)<"1"
OR A$(4)>"5" THEN GO TO 10
  70 PRINT A$
```

Note that you can shorten these two routines by using the DIM command. In the first program you can add

5 DIM A$(2)

and delete lines 20 and 25. For the second program add

5 DIM A$(4)

and delete lines 20 and 30. What both versions achieve is to ensure that the string A$ is neither shorter nor longer than the required length. If you enter an INPUT which is longer than four characters in the second routine, then the rest are ignored. If shorter than four characters then spaces are added if you have line 5 added (then rejected in line 60) or rejected in line 20 if you are using the unmodified version. Having sorted out the letters and numbers let us look at sorting out legal and illegal moves. You will need to look at the example board a couple of pages back for this. Suppose we have uncrowned draughts piece on square E3. We need to work out the legal moves from there. An uncrowned draughts piece can only move one square forward in a diagonal direction. The square it may end up on are D2 or D4. Before reading on, can you work out the relationship between the coordinates?

Since the piece can only move forward one square at a time, it has to end up on a square whose letter is alphabetically nearest to E. Now on your computer, the CODEs of letters that follow each other alphabetically step up or down by 1, so that the CODE of D is 1 less than the CODE of E. Therefore if the CODE of the *from* square letter is not 1 greater than the CODE of the *to* square letter then it is not a legal move. The number of the *from* square must be 1 greater or 1 less than the number of the *to* square, so we end up with:

```
  65 IF (CODE A$(1)<>CODE A$(3)+
1) OR (CODE A$(2)<>CODE A$(4)+1)
 OR (CODE A$(2)=CODE A$(4)-1) TH
EN GO TO 10
```

Obviously you will need to adapt these routines to suit your programs and they are only intended to show you the basis of routines that you may like to incorporate into your programs. They also help to demonstrate the approach you need to take to solve problems of this kind. For what it's worth, we suggest you try to follow these guidelines:

(i) Work out exactly what you want to accomplish.
(ii) Work out exactly what is permitted, and some of the things which are not allowed (e.g. the empty string).
(iii) How can you prevent these happening, or reject them when they do happen?
(iv) Quickly work out in your head whether your routine does what you think it will by using it with a couple of examples.
(v) If you are happy with your routine enter it into the computer and try it out with some permitted values or characters to check whether there is a bug that prevents these values being entered. When happy with this, try out the routine with all sorts of INPUTs (for example, try entering a non-existent coordinate such as F9 in the routines above). You are now ready for the most important test.
(vi) Let a friend loose on the routine with orders to make a fool of the routine. The above routines do have a fallacy but I'm not telling you what it is—that's an exercise for you.

Finally, let's look at numeric INPUTs. Clear the computer with NEW and enter the following:

```
10 INPUT A
20 GO TO 10
```

RUN this little program and see if you can cause it to crash in any way; it shouldn't be too difficult. Try entering a letter; try entering STOP; try entering a number too large or too small for the computer to handle; or try entering a keyword or arithmetic sign such as "+".

Arithmetic signs cause the computer to display a syntax error marker, although it does not stop the program. Keywords and symbols also cause this to happen, although letters cause the program to stop with error code 2, which means that an undefined variable has been used. Variable? Yes — when you

enter a letter in response to a numeric INPUT the computer thinks you're entering a variable and this can sometimes be very useful. With the same program, enter 1 on the first INPUT, then enter A the second time, and it is accepted! What the computer has done is look up the value of A and assigned it to A — in other words it hasn't changed the value of A. Now enter STOP. The program stops on an error message. Now try typing in PRINT A, and you get the number 1, so the program has stopped *before* updating the value of A. In fact if you do manage to crash a numeric INPUT then in general the computer retains the previous value of the variable. Not that it's all that useful, but under certain circumstances if you do manage to restart the program then the variable does have a value.

The easiest way to get around these problems is to use string INPUTs and evaluate using VAL. Try:

```
10 INPUT A$
20 LET A=VAL A$
30 PRINT A
40 GO TO 10
```

You should find this quite easy to crash, and many things true of numeric INPUTs seem to happen with this little routine. However, the advantage of this method is that it does not crash until you apply VAL if there is an error. You can process the string before applying VAL after INPUTing it and spot or remove errors before they crash — that is you can *process* the string. The thing to remember is that VAL can work with anything numeric, not just numbers. Try the following:

```
PRINT VAL "RND"
PRINT VAL "SGN -7"
PRINT VAL "A*2" (this only works if you have previously defined A of course)
PRINT VAL "COS1"
```

Undefined variable names are the curse of VAL, along with non-numeric statements or keywords or symbols. These have to be weeded out before you can apply VAL. The easiest case

is that where only numeric INPUTs are allowed and you can do that like this:

```
10 INPUT A$
20 FOR F=1 TO LEN A$
30 IF A$(F)<"0" OR A$(F)>"9" T
HEN GO TO 10
40 NEXT F
50 LET A=VAL A$
60 PRINT A
```

Can you see straight away what would defeat the routine? Our old friend the empty string of course, which would make the loop FOR F = 1 TO 0, so it would be totally bypassed and useless so you have to add 15 IF A$="" THEN GO TO 10. The routine makes you enter the number again if you have entered anything but numbers. You can extend the idea to permit arithmetic symbols and variable names if you like, but there is so little use for it it hardly seems worthwhile.

```
30 IF (A$(F)<"0" OR A$(F)>"9")
AND (A$(F)<>"+" AND A$(F)<>"") 
THEN GO TO 10
```

This allows you to enter addition symbols and exponentiation symbols. To permit additional functions then simply add them within the second set of brackets linked together by AND. This is not terribly useful but you may find a use for it some day.

We'll be looking at VAL and other string-processing functions in detail a little later in the book, but for now, we need to examine commands which lie at the heart of the computer's power to 'think'.

GO TO

One important ability in programming is to be able to branch to different parts of the program during execution. Without this, the program would always run from the lowest line number to the highest, and then stop. One statement which allows you to move around the program at will is GO TO. The GO TO statement consists of a line number followed by the word GO TO and another line number, or followed by a calculation (such as GO TO 2*X, or GO TO 200+340).

If the computer came across 140 GO TO 190, it would jump immediately from line number 140 to line 190. This is called an unconditional branch. That is, it is a jump that does not depend on the existence of any condition. Once at line 190, the program continues to execute in order, until it comes to the end, or comes to another line directing it somewhere else.

You can use GO TO to produce programs which run almost for ever. These can be quite effective, especially at the end of a game. Run the following to see this in action:

```
10 PRINT INK RND*6;"..You have
won...";
20 POKE 23692,-1
30 BEEP .01,RND*50
40 GO TO 10
```

IF....THEN GO TO

The IF statement has a similar function to GO TO, but it will only reroute the program IF certain conditions are fulfilled. This creates a conditional branch. The IF/THEN statement is made up of a line number followed by the words IF/THEN/GO TO separated by a relationship which must be determined before leaving the line. These are six relation operators which can be used to compare two variables. These are:

= equal to
> greater than
< less than
>< not equal to
> = greater than or equal to
< = less than or equal to

These operators are used to connect the IF....THEN statements to form the condition to be determined.

Here's an example: 70 IF Z> = 10 THEN GO TO 100

This will be read by the computer to mean IF the value of the variable Z is greater than, or equal to, 10 THEN the program will branch to line 100. If Z is less than 10, the program will continue normal execution, with line 80.

This gives the computer decision-making power, the real source of a computer's apparent ability to think.

As you've probably discovered, the computer isn't indecisive (unless you tell it to be), it makes a firm decision every time whether or not to do something. What it actually does depends on what you tell it to do, usually after the word THEN in the line. Let's illustrate this with a simple program to print out the number you have just entered in words instead of digits.


```
  10 INPUT INK RND*6;"Enter a nu
mber 1 to 3";a
  30 IF a=1 THEN PRINT "one"
  40 IF a=2 THEN PRINT "two"
  50 IF a=3 THEN PRINT "three"
  60 GO TO 10
```

You need not be limited to one condition between the IF & THEN. To take the example above, suppose you were allowed to go home at five o'clock only if you'd finished your work:

IF it's five o'clock AND you've finished your work THEN go home. When you want to join two or more condition expressions such as "you've finished your work", you can use three connecting words to join the expressions. These are AND, OR and NOT. If you have a conditional expression with AND joining the two parts, then the computer only does something if both parts are true. If it's five o'clock but you haven't finished your work then you are not entitled to go home, for example.

To illustrate TRUE and FALSE, try this program:

```
  10 INPUT a
  20 INPUT b
  30 IF a=1 AND b=1 THEN PRINT "
true"
  40 GO TO 10
```

Try entering different values and see the results. Try changing the values in line 30 to see what effect this has. Make a note of your answers until you understand what's going on.

Let's look at OR. Think of OR along the lines of IF it's five o'clock OR the boss says you can leave early THEN go home—that is do something when one of the alternatives is true. More correctly, do something when at least one of the alternatives is true, because it does not matter how many are true (they may all be) as long as at least one is true. So you go home at five o'clock anyway, but you may also go home when the boss says you may — either fact entitles you to go home. Try experimenting with this program in the same way as you experimented with the previous program.

```
10 INPUT a
20 INPUT b
30 IF a=1 OR b=1 THEN PRINT "t
rue"
40 GO TO 10
```

The last word is NOT. It doesn't join expressions like the other two, but changes their meanings. Study this:

IF NOT the manager has said you can go home THEN stay at work.

This means that unless you've been told that you may go home you must stay at work. What happens is that the computer looks at the expression and decides that if it isn't true then it does something (for this purpose ignore the NOT for deciding what is true and what isn't). That is, IF NOT . . . . . THEN is true when whatever follows NOT is false. Something is done only when a condition is *not* met. Try this:

```
10 INPUT a
20 INPUT b
30 IF a<>b THEN PRINT "true"
40 GO TO 10
```

This may confuse you at first, but if you experiment with the values of A and B you will notice a pattern of results which illustrate the workings of NOT.

You may have noticed that we have used the = symbol in all the examples so far. Remember, this is only one of six *relational operators*. Here is the list again of the six used on your computer:

```
<    is less than.
>    is greater than.
< =  is less than or equal to.
> =  is greater than or equal to.
<>   is not equal to (or is anything other than).
=    equals, is the same as.
```

Change the programs so that you use all of these RELATIONAL OPERATORS. Play around with the programs until you find yourself able to predict what happens each time. Try combinations of AND, OR & NOT and see in which order they are worked out.

See if you can work out how to change the result by putting brackets around expressions. Note that this will not in every case, so if a certain expression gives problems, leave it and try another one. This order of evaluation is called *priorities*, and is dealt with in detail later in this book.

We can apply conditional expressions to strings as well as numbers.

```
10 INPUT a$
20 IF a$<>"FRED BLOGGS" THEN N
EW
```

RUN this program to see what happens. The first time you RUN it, enter the name FRED BLOGGS in capital letters. The program comes to a halt normally. Pretty unexciting. RUN it again and this time try entering your own name (if your name happens to be FRED BLOGGS then enter somebody else's name). This time the program will self-destruct because of the NEW in line **20**. If you substitute your name or a code number for FRED BLOGGS then you will have a program that will only work for you or those that know the code, and self-destructs if anyone else attempts to use it.

Let us now look at values in conditional expressions. First of all we'll use the relational operators. You will find that true is represented by 1 and false by **0**.

```
10 INPUT a
20 INPUT b
30 LET x=(a=b)
40 PRINT x
50 GO TO 10
```

You don't actually need the brackets in line **30** but they help to make the meaning clearer. The expression in brackets is reduced to either **0** or 1 — which one depends on the values you enter. Try this using all six of the relational operators and make a note of the results. You should get either a **0** or a 1 every time — you might think that this is a bit restrictive. In fact since this value of **0** or 1 can be considered as a number, you can manipulate it as you would any number. The best way of manipulating the number is by

multiplying, since this will change true values but not false values (anything multiplied by 0 is 0). Try changing the program to this:

```
10 INPUT a
20 INPUT b
30 LET x=(a=b)*2
40 PRINT x
50 GO TO 10
```

This time you should get a value of 0 for a false expression and 2 for a true expression. The point of all this is that these values of conditional expressions are numbers and can be treated as numbers and this is very useful. There follows a simple game program, BLOB CATCHER, to illustrate the use of what we've just been discussing.

```
  10 REM *BLOB CATCHER*
  20 LET S=0
  30 FOR J=1 TO 9
  40 PRINT AT 10,2*J; INK J/2;J
  50 NEXT J
  60 FOR G=30 TO 1 STEP -1
  70 PRINT AT 5,3; INK 6; PAPER
2; FLASH 1;" TIME>";G;"    SCORE>
";S;" "
  80 LET A=INT (RND*9)+1
  90 PRINT AT 9,2*A; FLASH 1; IN
K 2;"■"
 100 FOR H=1 TO 24
 110 BEEP .01,50/H
 120 LET A$=INKEY$
 130 IF A$<>"" THEN GO TO 170
 140 NEXT H
 150 LET G=G-1
 170 LET S=S+(A$=STR$ A)*A
 180 PRINT AT 9,2*A;" "
 190 NEXT G
 200 PRINT AT 5,3; INK 6; PAPER
2; FLASH 1;" TIME>";0;"    SCORE>
"; PAPER 6; INK 2;S;" "
 210 PRINT AT 13,5; FLASH 1; INK
 0; PAPER 6; BRIGHT 1;"THAT'S AL
L, FOLKS!!"
```

The idea of the game is to press the same key as the number under the moving blob, for instance if the blob lands on 3 then you must press the 3 key, and you will score the number, in this case three is added to your score. The number of attempts you have left is continuously displayed on screen as is your score. Line 170 is the

one we're interested in at the moment. Here if STR$ A (the value of A converted to a string so that it can be compared with the key pressed) is the same as the key pressed then the logical value is 1 because the expression is true. Whatever the value, it is multiplied by the value of A. If 0 then the score does not change. If 1 then the score changes by 1 * A, in other words by A. The score is counted by the variable S. The number of attempts left is counted by F.

Let us now move to look at values in conditional expressions involving the logical operations AND, OR & NOT. X AND Y have the value (X if Y is true/non-zero
(0 if Y is false/zero.

X & Y can be expressions like X = 2 or Y = 2 * B. One common application is to control on-screen movement. Many games use the cursor-arrow keys to control movement on-screen. This is one way of moving an object left or right along the screen:

```
  10 LET X=15
  20 IF INKEY$="5" AND X>1 THEN
LET X=X-2
  30 IF INKEY$="8" AND X<30 THEN
 LET X=X+2
  40 PRINT AT 21,X; INK 2;"■";AT
21,X;" "
  50 GO TO 20
```

This moves a red blob two columns at a time along the bottom row of the screen. You can do the same thing with:

```
  10 LET X=15
  20 LET X=X-(INKEY$="5" AND X>1
)*2+(INKEY$="8" AND X<30)*2
  40 PRINT AT 21,X; INK 2;"■";AT
21,X;" "
  50 GO TO 20
```

Or with:

```
  10 LET X=15
  20 LET X=X-(2 AND INKEY$="5" A
ND X>1)+(2 AND INKEY$="8" AND X<
30)
  40 PRINT AT 21,X; INK 2;"■";AT
21,X;" "
  50 GO TO 20
```

The point to note with the last two programs is that the expressions in brackets take the value of the number before the first AND if all the expressions after the AND are true. Compare these with X AND Y which we have just discussed. Here X is a number (2 in this case) and not an expression. You can think of line 20 above as:

20 LET X = X − (2 if the "5" key is pressed and if the value of X is greater than 1, otherwise 0) + (2 if the "8" key is pressed and the value of X is less than 30, otherwise 0).

You might think why go to these complications to do something that could be done equally well by a series of IF...THEN lines. The answer is that used properly and in the right circumstances you can replace several program lines by one long conditional expression thus saving memory and possibly making the program RUN faster. In addition when you become more familiar with these conditional expressions you will find that sometimes they can actually clarify listings over a long set of IF...THEN statements.

Let's now look at OR in operation.

X OR Y has the value (1 if Y is non-zero/true
(X if Y is zero/false

Suppose a conductor on a bus wanted a program to let him know what fare to charge a schoolchild, and that the age limit for these reduced fares was 14.

```
10 INPUT "Enter fare ";fare
20 INPUT "Enter age ";age
30 LET fare=fare*(0.5 OR age>1
4)
40 PRINT "The fare is ";fare
```

Lines 10 & 20 ask you to enter the normal adult fare and the age of the passenger/commuter. Now then, to understand this a bit easier, let us convert it to plainer English:

LET FARE = FARE * (0.5 unless his or her age is over 14)

If the expression following OR in brackets is true, then the expression in brackets has the value 1. However, if the expression

following OR is false (he or she is 14 or younger) then the expression takes the value before the OR. This number can also be a variable if you like. On its own this routine does not have much to offer against:

30 IF AGE < = 14 THEN LET FARE = FARE *0.5

However, if you had several categories of fares on offer then the method using OR can be extended to evaluate all the categories on one line.

'NOT X' takes the value of 0 if the relation X is true and the value 1 if the relation is false. The best way to illustrate this is with this kind of example:

```
10 INPUT a
20 INPUT b
30 PRINT a;TAB 4;b;TAB 8;NOT a
=b
40 GO TO 10
```

What you will see on the screen are the two numbers you entered in lines 10 & 20, followed by a 0 or a 1. From the results you get, see if you can work out which relationships between A and B produce which value in the third column. Try other relational operators in place of = in line 30 to:-

Finally, let us look at two interesting little oddities. First, consider this line:

```
10 IF a=1 THEN IF b=2 THEN PRI
NT "true"
```

It's to all intents and purposes the same as

```
10 IF a=1 AND b=2 THEN PRINT "
true"
```

except that it requries extra memory. There is a slight difference in that if you haven't previously defined B then the version using AND will crash with report 2. However, if the first part of the other

version is false then the program skips over the remainder of the line. You may be able to find an application for this.

The second oddity is not really an oddity, more something that is missed by many people. Try these programs:

```
10 INPUT a
20 IF a THEN PRINT a
```

```
10 INPUT a
20 IF NOT a THEN PRINT a
```

You might not expect these programs to work because there are no relational operators for comparing A with anything. Here, however, the value of A is considered to be true if it is not zero, or zero if you use NOT. As with everything else in this section, experiment with the examples until you understand exactly what each routine does. You will find that these statements can be very powerful programming facilities, and your programming can be greatly improved as a result.

You can use IF/THEN GO TO to terminate a 'win condition' message after a certain number of cycles. Enter and run the following:

```
10 LET x=0
20 PRINT "You have won ";
30 LET x=x+1
40 IF x<25 THEN GO TO 20
```

This will ensure that YOU HAVE WON is printed out a limited number of times.

As you've seen, IF/THEN is not just used to branch to new lines. NEW the program, and enter the following. You'll see it has a similar effect, although the IF.... is not just sending the program to a line number.

```
10 LET x=0
20 LET x=x+1
30 IF x<25 THEN PRINT "You hav
e won ";
40 GO TO 20
```


This program is not as useful as the other one, as it will not terminate even when it has finished printing out YOU HAVE WON. You can easily discover this by running it, then pressing BREAK, and then PRINT X, ENTER.

It is perhaps worth mentioning that the computer is a fairly dogmatic creature. If you specify that a program branch is to be made only if the value of Z, for example, is equal to 6, the program will continue in a never-ending loop if Z is not exactly equal to 6, no matter how close it is (like 5.999999). If you think the value might be fractionally different from the one you want as a condition for branching, make sure you specify that the relational operator should be, say, greater than 5.5, or greater than or equal to 5.9, rather than just equal to 6.

IF/THEN/ELSE

Many dialects of BASIC include an ELSE option, used in the statement IF...THEN...ELSE. There is no such function in our computer's BASIC, but its logic can be used to emulate this.

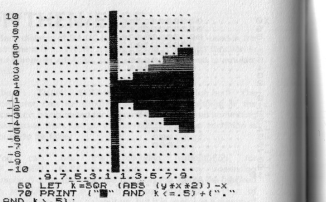
The IF...THEN...ELSE is a very useful variation on IF. The computer can be programmed to do something if the condition being tested for is found to be true, and something else, other than just go to the next line, if the condition is found to be false.

You can use the following substitution for IF...THEN...ELSE to produce some very interesting graphs. You simply enter the function you would like graphed in line 60. This is not the most efficient method of programming on the computers, but it is useful as a means of demonstrating the IF...THEN...ELSE substitution. As the program runs, it evaluates K each time it comes to line 60. Line 70 looks at the value of K and prints a zero if K is greater than or equal to point five, and a full stop if K is less than 0.5. This is the same as a line reading IF K is greater than or equal to point five print "0" ELSE print ".". Each of the other graphs uses different values for K, as generated by line 60. The condition tested for in line 70 also varies. Run the samples given, using your own choice of graphics symbol in line 70, and then create a few of your own. It is likely that you'll have to change the scaling for certain functions.

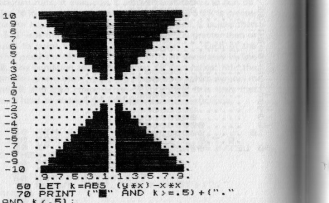
```
  10 REM Graph plotter
  20 FOR y=10 TO -10 STEP -1
  30 IF y<>10 AND y<>-10 AND y>-
1 THEN PRINT "";"";
  40 PRINT y;TAB(4);
  50 FOR x=-10 TO 10
  60 LET k=y-x*x/2+7
  70 PRINT (k<0 AND k>=.5)+("."
AND k<.5);
  80 NEXT x
  90 PRINT
 100 NEXT y
 110 PRINT TAB(4);".9.7.5.3.1.1.3
.5.7.9."
```



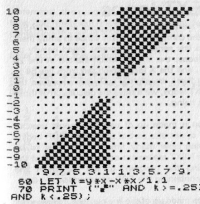
```
10  .....0000000000.....
 9  .....0000000000.....
 8  .....0000000000.....
 7  .....0000000000.....
 6  .....0000000000.....
 5  .....0000000000.....
 4  .....0000000000.....
 3  .....0000000000.....
 2  .....0000000000.....
 1  .....0000000000.....
 0  .....0000000000.....
-1  .....0000000000.....
-2  .....0000000000.....
-3  .....0000000000.....
-4  .....0000000000.....
-5  .....0000000000.....
-6  .....0000000000.....
-7  .....0000000000.....
-8  .....0000000000.....
-9  .....0000000000.....
-10 .....0000000000.....
    .9.7.5.3.1.1.3.5.7.9.

 9  .....0000000000.....
 8  .....0000000000.....
 7  .....0000000000.....
 6  .....0000000000.....
 5  .....0000000000.....
 4  .....0000000000.....
 3  .....0000000000.....
 2  .....0000000000.....
 1  .....0000000000.....
 0  .....0000000000.....
-1  .....0000000000.....
-2  .....0000000000.....
-3  .....0000000000.....
-4  .....0000000000.....
-5  .....0000000000.....
-6  .....0000000000.....
-7  .....0000000000.....
-8  .....0000000000.....
-9  .....0000000000.....
-10 .....0000000000.....
    .9.7.5.3.1.1.3.5.7.9.
60 LET k=3*ABS (y)-x*x
```



```
 60 LET k=SQR (ABS (y*x^2))-x
 70 PRINT (" " AND k<=.5)+("."
AND k>.5);
```

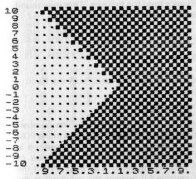


```
 60 LET k=ABS (y*x)-x*x
 70 PRINT (" " AND k>=.5)+("."
AND k<.5);
```

```
 60 LET k=y*x-x*x/1.1
 70 PRINT (" " AND k>=.25)+("."
AND k<.25);
```

Try to work out what line 60 should read to produce this graph.



FOR/NEXT loops

FOR/NEXT loops are additional useful parts of your BASIC working tools on the computer. It makes sense to study them now, because the last series of programs relied heavily on two FOR/NEXT loops, the Y loop which started at line 20 and ended at 100, and the X loop which ran from line 50 to line 80. Because these are slightly more complex than the simplest FOR/NEXT loops, we'll leave the discussion of those alone for the time being.

A FOR/NEXT loop is made up of two statements used to control a series of cycles of a part of a program. FOR begins the loop, specifying how many times the loop is to be executed, and the NEXT statement occurs at the end of the sequence, returning the program to the statement line followed the one containing the FOR command.

FOR statements are made up of the line number, following by the FOR, a numeric variable (a single letter), an equals sign, a numeric expression (a number, or a previously assigned numeric variable), the word TO and finally, another numeric expression (the number of the previously assigned numeric variable) which is different from the first one. That may sound incredibly complicated, but it is really quite simple.

The FOR line reads:

```
100 FOR J = 1 TO 100
```

The NEXT line, which terminates the loop, is of the form:

```
200 NEXT J
```

The NEXT statement then, is made up from a line number, the word NEXT, and the variable set as the control in the FOR statement, earlier in the program. The NEXT sequence is used solely to tell the computer when the sequence of programming which is being repeated is to stop. When the value of the control variable (J) reaches the value set in the FOR statement (the second numeric variable set in the FOR

statement), the program passes through the loop for the final time and then continues with the line following the one containing the word NEXT.

Enter and run this example:

```
10 FOR a=1 TO 10
20 PRINT TAB 4;a;TAB 8;a*a
30 NEXT a

 1   1
 2   4
 3   9
 4   16
 5   25
 6   36
 7   49
 8   64
 9   81
 10  100
```

The control variable is A, and line 20 prints out A and A squared. Note that the limits of the control loop are stated explicitly in line 10 (1 TO 10).

Look at this next example:

```
  10 LET a=5
  20 LET b=16
  30 FOR c=a TO b
  40 PRINT TAB 4;c;TAB 8;c/10;TA
B 14;c/a
  50 NEXT c

    5    0.5    1
    6    0.6    1.2
    7    0.7    1.4
    8    0.8    1.6
    9    0.9    1.8
    10   1      2
    11   1.1    2.2
    12   1.2    2.4
    13   1.3    2.6
    14   1.4    2.8
    15   1.5    3
    16   1.6    3.2
```

Note that in this program the limits of the FOR/NEXT loop are two variables, A and B, which have been previously

defined. You will find there are many programs where you will want a limited FOR/NEXT loop, with the limits being a result of things that have occurred elsewhere in the program.

Nested loops

As you've just seen, a FOR/NEXT loop allows us to alter the value of one variable (by a count of one in the cases we've studied), to repeat a programmed series of events a specified number of times. Now, suppose there were two or more variables to be operated upon. In this case, you would need to vary both values. This can be done quite simply by *nesting* loops, in which one loop, controlled by one set of FOR/NEXT statements, operates within another set.

Enter and run the following program, which *nests* a B loop within an A loop.

```
 10 REM Nested loops
 30 FOR a=1 TO 12
 40 FOR b=1 TO 12
 50 PRINT TAB 8;b;" times ";a;"
is ";a*b
 60 NEXT b
 70 PRINT
 80 POKE 23692,-1
 90 NEXT a
```

When you run this, you'll see it prints out the multiplication table, from 1 × 1, to 12 × 12. Part of the run is:

```
8 times 4 is 32
9 times 4 is 36
10 times 4 is 40
11 times 4 is 44
12 times 4 is 48

1 times 5 is 5
2 times 5 is 10
3 times 5 is 15
4 times 5 is 20
```



```
5 times 5 is 25
6 times 5 is 30
7 times 5 is 35
8 times 5 is 40
9 times 5 is 45
```

In this program, the control variable A stays at one, while the loop controlled by B runs from one to 12. After the PRINT (line 70) the control variable A increases by one, and the B loop runs through again, this time with the A equal to two, and so on, until the B loop has run through with the A equal to 12. There is no reason why you should have only two nested loops.

It is vital that the control variables of nested loops be in the correct order, that is, the first loop begun is the last one to end. Try swapping lines 60 and 90 of this program, and see what happens.

This is part of the output, obviously not what was required.

```
2 times 13 is 26
3 times 14 is 42
4 times 15 is 60
5 times 16 is 80
6 times 17 is 102
7 times 18 is 126
8 times 19 is 152
9 times 20 is 180
10 times 21 is 210
11 times 22 is 242
12 times 23 is 276
```

Use the same variable for as many purposes as you can, especially when you use FOR/NEXT loops. Don't use another letter as the name for a second FOR/NEXT loop if you've already finished with a previous one as this wastes memory.

STEP

For this next discussion, we need the program TABULATOR ROCKET RANGE which was introduced earlier.

The important lines for our discussion at this point are **30**, **40** and **70**. You'll see when you run the program that this causes the numbers **10** down to 1 to appear on the screen. The word STEP (in line **20**) after the 1 controls this. Change the −1 following the word STEP to −2, and see what happens. If no STEP is specified, the computer assumes you want a positive STEP of 1, which is what has been needed in the earlier examples in this section.

The STEP command, then, is used within a FOR/NEXT loop to allow the user to specify the value of the increment (or decrement) of the control variable. The STEP does not have to be a whole number, although you must ensure — if the number which follows the word TO in the initial FOR statement is lower than the number before the TO — that the STEP is negative. Try the following examples:

```
10 FOR a=100 TO 1 STEP -12.5
20 PRINT TAB 5;a
30 NEXT a
```

```
 100
 87.5
 75
 62.5
 50
 37.5
 25
 12.5
```

```
10 FOR a=10 TO 1 STEP -0.175
20 PRINT TAB 5;a
30 NEXT a
```

```
 10
 9.825
 9.65
 9.475
```



```
 9.3
 9.125
 8.95
 8.775
 8.6
 8.425
 8.25
 8.075
 7.9
 7.725
 7.55
 7.375
```

In a FOR/NEXT loop, STEP does not have to be a whole number; it may be a fraction, decimal, the result of a calculation and does not have to hit the limit value of the loop exactly. It carries on looping as long as it is less than or equal to the limit. You cannot easily change the value of STEP during the course of a loop. If the limit value has already been exceeded then the loop will be totally bypassed:

```
10      FOR F = 1 TO 0
20      PRINT "X"
30      NEXT F
```

You may be able to use this idea to prevent loops being executed if certain conditions exist, e.g. if you didn't want a black line to be drawn if X was equal to 6:

```
1000    FOR F = (X = 6) * 33 TO 31
1010    PRINT CHR$ 143;
1020    NEXT F
```

The test for whether the limit value has been exceeded is made at the line containing the FOR statement. An interesting experiment is to try a STEP value of **0**. The control variable is never incremented and so the loop never ends! You can jump out of FOR/NEXT loops without any problems, but you cannot jump into a loop unless the control variable has already been set up (effectively if you've used that loop before). In a FOR/NEXT loop, the loop jumps from NEXT to the line following the FOR statement. Some versions of BASIC allow you to omit the variable after NEXT and the most recent control variable is then incremented; you must specify the control variable on your computer.

GOSUB and RETURN

A subroutine is a block of program within a larger program which performs one specific task. The main program is executed, line by line, until the subroutine is called, by the GOSUB command. The computer goes to the specified number, works through in line order from that point, until it hits the word RETURN. This is the signal for the computer to return to the main program, to the line *after* the one which sent it to the subroutine.

A subroutine is useful if a particular set of calculations has to be carried out a number of times within a program, and at different places within the program. For example, in a financial program, there may be a number of tax calculations to be carried out at different points within the program. Whenever this need arises, the program is told to GOSUB, and it stays in this subroutine until it hits the word RETURN, when it returns to the line *after* the GOSUB command.

A subroutine is written exactly like the main program, except that it is a program within a program, and is bounded by two lines, one containing the GOSUB and the other the RETURN line. The GOSUB command is made up from a line number, followed by the word GOSUB, and another line number. The line **40** GOSUB **100** tells the computer to branch to line **100** and continue executing the program in order, just as if line **40** had said GOTO **100**. However, when the program reaches a line containing the word RETURN, the action reverts to the main program, at the line number which follows the one containing the GOSUB statement (in this case, the first line number after **40**).

A simple example, showing GOSUB and RETURN, follows. Enter and run it a few times, then come back to the book for a discussion on it.

```
Your number is 234
234 squared is 54756

Your number is 23.76
23.76 squared is 564.5376
```



```
Your number is 4
4 squared is 16

Your number is 33
33 squared is 1089
```

```
  10 REM GOSUB/RETURN DEMO
  20 POKE 23609,100: REM
   ADDS BEEP ON KEY PRESS
  30 INPUT "Enter a number ";A
  40 GO SUB 100
  50 GO TO 30
  90 REM SUBROUTINE FOLLOWS
 100 PRINT '"Your number is ";A
 110 PRINT A;" squared is ";A*2
 120 RETURN
```

Line **30** asks you to enter a number, then line **40** transfers control to the subroutine starting at line **100**. The required calculations are carried out, and the results of them printed, within the subroutine, then line **120** returns control to the line *after* the one which sent control to the subroutine, that is line **50**. As line **50** is a GOTO, action goes back to line **30**, where a new number is requested, and the whole merry dance begins again.

Enter and run the following program, which pits two submarines against each other in a race, to see a subroutine doing something a little more interesting than in the preceding program.

```
  10 REM GOSUB RACE
  15 PAPER 5: BORDER 5: CLS
  20 LET A$="
                    "
  30 LET COMPUTER=28
  40 LET HUMAN=28
  50 LET X=5
  55 BEEP .01,RND*50
  60 GO SUB 100
  70 LET X=10
  75 BEEP .01,RND*50
  80 GO SUB 100
  90 GO TO 50
 100 IF X=5 THEN LET COMPUTER=CO
MPUTER-RND: PRINT AT X,COMPUTER;
 INK 6;A$: IF COMPUTER<2 THEN PR
INT AT 0,0; PAPER 6; FLASH 1; BR
IGHT 1;"COMPUTER WINS": STOP
```

```
 110 IF X=10 THEN LET HUMAN=HUMA
N+RND: PRINT AT X,HUMAN; INK 2;A
$: IF HUMAN<2 THEN PRINT AT 0,0;
 PAPER 6; FLASH 1;"HUMAN WINS":
STOP
 120 RETURN
```

There are two 'submarines' on the screen. The top one is the computer's, and the bottom one belongs to you. You just press RUN, then ENTER, and the submarines move across the screen from right to left. When one or the other reaches the side, the program stops, printing out COMPUTER WINS or HUMAN WINS, as the case may be.

Note that A$, the submarine, extends over more than one line. Just keep pressing SPACE over and over again once you've put in the 'periscope' part of the picture. Note also that there is a space *after* the end of the submarine. This is vital, as you'll discover if you leave it out.

Sound

The BEEP command on the Spectrum can be used to enhance your programs, adding appropriate noises when aliens are zapped, balls bounce off walls, or you manage to defeat the computer in a game of skill.

Although it may appear at first sight that a single-voice, fairly quiet sound channel, which stops all other computer action while it is sound, is limited, it can still be used to add a surprising degree of interest to your programs.

We have connected a small extension speaker to our Spectrum, plugging it into the EAR socket. Although this does not increase the volume very much, having a second source for the sound definitely improves its effectiveness.

The BEEP command (both SHIFT keys, then the red SHIFT key held down as you press Z) has two parameters (that is, it

has two numbers after the word BEEP). The first number is the length of the note it plays, and the second number is its pitch.

You'll get an idea of the kind of noises it can make by running this 'Random music' routine:

```
  5 REM Random music
 10 BEEP RND/RND/3,RND*60-35
 12 BORDER RND*7
 15 BEEP RND/RND/2,RND*80-45
 20 BORDER RND*7
 25 BEEP RND/RND/3,RND*130-65
 30 PAPER RND*7
 40 CLS
 45 BEEP RND/RND/2,RND*40-5
 50 GO TO 10
```

The result of using BEEP within a loop, or a couple of loops, can be very interesting, as this demonstration shows:

```
  10 REM LOOPING MUSIC
  20 FOR A=-60 TO 60
  30 FOR B=.01 TO .03 STEP .01
  40 BEEP B,A: BEEP B,A/10+B: BE
EP B,ABS (A)
  50 NEXT B
  60 NEXT A
```

Randomly produced BEEPs, if held within limits, can also be interesting:

```
  10 LET PITCH=INT (RND*24)-12
  20 LET DURATION=(INT (RND*20)+1
)/30
  30 BEEP DURATION,PITCH
  40 IF RND>=.7 THEN GO TO 30
  50 GO TO 10
```

As you know from your manual, certain pitch numbers produce certain notes, with — for example — zero for Middle C, 12 for the C an octave above it, and −3 for two notes below it. The next program turns your ZX Spectrum into a 'vibraphone' in which you use the bottom row of keys to produce the notes (in the key of C):C (Z key), D(X), E(C), F(V), G(B), A(N), B(M), C(K). The notes will continue to sound so long as you hold the key down.

```
  10 REM PIANO
  20 LET A=CODE INKEY$
  30 IF A<66 OR A>90 THEN GO TO
20
  40 LET B=12*(A=75)+2*(A=88)+4*
(A=67)+5*(A=86)+7*(A=66)+9*(A=78
)+11*(A=77)
  50 BEEP .04,B
  60 IF INKEY$<>"" THEN GO TO 50
  80 GO TO 20
```

Once you've mastered that, you can add a little colour to go along with the pitch of the notes, with this variation (line 70 has been added to the preceding program).

```
  10 REM PIANO
  20 LET A=CODE INKEY$
  30 IF A<66 OR A>90 THEN GO TO
20
  40 LET B=12*(A=75)+2*(A=88)+4*
(A=67)+5*(A=86)+7*(A=66)+9*(A=78
)+11*(A=77)
  50 BEEP .04,B
  60 IF INKEY$<>"" THEN GO TO 50
  70 BORDER B/2: PAPER B/2: CLS
  80 GO TO 20
```

If you find playing the Spectrum vibraphone too much work, you can get the computer to do it all. As you'll see from the DATA line, the 'well-tempered Spectrum' uses the natural scale of C.

```
  10 REM The Well-tempered
          Spectrum
  20 DIM A(8)
  30 FOR B=1 TO 8
  40 READ A(B)
  50 NEXT B
  70 LET B=INT (RND*8)+1
  75 LET M=(INT  (RND*4)+1)/10
  80 BEEP M,A(B)
  85 IF RND>.9 THEN GO SUB 110
  90 GO TO 70
 100 DATA 0,2.039,3.86,4.98,7.02
,9.04,10.86,12
 110 LET Z=RND
 112 LET M=A(1)*(Z>=.5)+A(8)*(Z<
.5)
 115 BEEP 1,M
 120 PAUSE 25
 130 RETURN
```


You will see many examples of use of BEEP in the programs in this book, and these should give you ideas of sounds which you can add to your own programs.

The final program in this section shows a program in which the sound is an important ingredient, and not just an afterthought. In this variation of SIMON, the computer picks a number between one and four, and puts it on the screen, with a relevant BEEP and a burst of colour. You have to press the same number. The computer will then repeat its first number, and add a second one. You have to repeat both numbers, in sequence. If the computer selects the same number twice in a row, you have to press it twice, taking your finger off the key briefly before doing so. You win the game if you manage to remember a sequence of seven digits correctly.

```
   1 REM ___Simon___
   5 LET a$=""
  10 FLASH 1
  20 PAPER 7
  30 FOR a=1 TO 7
  35 PRINT AT 10,10;"Please stan
d by"
  36 BORDER RND*7
  40 LET a$=a$+STR$ (INT (RND*4)
+1)
  45 PAUSE 5
  50 NEXT a
  55 FLASH 0: CLS
  60 LET x=1
  70 FOR q=1 TO x
  72 LET l=4*(CODE a$(q)-48)
  73 LET t=VAL a$(q)
  75 BEEP .05,10*t
  80 PRINT AT l,7; INK t;"▀▀▀";"
";a$(q);AT l+1,7;"▀▀▀";AT l-1,7
;"▀▀▀"
  85 BORDER RND*7
  90 PAUSE 20-x
 100 PRINT  AT l,7; INK 6;"▚▚▚";
AT l+1,7;"▚▚▚";AT l-1,7;"▚▚▚"
 102 PAUSE 4
 105 CLS
 110 NEXT q
 120 FOR b=1 TO x
 122 IF INKEY$<>"" THEN GO TO 12
2
 130 LET t$=INKEY$
 124 IF CODE t$=0 THEN GO TO 123
 125 CLS
 126 LET y=4*(CODE t$-48)
 130 PRINT AT y,7; INK y/4;"▀▀▀"
;t$;AT y-1,7;"▀▀▀";AT y+1,7;"▀▀▀
```

```
 145 BEEP .04,2.5*y
 146 IF CODE (a$(b)) TH
EN GO TO 300
 147 PAUSE 7
 148 CLS
 150 NEXT b
 155 IF x=7 THEN PRINT "You win!
": BORDER RND*7: PAPER RND*7: GO
TO 155
 160 LET x=x+1
 165 PAUSE 50
 170 GO TO 70
 300 PRINT "You scored ";x-1
 310 BORDER RND*7
 320 PAPER RND*7
 330 CLS
 335 BEEP .02,RND*30
 340 GO TO 300
```

DEFining functions

This feature allows you to DEFine functions within a program, which you can then call whenever you need to while running the program. DEF FN can save space as well as time, as complex calculations can be defined with a short name, and called up at will by use of this name.

There are four things in the statement which define the function:

- The word DEF.
- The name of the function, which consists of the letters FN, followed by the name (a letter if it is a numerical function, a letter and a $ if it is a string function).
- The argument of the function which follows the name, in brackets.
- The formula, using the argument, for working out the function.

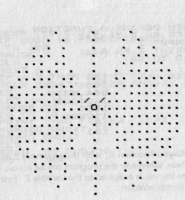
This sounds a lot more complicated than it is in practice. Look at this program.


```
 10 REM DEFINE A FUNCTION
 20 DEF FN A(Z)=Z*Z
 30 INPUT Z
 40 PRINT Z,FN A(Z)
 50 GO TO 30

12              144
44              1936
23.76           564.5376
1111            1234321
44              1936
345             119025
4               16
11              121
```

Line 20 defines a function A, with the argument Z as being Z squared. Then, whenever the program comes across FNA(Z), it will square the value assigned to the variable Z. You can see this in the demonstration.

In the next program — BAT — a function is defined in line 70. The function gets the square root of the difference between the square of two variables, and in the routine 120 to 210, uses the value H (see line 130) to determine the printing positions of the dots which will draw up the bat.

```
 10 REM BAT
 20 REM SHOWING DEF FN
 30 LET L=8
 40 LET P=10
 50 LET Q=17
 60 LET B=0
 70 DEF FN A(B)=SQR (L*L-B*B)
 80 PAPER 6: CLS
 90 INK 0: BORDER 2
100 PRINT AT P,Q;"O"
110 LET L=L+1
120 FOR B=0 TO L
130 LET H=FN A(B)
140 PRINT AT P+B,Q+H;"."
150 PRINT AT P+B,Q-H;"."
160 PRINT AT P-B,Q-H;"."
190 PRINT AT P-B,Q+H;"."
200 BEEP .02,50+H
210 NEXT B
220 IF L<11 THEN GO TO 110
230 PRINT INK 2;AT 9,16;"/ /"
```



DIM and arrays

The DIM statement is used to set up a *list* which you can easily access. You may find it necessary, in some programs, to refer to elements of a long list of numbers, such as if you INPUT a quantity of DATA, and you wish to use it in a certain way, such as PRINTing it in order or magnitude.

An array is a set of memory spaces reserved in the computer, and referred to by the name of the array, and by a subscript. To produce an array to hold three elements, you enter DIM A(3) which creates spaces for an array called A. To hold four elements, you enter DIM B(4).

Enter and run the following program which should make it a little easier to understand.

```
10 REM **ARRAYS DEMO**
20 DIM B(4)
30 FOR A=1 TO 4
40 LET B(A)=INT (RND*9)+1
50 NEXT A
60 FOR A=1 TO 4
```



```
70 PRINT TAB 6;"B(";A;") IS
B(A)
80 NEXT A

       B(1) IS 6
       B(2) IS 9
       B(3) IS 7
       B(4) IS 1
```

As you can see from line **20** of the program you've just run, the computer needs you to DIMension an array before you can use it, with a DIM statement. The DIM statement is made up of a line number followed by the word DIM, and the name of the array, with the size of the array enclosed in brackets.

The arrays we've been talking about so far are one-dimensional arrays, suitable for such things as holding a list of numbers. However, you can have arrays of more than one dimension. These arrays are called, reasonably enough, multi-dimensional arrays, and are set up with a DIM command having more than one subscript. Enter and run the following program:

```
10 REM MULTI-DIMENSIONAL
20 REM          ARRAYS
30 DIM A(4,4)
40 FOR B=1 TO 4
50 FOR C=1 TO 4
60 LET A(B,C)=INT (RND*9)+1
70 PRINT "A(";B;",";C;") IS ";
A(B,C)
80 NEXT C
90 NEXT B
100 PRINT AT 16,15;"1 2 3 4"
105 PRINT
110 FOR B=1 TO 4
120 PRINT TAB 13;B;TAB 15;A(B,1
);" ";A(B,2);" ";A(B,3);" ";A(B,
4)
130 NEXT B
```

When you run it you'll see something like this:

```
A(1,1) IS 9
A(1,2) IS 8
A(1,3) IS 7
A(1,4) IS 4
A(2,1) IS 3
A(2,2) IS 3
```

```
A(2,3)  IS 4
A(2,4)  IS 6
A(3,1)  IS 9
A(3,2)  IS 8
A(3,3)  IS 2
A(3,4)  IS 8
A(4,1)  IS 7
A(4,2)  IS 8
A(4,3)  IS 6
A(4,4)  IS 1
```

	1	2	3	4
1	9	8	7	4
2	3	3	4	6
3	9	8	2	8
4	7	8	6	1

Firstly the elements of the array will be filled with numbers between one and nine, and these are printed out by line 70 so you can see what is held by each element of the array. The little table printed beside them shows how the elements of the array are organised. Any element can be accessed by giving its coordinates within the array. If this is so, element 3, (3) should lie where the two threes intersect, i.e. on the number 2. You'll see from looking above in our sample run that, in fact, A(3,3) does equal 2.

DIMensioning an array consumes memory, so do not set up an array larger than you need. The number of elements in an array is the first number within the brackets, multiplied by the second number. Therefore, the array A (4,4) has 16 (4x4) elements. You can see from our sample run that this is so.

There is no reason why you should not have arrays with more than two dimensions, except for the fact that they can quickly become quite difficult to handle, and the number of elements rockets quite alarmingly. Here is a program to DIMension and fill a three-dimensional array. Although the array is only A(3,3,3), you can see the number of elements is quite large (3*3*3).

```
 10 REM MULTI-DIMENSIONAL
 20 REM       ARRAYS
 30 DIM A(3,3,3)
 40 FOR B=1 TO 3
 50 FOR C=1 TO 3
 55 FOR D=1 TO 3
```



```
 60 LET A(B,C,D)=INT (RND*9)+1
 70 PRINT "A(";B;",";C;",";D;")
IS ";A(B,C,D)
 75 NEXT D
 80 NEXT C
 90 NEXT B
```

```
A(1,1,1)  IS 5
A(1,1,2)  IS 3
A(1,1,3)  IS 3
A(1,2,1)  IS 5
A(1,2,2)  IS 7
A(1,2,3)  IS 8
A(1,3,1)  IS 9
A(1,3,2)  IS 2
A(1,3,3)  IS 2
A(2,1,1)  IS 5
A(2,1,2)  IS 6
A(2,1,3)  IS 5
A(2,2,1)  IS 8
A(2,2,2)  IS 4
A(2,2,3)  IS 2
A(2,3,1)  IS 1
A(2,3,2)  IS 8
A(2,3,3)  IS 6
A(3,1,1)  IS 4
A(3,1,2)  IS 8
A(3,1,3)  IS 9
A(3,2,1)  IS 3
A(3,2,2)  IS 8
A(3,2,3)  IS 6
A(3,3,1)  IS 4
A(3,3,2)  IS 7
A(3,3,3)  IS 9
```

Increase the number of dimensions to five, as in our next example, and although it is only A(2,2,2,2,2), there are now 32 (2*2*2*2*2) elements.

```
A(1,1,2,1,1)  IS 9
A(1,1,2,1,2)  IS 6
A(1,1,2,2,1)  IS 4
A(1,1,2,2,2)  IS 3
A(1,2,1,1,1)  IS 7
A(1,2,1,1,2)  IS 5
A(1,2,1,2,1)  IS 4
A(1,2,1,2,2)  IS 6
A(1,2,2,1,1)  IS 4
A(1,2,2,1,2)  IS 4
A(1,2,2,2,1)  IS 4
A(1,2,2,2,2)  IS 9
A(2,1,1,1,1)  IS 6
A(2,1,1,1,2)  IS 4
A(2,1,1,2,1)  IS 5
A(2,1,1,2,2)  IS 7
```

```
A(2,1,2,1,1) IS 1
A(2,1,2,1,2) IS 7
A(2,1,2,2,1) IS 0
A(2,1,2,2,2) IS 8
A(2,2,1,1,1) IS 6
A(2,2,1,1,2) IS 2
```

```
  10 REM MULTI-DIMENSIONAL
  20 REM         ARRAYS
  30 DIM A(2,2,2,2,2,2)
  40 FOR B=1 TO 2
  50 FOR C=1 TO 2
  60 FOR D=1 TO 2
  70 FOR E=1 TO 2
  80 FOR F=1 TO 2
  90 LET A(B,C,D,E,F)=INT (RND*9
)+1
 100 PRINT "A(";B;",";C;",";D;",
";E;",";F;") IS ";A(B,C,D,E,F)
 110 NEXT F
 120 NEXT E
 130 NEXT D
 140 NEXT C
 150 NEXT B
```

Code-breaker

Here is the game CODE-BREAKER to show a single-dimensional array in use. The game is simple to play. The computer 'thinks of' a four-digit number, and you have ten guesses to work it out. A correct digit in the wrong position within the code gives you a 'white', while a correct digit in the correct position gives you a 'black'.

```
  10 REM CODE-BREAKER
  20 DIM C(4)
  30 DIM G(4)
  40 PRINT '' PAPER 2; INK 6;"
    I am thinking of a 4-digit"
  50 PRINT PAPER 2; INK 6;"numbe
r, which you have 10 goes"
  60 PRINT PAPER 2; INK 6;"to di
scover. All 4 digits are "
  70 PRINT PAPER 2; INK 6;"diffe
rent.  Press any key
```



```
  80 PRINT PAPER 2; INK 6;"
    to begin..."
  90 PAUSE 464
 100 CLS
 110 LET C(1)=INT (RND*9)+1
 120 FOR Z=2 TO 4
 130 LET C(Z)=INT (RND*9)+1: BEE
P .5,C(Z)
 140 FOR J=1 TO Z-1
 150 IF C(J)=C(Z) THEN GO TO 130
 160 NEXT J
 170 NEXT Z
 180 FOR G=1 TO 10: BEEP 1,2*G
 200 PRINT PAPER 1; INK 7;TAB 8;
"Enter guess ";G;" "
 210 INPUT A
 220 LET A1=A
 230 FOR Z=1 TO 4
 240 LET G(Z)=A-10*INT (A/10)
 250 LET A=INT (A/10)
 260 NEXT Z
 270 LET B=0
 280 LET U=0
 290 FOR Z=1 TO 4
 300 IF C(Z)<>G(Z) THEN GO TO 33
0
 310 LET B=B+1: BEEP .02,10+B
 320 LET G(Z)=0
 330 NEXT Z
 340 FOR Z=1 TO 4
 350 IF G(Z)=0 THEN GO TO 400
 360 FOR J=1 TO 4
 370 IF C(Z)<>G(J) THEN GO TO 39
0
 380 LET U=U+1: BEEP .02,7+U
 390 NEXT J
 400 NEXT Z
 420 PRINT INK 2;A1," SCORED ";
INK 0;B;" BLACK";
 430 IF B<>1 THEN PRINT "S";
 440 PRINT " and "; INK 2;U;" WHITE
";
 450 IF U<>1 THEN PRINT "S";
 460 PRINT
 470 IF B=4 THEN PRINT "You gues
sed in ";G;" guesses!!"
 480 IF B<4 THEN NEXT G
 490 PRINT "The code was "; INK
2;C(4);C(3);C(2);C(1)
```

String arrays

You can also have string arrays, which are very similar to numeric arrays. Enter and run the following program to see the string array in practice, entering four words (each followed by ENTER), when prompted.

```
 10 REM STRING
 20 DIM A$(4,10)
 30 FOR B=1 TO 4
 40 INPUT A$(B)
 50 NEXT B
 60 FOR B=1 TO 4
 70 PRINT "A$(";B;") IS ";A$(B)
 80 NEXT B
```

```
A$(1) IS WATER
A$(2) IS REASON
A$(3) IS WASTE
A$(4) IS ENGLAND
```

Although the second number in the DIM statement (in this case **10**) has to be as long as the longest string you intend to enter, you only need to specify the first element (as in line **70**) to get the full string to print out.

Note that the main difference between a string array and a numeric array is the dollar sign immediately following the letter. This tells the computer the name refers to a string.

Here's a string sort program to show string arrays in use. As set up, and as demonstrated in the sample run, the program caters for five words. To adapt it for more, change the 5 in lines **20**, **50**, and **40** to the number of words you need to sort.

```
 10 REM STRING SORT
 20 DIM U$(5,10)
 30 LET B=0
 40 LET G=5
 50 FOR A=1 TO 5
 60 INPUT U$(A)
 70 PRINT U$(A)
 80 NEXT A: PRINT
 90 LET Z=1
100 LET B=Z+1
```



```
110 IF B>G THEN GO TO 190
120 IF U$(B)>U$(Z) THEN GO TO 1
80
130 LET Z=Z+1
140 GO TO 100
150 LET Q$=U$(Z)
160 LET U$(Z)=U$(B)
170 LET U$(B)=Q$
180 GO TO 130
190 PRINT U$(G)
200 LET G=G-1
210 IF G>0 THEN GO TO 90
```

```
RUN
RANDOM
RAZZLE
RUNNER
RANSACK

RANDOM
RANSACK
RAZZLE
RUN
RUNNER
```

String handling

Our discussion of string arrays leads us neatly into strings, and string handling. As you've probably realised by now, a string is a collection of alphanumeric characters within quote marks (including symbols and spaces, if desired). It is assigned to a variable whose name is a letter, followed by a dollar sign. Strings are assigned in much the same way as are numeric variables, by a statement of the form LET A$ = "HI".

There are a number of very useful string functions, which can be used for manipulation of strings, and for extracting parts of the strings. The functions are:

CODE X$ —This gives the character code of the first character in X$, so if X$ equalled MICRO, CODE X$ would give 77.

CHR$ 77 — We can check to see if, in fact, 77 is the code of the first letter of X$ (i.e., if it is the code of M) by asking the computer to PRINT CHR$ 77. This gives an M. In effect, CHR$ is the opposite of CODE, and turns a code back into a character.

X$(TO 3) — This gives a string containing the n left most characters of X$, so X$ (TO 3) will give "MIC".

LEN X$ — This function gives the length of a string, so using our string, X$, of "MICRO", we get LEN X$ of 5.

X$(n to m) — This string function produces a string from X$ which lies between characters n and m, starting from character number n. X$ (2 TO 4) gives "ICR".

X$(3 TO) — This function is the opposite, as may be expected, of X$ (to n), and gives the n rightmost characters in the string. X$ (3 TO) gives "CRO".

STR$ A — This turns a variable A into a string, so if the variable was 234, the string version would be "234". This may not seem to be much use, but allows certain manipulation of numbers when they are strings which would be extremely difficult in their numeric form. We look at STR$ in more detail shortly.

VAL X$ — This is the 'opposite' of STR$ A and takes the numeric value of the string and turns it into a number. Thus VAL X$, where X$ equals "22+34", would return 56.

Here is a printout from the computer showing the string functions in operation.

```
 10 PRINT "LET X$="MICRO""
 20 LET X$="MICRO"
 30 PRINT "CODE X$=";CODE X$
 40 PRINT "CHR$ 77=";CHR$ 77
 50 PRINT "X$(3 TO )=";X$(3 TO
)
 60 PRINT "X$( TO 3)=";X$( TO 3
)
```



```
 70 PRINT "LEN X$=";LEN X$
 80 PRINT "X$(2 TO 4)=";X$(2 TO
4)
 90 PRINT "LET X$="23+35""
100 LET X$="23+35"
110 PRINT "VAL X$=";VAL X$
120 PRINT "LET X=34"
130 LET X=34
140 PRINT "LET X$=STR$ X"
150 LET X$=STR$ X
160 PRINT "X$=";X$

LET X$="MICRO"
CODE X$=77
CHR$ 77=M
X$(3 TO )=CRO
X$( TO 3)=MIC
LEN X$=5
X$(2 TO 4)=ICR
LET X$="23+35"
VAL X$=58
LET X=34
LET X$=STR$ X
X$=34
```

Using LEN

If you wish to PRINT a certain number of a particular character, for example if you want to draw a line of "–" characters for underlining, then here are two methods. Obviously, different headings will be of different lengths, so you need to know how many characters to PRINT. If you're printing a string, such as A$, you use the function LEN to tell you the length of A$, hence this is the amount of characters to PRINT.

(1)
```
10 FOR A = 1 TO LEN A$
20 PRINT "–";
30 NEXT A
40 PRINT
```

Line 40 moves the PRINT position to the next line ready to continue. Omit it if you do not need it. The next method is a lot faster and uses only one program line.

(2) 10PRINT"_____"(TO LEN A$)

The only disadvantage is that you need to specify how many characters are required in quotes even though they may never be printed. That is, you need to know the longest that A$ can possibly be so that you can put that many characters in the string constant in quotes after PRINT.

Using STR$

STR$ is a very useful and often neglected function. As we mentioned a few pages ago, it converts a number into its string equivalent, as it would appear when PRINTed on the screen. Try this program:

```
10 PRINT 2
20 PRINT STR$ 2
30 PRINT 1/3
40 PRINT STR$ (1/3)
50 PRINT 9E15
60 PRINT STR$ 9E15
```

You should get these results:

```
 2
2
 .33333333
".33333333
 9E+15
"9E+15
```

We can learn a lot from these examples. Firstly, the string generated by STR$ is the same as you would get if you PRINTed the number on the screen. Secondly, numbers of less than 1 are assigned to a string with a zero before the decimal point, providing the first digit after the decimal point is anything by 0 (i.e. the number is equal to or greater than 0.1 and less than 1) and there may be up to eight digits after the decimal point, although there may be less if all are not

required. So there may be up to ten characters in the entire string. However, if the number to which STR$ is applied has more than 8 digits after the decimal point, it is rounded off to 8 decimal places, e.g. STR$ .333333339 is "0.33333334". STR$ is also capable of generating scientific notation (which you'll recall, we discussed earlier) such as 9E + 15. Note that although the computer accepts 9E15, STR$ assigns it as 9E + 15 — that is, the exponent part is always signed. Very small numbers, e.g. 0.000009 are assigned thus: STR$ 0.00000009 is "9E−7". When using STR$, it is often wise to limit the values of the number so that STR$ does not begin to use scientific notation, which will cause problems.

You are by now almost certainly thinking: fine, but what can you do with it?

The main use is to convert numbers to strings so that we can apply the computer's string handling facilities for formatting or rounding off to a given number of decimal places or other purposes for where you need to be able to assess a number digit by digit. Here are a few examples of the application of STR$.

(i) Lining up decimal points. Suppose you had a list of numbers to print. Try this program:

```
10 LET a=RND*100
20 PRINT a
30 LET a=a*10
40 GO TO 20
```

You should get something like this:

```
 .5315143
 55.31543
 558.1543
 5581.543
 55815.43
 558154.3
 5581543
 55815430
```

It would be much more legible and readable if we could line up the decimal point and this is often very useful. Try this routine:

```
          79.025269
         790.25269
        7902.5269
       79025.269
      790252.69
     7902526.9
 10 LET a=RND*100
 20 PRINT TAB 15-LEN STR$ INT a
;a
 30 LET a=a*10
 40 GO TO 20
```

This spaces eveything out so that the decimal points appear beneath each other — useful for a chart or list of numbers where you may wish to quickly compare several numbers. Can you see how the program works? Suppose the value of A was 69.433594. What the program does is take the integer part of A (INT A, which is 69), converts this to a string (STR$ INT A) then measures the length of this string (LEN STR$ INT A) which in this case is 2. It then uses this number to work out how far back across the screen to start PRINTing the value of A. Note how this is done:

TAB 15 — LEN STR$ INT A

This means that 15 is how far across the screen the decimal point is placed, and then it counts back by the number of digits in STR$ INT A.

(ii) PRINT to a given number of decimal places. It is often necessary to PRINT to, say, three decimal places, You will remember that the above example printed numbers with all the digits known. We can use STR$ to regulate how many numbers are PRINTed after the decimal point. Consider this routine:

```
     0.189
     1.693
    16.934
   189.346
  1893.463
```



```
 10 LET a=RND
 20 LET a$=STR$ a
 22 IF a$(1)="." THEN LET a$="0
"+a$
 25 LET b=LEN STR$ INT VAL a$
 27 PRINT TAB 15-b;(a$+(".",
b=LEN a$)+"000") ( TO b+4);
 30 LET a=a*10
 40 GO TO 20
```

This will PRINT to three decimal places, adding both leading zeros (0 at a beginning) and trailing zeros (0 at end) if required. To get it to PRINT to Z decimal places, make the following changes to line 27; add as many spaces to A$ as the number of decimal places you require (i.e. Z zeros), and you should make the slicer statement (TO B + 1 + Z).

Here is a line by line explanation:

Line 10 sets the value of A to start off with.

Line 20 converts A to a string

Line 22 adds a zero before the first digit if it is a decimal point. Unfortunately, the STR$ function is not uniform in its action in that it sometimes supplies a leading zero for numbers less than 1 and sometimes doesn't, depending on whether the first digit after the decimal point is 0. Therefore it is a simple matter to check if a zero is required or not — if the first character is a decimal point, add a zero.

Line 25 makes B equal to the number of digits in the integral part of the number we're PRINTing by measuring the length of the integral part of the string A$. It is necessary to use VAL A$ rather than A because the computer may have an anomaly between the last digit of the value of A and A$ as set up by STR$, which may in the odd case cause a problem.

Line 27 which spaces the PRINT position as in (i) above, then sets about PRINTing A$ to 3 decimal places. First of all, A$ is PRINTed completed, then a decimal point is added if A$ already represents a whole number and enough zeros to make up three decimal places. You may be wondering why add 4 to B — surely we're PRINTing to three decimal places? Remember the decimal point — that's an extra character. For the purpose of the slicing statement, the part before is treated

as one long string provided it is all in brackets. All we've done is add characters to pad out A$ to at least three decimal places, then PRINT up to 3 digits after the decimal point. This routine does not round off the third decimal place.

(iii) Saving memory. It is often possible to save memory by using strings to hold numbers, instead of numeric variables and decode them later using VAL. You can store the number in a string variable using STR$:

LET A$ = STR$(1024)

and decode them later as required using VAL:

PRINT VAL A$

You will often find that you use up more memory in converting numbers in this fashion than you would in using proper numeric variables, but sometimes this method can work wonders.

Try applying VAL to an expression like "ATN 1 x 4". It works, and this is often quite a useful facility. You can have the name of a numeric variable in quotes and provided it has previously been defined or assigned, it will be successfully evaluated. In fact VAL can be applied to all sorts of numeric expressions, and is sometimes used in place of DEF FN.

It may also be useful if you wish to generate random numbers several times in a program. At the start of the program have a statement line LET A$ = "RND * 6" and every time you want a random number you would type LET R = VAL A$.

INKEY$

You do not need to press ENTER after pressing a key when INKEY$ is used, as our next program makes clear.

Try the following. Enter a number from one to nine, by pressing the key of that number, and you'll see it print out YOU PRESSED 6, YOU PRESSED 1 and so on. Touch the zero key to end, and it will print out YOU PRESSED 0 and then stop.

```
10 REM ** INKEY$ DEMO **
20 PAUSE 100
40 LET A$=INKEY$
50 PRINT "YOU PRESSED ";A$
60 IF A$="0" THEN STOP
70 GO TO 20
```

```
YOU PRESSED 6
YOU PRESSED 7
YOU PRESSED 3
YOU PRESSED 5
YOU PRESSED 8
YOU PRESSED 6
YOU PRESSED 3
YOU PRESSED 6
YOU PRESSED 5
YOU PRESSED 4
YOU PRESSED 6
YOU PRESSED 5
YOU PRESSED 6
YOU PRESSED 0
```

The next program — PREDICTION — also uses INKEY$. In this game, you have to try and anticipate the number (from one to nine) the computer will think of next. The computer's number is shown on the screen near the middle, and the lowest number is the score. The lower the score at the end (when you manage to successfully predict the computer's number), the better. The screen will stay blank until you press a key. The words "THE SCORE IS" will flash off and on.

```
YOUR NUMBER IS 4

MY NUMBER IS 1
THE SCORE IS 1
```

```
 10 REM **PREDICTION**
 20 LET S=0
 40 LET O=0
 50 PAUSE 100
 70 LET A$=INKEY$: INK RND*7: PA
PER 9;"      YOUR NUMBER IS ";A$;"
 "
 90 LET O=O+1: BEEP .02,0*3
 110 LET U$=STR$ (INT (RND*9)+1)
 120 PRINT AT 12,6; INK RND*7;
PAPER 9;"    MY NUMBER IS ";U$;
 "
 130 PRINT AT 14,6;: INK RND*7;
PAPER 9; FLASH 1; BRIGHT 1;"
THE SCORE IS ";O;"  "
 135 STOP
 140 IF U$=A$ THEN BEEP .01,RND*
30: GO TO 130
 150 GO TO 50
```

The next program — MAZE MAKER — also shows INKEY$ in action. Using the "A", "Z", "K" and "M" keys, you have to move the $ sign from the bottom left-hand corner to the top right-hand one, without crossing any of the little white squares. Note that no path through is guaranteed, and there is no mechanism for checking that you don't cheat. At the end, the number of 'moves' it took you is printed on the board.

```
10 REM **MAZE MAKER**
20 REM USE THE A Z N K KEYS
30 REM TO MOVE THE $ SIGN
40 REM FROM THE BOTTOM
50 REM LEFT-HAND CORNER TO THE
60 REM     TOP RIGHT.
```



```
 70 BORDER 2: PAPER 7: CLS : BR
IGHT 1
 80 LET S=0
 90 FOR Y=1 TO 704
 100 LET T=INT (RND*3)
 110 PRINT INK (RND*5+1); ("■" AN
D T=0)+(" " AND T<>0);
 120 NEXT Y
 130 PRINT AT 0,0;"    ";AT 1,0;"
"
 140 LET X=30: LET Y=19
 150 LET M=X
 160 LET N=Y
 170 PRINT AT Y,X; INK 2;"$"
 180 LET S=S+1
 190 LET A$=INKEY$
 200 IF A$="" THEN GO TO 190
 210 IF A$="A" AND Y>0 THEN LET
Y=Y-1
 220 IF A$="Z" AND Y<20 THEN LET
 Y=Y+1
 230 IF A$="K" AND X<31 THEN LET
 X=X+1
 240 IF A$="M" AND X>0 THEN LET
X=X-1
 250 IF X=0 AND Y=0 THEN GO TO 2
80
 260 PRINT AT N,M;" "
 270 GO TO 150
 280 PRINT AT 10,10; FLASH 1; BR
IGHT 1;"YOU MADE IT"
 290 PRINT  TAB 6; FLASH 1; BR
IGHT 1;"IT TOOK YOU ";S;" MOVES"
```

Note in line 110 how use is made of the computer's logic methods of evaluation. This line ensures that if T (generated in the previous line) is zero, a solid square is printed, and a blank space is generated if T is greater than zero. This is a memory-efficient way of emulating the IF/THEN/ELSE command available on some other computers. This technique, and similar ones, are discussed in greater detail elsewhere in the book.

If you want a time limit on user responses, use this method. Suppose the user had only a few seconds to decide whether or not to have another game. If he or she was too slow deciding then the program stopped. For the purpose of this routine suppose the user had to press R for a re-run:

```
10 FOR F=1 TO 100
20 LET A$=INKEY$
30 IF A$="A" THEN GO TO 60
40 NEXT F
50 STOP
60 PRINT "RE-RUN"
70 RUN
```

Alternatively, you can make use of the frame counter, for timing inputs or anything else.

To use the frame counter use this routine to first set the timer:

POKE 23673, 255
POKE 23672, 255

and to use its value at any time use:

LET T = (256 * PEEK 23673 + PEEK 23672)/50

This will give you a fairly accurate readout in seconds if you PRINT T. Remember that PAUSE and BEEP use the frame counter so it cannot be used for timing if you are using PAUSE or BEEP in your program. Try this, for a printout in seconds.

```
  10 POKE 23673,255
  20 POKE 23672,255
  30 LET t=(256*PEEK 23673+PEEK
23672)/50
  40 PRINT AT 0,0;t;"    "
  50 GO TO 30
```

And this for a digital clock:

```
  10 REM Digital Clock
  20 BORDER 1: PAPER 2: CLS
  30 INK 7
  40 INPUT "Enter hours ";h
  50 INPUT "Enter minutes ";m
  60 LET s=0
  70 POKE 23673,255
  80 POKE 23672,255
 110 PRINT AT 4,12;h;":";m;":";s
 120 IF m<10 THEN PRINT "0";
 130 PRINT m;":";"";
 140 IF s<10 THEN PRINT "0";
 150 PRINT s
 160 LET s=INT ((256*PEEK 23673+
PEEK 23672)/50)
```



```
 170 IF s>59.9999 THEN LET m=m+1
: POKE 23673,255: POKE 23672,255
 180 IF m=60 THEN LET h=h+1: LET
m=0
 190 IF h=13 THEN LET h=1
 200 GO TO 110
```

If you wish to time accurately, without resetting 23673 and 23672 back to zero over and over again, refer to the PEEK and POKE section, a little later in the book.

READ/DATA/RESTORE

READ and DATA are very convenient ways of accessing information within a program, and are relatively simple to use. Enter and run the following program, which shows READ and DATA in action, and then return to the book for an explanation of how it works.

```
  10 REM READ/DATA
  20 REM ***********
  30 REM READ THE DATA
  40 REM ***********
  50 DIM B(5)
  60 FOR A=1 TO 5
  70 READ B(A)
  80 NEXT A
  90 REM ***********
 100 REM READ IT BACK
 110 REM ***********
 120 FOR C=5 TO 1 STEP -1
 130 PRINT B(C)
 140 NEXT C
 150 DATA 13,35241,88,2,199999
```

```
RUN

199999
2
88
35241
13
```

In line **70**, the computer comes across the instruction READ... Whenever it finds a READ instruction, it goes to the first item following the word DATA, and READS that, in

this case, into an array. The DATA items can be anywhere in the program (although it is useful to keep them fairly close to the READ statement which refers to them).

Return to the program TABULATOR ROCKET RANGE which we used earlier in this book.

In the first program in this section, the DATA is numbers, and these are assigned to numerical variables (the elements of the array). In TABULATOR ROCKET RANGE, the DATA is strings (line 220) and these are assigned in turn to the string variables A$ and then printed out in line 120, through the loop labelled 'rocket'. There is a third word which goes with READ and DATA — RESTORE. This tells the computer to go back to the start of the list of DATA and start READing from the first item again. Here is another sample program, showing DATA in the form of strings, and illustrating RESTORE in action.

```
  10 REM READ/DATA
  20 REM ***********
  30 REM READ THE DATA
  40 REM ***********
  50 DIM B$(21,5)
  60 FOR A=1 TO 21
  70 READ B$(A)
  75 IF 3*INT (A/3)=A THEN RESTO
RE
  80 NEXT A
  90 REM ***********
 100 REM PRINT IT BACK
 110 REM ***********
 120 FOR C=21 TO 1 STEP -1
 130 PRINT B$(C);       ";
 140 NEXT C
 150 DATA "WET","KILL","DIE"

DIE     KILL    WET     DIE
KILL    WET     DIE     KILL
WET     DIE     KILL    WET
DIE     KILL    WET     DIE
KILL    WET     DIE     KILL
WET
```

In this program, there are only three items of DATA, so RESTORE must operate once the three have been read. Line 80 ensures that this occurs every time the three are read while running through the A loop from 1 to 21. Notice that string DATA must be enclosed within quote marks.

As you have seen, a READ command is used within a line to assign values to variables from a sequence of items contained within a DATA statement. Each item of DATA is separated from others by a comma. A READ statement is made up of a line number, followed by the word READ, and the variable names which are to be assigned to the variables taken from the DATA line.

When a program comes to a READ statement, it will — as we pointed out — move to the first DATA statement, no matter where it is in the program. The first value of the DATA statement will be assigned to the first variable in the READ statement. Apart from reading a DATA statement, the computer takes no notice of it, and will treat it as a REM statement. Move line 150 up to line 25, and run the preceding program again. You'll see that (a) the computer ignores line 25, and (b) still READs it successfully.

Even if the DATA is scattered all over the program, the computer will seek it out, as the following program shows.

```
  10 REM READ/DATA
  20 DATA "HI",2
  30 DIM B$(21,4)
  40 DIM Z(21)
  60 FOR A=1 TO 21
  70 READ B$(A),Z(A)
  80 IF 3*INT (A/3)=A THEN RESTO
RE
  85 DATA "GOSH"
  90 NEXT A
  95 DATA 56,"BOB"
 130 FOR C=1 TO 21
 140 PRINT B$(C),Z(C)
 145 DATA 22
 150 NEXT C
```

It is important to ensure that you have enough DATA items for the number of times you tell the computer to READ. Delete line 145 in the above program, and run it again. You will get the error message "E OUT OF DATA, 70:1" where the computer had read two items of numeric DATA, then was unable to find a third because RESTORE had not yet been evoked.

Remember that although it is not essential to have the DATA items near the READ lines which are looking for them, it will

probably make your programs easier to understand if they are held in this manner. It also makes it easier to know which lines to alter if you are working on a program.

User-defined graphics

One of the most exciting features on the ZX Spectrum is the facility for defining your own graphics. It is simple to do so, and allows you to tailor the visual output of your program to your own wishes. In this chapter, we'll develop a greatly simplified form of the arcade game 'Pacman', to show the user-defined graphics in action. Our game will be called DOTMAN, and will consist of a single 'Pacman' creature who is trying to escape a single 'Ghost', while at the same time trying to eat as many dots as possible.

Graphics are defined on an eight by eight grid, like the following:

We can select any key, from A to U, and impose our own graphic on that key, so whenever it is selected, instead of printing the letter, it will print our own graphic. Although — of course — you lose graphics when you turn the computer off, they are not effected by NEW, so you can NEW a program, and still have your graphics available for a subsequent game.

You change the contents of the graphics of a key by a short POKE loop. Here's an example. If you wanted a diamond shape to appear every time you pressed the CAPS SHIFT and GRAPHICS, so the cursor was a G, then pressed the A, you would proceed as follows.

First, on your grid (and there are several of them at the end of this chapter, which you could photocopy to get a number of them for future use), we draw the pattern we want to create. A diamond could look like this:

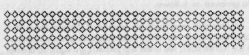
We POKE it into place using a series of eight DATA statements, in which a zero equals a blank little square, and a one equals a filled-in square. The top line of this diamond shape is then represented by **00001000**, an eight-digit binary number. The second line of the diamond is **00010100**. Compare this with the black squares on our diagram. We indicate to the computer that the number we are using is a binary one, by preceding the number in the DATA statement with BIN, which is available from the B key. Here is the 'diamond creating' routine in full:

```
 10 REM DIAMOND
 20 FOR J=0 TO 7
 30 READ Q
 40 POKE USR "A"+J,Q
 50 NEXT J
 60 PRINT "A"
 70 DATA BIN 00001000,BIN 00010
100,BIN 00100010,BIN 01000001,BI
N 00100010,BIN 00010100,BIN 0000
1000,BIN 00000000
```

Note that the A in line **60** is an A achieved after we have gone into the graphics mode. Run the program, and even the A in the listing changes, so that line **60** now looks like this:

```
60 PRINT "◇";
```

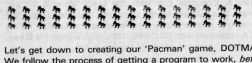
There is no doubt that we have created quite a passable diamond shape, as the printout shows:



You may have realised that the leading zeroes in the DATA items were redundant. However, we prefer to leave them in, just because it makes it much easier to check up on the elements within the DATA line if something goes wrong. Let's try for a different shape. Enter the following DATA statement, and then run it to see what you get:

```
 70 DATA BIN 00110000,BIN 01110
000,BIN 01111110,BIN 01011111,BI
N 10011111,BIN 00010010,BIN 0001
0010,BIN 00000000
```

This was intended to be an elephant, and it worked pretty well as you can see on your screen.



Let's get down to creating our 'Pacman' game, DOTMAN. We follow the process of getting a program to work, *before* defining the graphics characters which are part of the program. This ensures that the program itself becomes the important thing, not the graphics which will be used in it. It is perfectly possible to see a whole program working, just using letters, before deciding exactly what each letter will represent.

Our game will be fairly simple. On a 14 x 14 grid, there will be one DOTMAN, who will be under our control, a GHOST under the control of the computer, and a series of dots which can be eaten by the DOTMAN. A few walls will be on the grid

(a maze) and neither the GHOST nor the DOTMAN can get through those.

At the end of the book, in the section called 'Improving your programs', it is suggested that you map out the major parts of the program before you begin, so the structure of the program is clear, even before you begin working on the program. If you follow through the process we'll describe, you'll see how useful this can be.

Dotman

For DOTMAN, the program structure will be as follows:

1- 999: Send action to parts of the program to start
1000-1999: Player controls DOTMAN
2000-3999: GHOST moves
4000-4999: End of game if GHOST lands on DOTMAN
5000-5999: Define DOTMAN
6000-6999: Define GHOST
7000-7999: Print maze
8000-8999: Initialise variables

Once this program is underway, it will loop from 1000 through to 3999 (or whichever is the highest 'GHOST moves' number) over and over again, until the GHOST gets the DOTMAN, at which time the program will proceed to line 4000 for the end of game routine.

Let's write the first control routines:

```
10 GO SUB 8000
20 GO SUB 7000
30 GO SUB 5000
40 GO SUB 6000
```

This is all that is needed at this point. And, as we're going to write the whole game before defining the DOTMAN and the GHOST, we can add 6999 RETURN and 5999 RETURN right

now, and get on with building the maze, after we have defined the variables. GA is the position of the GHOST across, GD the 'down' coordinate of the GHOST. EGA and EGD are the variables to 'erase' the GHOST when it moves. Similarly, DD and DA are for DOTMAN across and down, and EDD and EDA for erasing.

We put in the maze, and the routines to move the DOTMAN and the GHOST, and our program looks like this:

```
  10 GO SUB 6000
  20 GO SUB 7000
  30 GO SUB 5000
  40 GO SUB 6000
1000 REM move dotman
1010 REM dotman is graphic B,C,D
 ,E
1020 PRINT AT EDD,EDA;" "
1030 PRINT AT DD,DA;A$
1035 LET EDD=DD: LET EDA=DA
1040 IF INKEY$="9" THEN IF DA<13
 THEN IF SCREEN$ (DD,DA+1)<>"X"
THEN LET DA=DA+1: LET A$="B"
1050 IF INKEY$="8" THEN IF DA>0
THEN IF SCREEN$ (DD,DA-1)<>"X" T
HEN LET DA=DA-1: LET A$="C"
1060 IF INKEY$="7" THEN IF DD>0
THEN IF SCREEN$ (DD-1,DA)<>"X" T
HEN LET DD=DD-1: LET A$="E"
1070 IF INKEY$="6" THEN IF DD<13
 THEN IF SCREEN$ (DD+1,DA)<>"X"
THEN LET DD=DD+1: LET A$="D"
1980 IF RND>.2 THEN GO TO 1000
1990 REM GHOST IS GRAPHIC G
2000 PRINT AT EGD,EGA;" "
2010 PRINT AT GD,GA;"G"
2015 LET EGD=GD: LET EGA=GA
2020 IF DD<GD THEN IF SCREEN$ (G
D-1,GA)<>"X" THEN LET GD=GD-1
2030 IF DD>GD THEN IF SCREEN$ (G
D+1,GA)<>"X" THEN LET GD=GD+1
2040 IF DA>GA THEN IF SCREEN$ (G
D,GA+1)<>"X" THEN LET GA=GA+1
2050 IF DA<GA THEN IF SCREEN$ (G
D,GA-1)<>"X" THEN LET GA=GA-1
2999 GO TO 1000
4999 STOP
5999 RETURN
6999 RETURN
7000 REM BUILD MAZE
7010 FOR G=1 TO 14
7020 FOR H=1 TO 14
7030 PRINT "."
7040 NEXT H
```



```
7050 PRINT
7060 NEXT G
7070 PRINT AT 2,7;"XXXXX"
7080 PRINT AT 3,3;"X";AT 3,11;"X
"
7090 PRINT AT 4,3;"X";AT 4,11;"X
"
7100 PRINT AT 5,3;"XXXX";AT 5,9;
"XX"
7110 PRINT AT 6,6;"X";AT 6,9;"X"
7120 PRINT AT 7,5;"X";AT 7,9;"X"
7130 PRINT AT 8,5;"X";AT 8,8;"X"
;AT 8,9;"X"
7140 PRINT AT 10,3;"X"
7150 PRINT AT 11,3;"XXXX"
7160 PRINT AT 12,9;"XXX"
7170 RETURN
8000 REM VARIABLES
8010 LET DA=0
8020 LET DD=0
8030 LET EDA=0
8040 LET EDD=0
8050 LET GA=13
8060 LET GD=13
8070 LET EGA=13
8080 LET EGD=13
8090 LET SCORE=0
8100 LET A$="B"
8999 RETURN
```

If you run this, you'll see we have a version of the program which works, after a fashion. It is pretty dull, at the moment, but do not be disheartened. You'll be pleasantly surprised at how good it looks when we finish. It can be improved very simply, even at this point by adding:

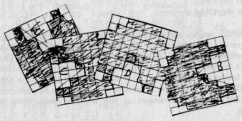
7065 INVERSE 1
7165 INVERSE 0

INVERSE works somewhat like BRIGHT and FLASH, one means on, zero means off. Run the program again, and see the effect INVERSE has had on the walls of the maze. This lifts it immediately, doesn't it. There must be four DOTMEN, one facing each direction, so that the mouth points the way the DOTMAN is moving. This is why the variable A$, which is the DOTMAN, is changed at the ends of lines 1040 to 1070.

Add the following, to define the DOTMEN:

```
5000 REM DEFINE DOTMEN
5010 FOR J=0 TO 7
5020 READ Q
5030 POKE USR "B"+J,Q
5040 NEXT J
5050 DATA BIN 00111100,BIN 01111
111,BIN 11111100,BIN 11110000,BI
N 11111000,BIN 11111100,BIN 0111
1111,BIN 00111100
5060 FOR J=0 TO 7
5070 READ Q
5080 POKE USR "C"+J,Q
5090 NEXT J
5100 DATA BIN 00111100,BIN 11111
110,BIN 00111111,BIN 00011111,BI
N 00001111,BIN 00011111,BIN 1111
1110,BIN 00111100
5110 FOR J=0 TO 7
5120 READ Q
5130 POKE USR "D"+J,Q
5140 NEXT J
5150 DATA BIN 00111100,BIN 01111
110,BIN 11111111,BIN 11111111,BI
N 11110111,BIN 11100111,BIN 0100
0010,BIN 01000010
5160 FOR J=0 TO 7
5170 READ Q
5180 POKE USR "E"+J,Q
5190 NEXT J
5200 DATA BIN 01000010,BIN 01000
010,BIN 11100111,BIN 11101111,BI
N 11111111,BIN 11111111,BIN 0111
1110,BIN 00111100
```

We sketched a number of possible DOTMAN figures, before deciding on the one labelled B. This diagram was rotated to get the BIN numbers for the four figures.



Add some colour, by inserting INK 2 in line **1030**. This will print red DOTMEN. INK 1, in line **2010** will turn the GHOST, when he's complete, blue.

Here are our working sketches for the GHOST.

The GHOST we finally ended up with is a close cousin of the one on the left in the diagram. Although it does not look particularly promising here, it worked quite well on the screen. You'll probably find, from time to time, that you will want to modify the final graphic when you see it on the screen, and you'll find it fairly easy to work directly from the screen, soon learning which bits of the DATA statement to change. Anyway, here's our final version of DOTMAN, complete with a few screen printouts. As is often the case, the printouts really don't do justice to the appearance of the game. You're sure to enjoy playing it, and working out ways to improve it, and make it more like the arcade game.

One change you might like to make is to add a routine to generate a random maze (by putting X's at random on the grid in the **7000** subroutine). As well as this, you can let the DOTMAN and the GHOST start at random positions on the maze. A 'highest score' feature could also add interest.

SCORE IS 0

SCORE IS 9428

SCORE IS 21213

SCORE IS 47140

SCORE IS 105422

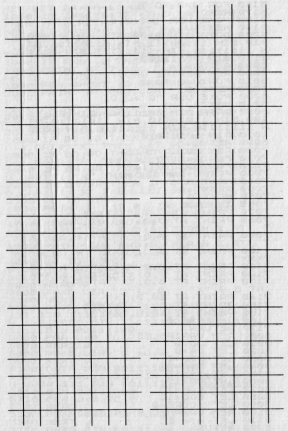
END OF GAME

```
  10 GO SUB 8000
  20 GO SUB 7000
  30 GO SUB 5000
  40 GO SUB 6000
1000 REM move dotman
1010 REM dotman is graphic B,C,D
or
1020 PRINT AT EDD,EDA;" "
1025 IF SCREEN$ (DD,DA)="." THEN
 LET SCORE=SCORE+2357
1030 PRINT AT DD,DA; INK 2;A$
1032 IF GA=DA AND GD=DD THEN GO
TO 4000
1035 LET EDD=DD: LET EDA=DA
1040 IF INKEY$="8" THEN IF DA<13
THEN IF SCREEN$ (DD,DA+1)<>"X"
THEN LET DA=DA+1: LET A$="C"
1050 IF INKEY$="5" THEN IF DA>0
THEN IF SCREEN$ (DD,DA-1)<>"X" T
```

```
HEN LET DA=DA-1: LET A$="➔"
1060 IF INKEY$="7" THEN IF DD>0
THEN IF SCREEN$ (DD-1,DA)<>"X" T
HEN LET DD=DD-1: LET A$="➘"
1070 IF INKEY$="6" THEN IF DD<13
 THEN IF SCREEN$ (DD+1,DA)<>"X"
THEN LET DD=DD+1: LET A$="➙"
1080 IF RND>.2 THEN GO TO 1000
1085 PRINT AT 7,16; FLASH 1;"SCO
RE IS ";SCORE
1090 REM GHOST IS GRAPHIC G
2000 PRINT AT EGD,EGA; INK RND*6
;"▚"
2010 PRINT AT GD,GA; INK 1;"▚"
2012 IF GA=DA AND GD=DD THEN GO
TO 4000
2015 LET EGD=GD: LET EGA=GA
2020 IF DD>GD THEN IF SCREEN$ (G
D+1,GA)<>"X" THEN LET GD=GD+1
2030 IF DD>GD THEN IF SCREEN$ (G
D+1,GA)="X" THEN LET GD=GD+1
2040 IF DA>GA THEN IF SCREEN$ (G
D,GA+1)<>"X" THEN LET GA=GA+1
2050 IF DA<GA THEN IF SCREEN$ (G
D,GA-1)<>"X" THEN LET GA=GA-1
2999 GO TO 1000
4000 PRINT AT 17,0; INK RND*6; P
APER 9; FLASH 1;"END OF GAME"
4010 BEEP .01,RND*60
4020 PRINT INK RND*6; FLASH 1;AT
 EGD,EGA;" ";AT GD,GA;"▚"
4030 PRINT INK RND*6; FLASH 1;AT
 EDD,EDA;" ";AT DD,DA;"▚"
4040 GO TO 4000
5000 REM DEFINE DOTMEN
5010 FOR J=0 TO 7
5020 READ D
5030 POKE USR "B"+J,D
5040 NEXT J
5050 DATA BIN 00111100,BIN 01111
111,BIN 11111100,BIN 11100000,BI
N 11111000,BIN 11111100,BIN 0111
1111,BIN 00111100
5060 FOR J=0 TO 7
5070 READ D
5080 POKE USR "C"+J,D
5090 NEXT J
5100 DATA BIN 00111100,BIN 11111
110,BIN 00111111,BIN 00011111,BI
N 00001111,BIN 00111111,BIN 1111
1110,BIN 00111100
5110 FOR J=0 TO 7
5120 READ D
5130 POKE USR "D"+J,D
5140 NEXT J
5150 DATA BIN 00111100,BIN 01111
110,BIN 11111111,BIN 11111111,BI
N 11110111,BIN 11100111,BIN 0100
0010,BIN 01000010
```



```
5160 FOR J=0 TO 7
5170 READ D
5180 POKE USR "E"+J,D
5190 NEXT J
5200 DATA BIN 01000010,BIN 01000
010,BIN 11100111,BIN 11101111,BI
N 11111111,BIN 11111111,BIN 0111
1110,BIN 00111100
5999 RETURN
6000 REM DEFINE GHOST
6010 FOR J=0 TO 7
6020 READ D
6030 POKE USR "G"+J,D
6040 NEXT J
6050 DATA BIN 00111000,BIN 01111
100,BIN 11010110,BIN 11010110,BI
N 11111110,BIN 11111110,BIN 1010
1010,BIN 10101010
6999 RETURN
7000 REM BUILD MAZE
7010 FOR G=1 TO 14
7020 FOR H=1 TO 14
7030 PRINT INK RND*6;".";
7040 NEXT H
7050 PRINT
7060 NEXT G
7065 INVERSE 1
7070 PRINT AT 2,7;"XXXXX"
7080 PRINT AT 3,9;"X";AT 3,11;"X
"
7090 PRINT AT 4,3;"X";AT 4,11;"X
"
7100 PRINT AT 5,3;"XXXX";AT 5,9;
"XXX"
7110 PRINT AT 6,6;"X";AT 6,9;"X"
7120 PRINT AT 7,6;"X";AT 7,9;"X"
7130 PRINT AT 8,3;"X";AT 8,6;"X"
;AT 8,9;"X"
7140 PRINT AT 10,3;"X"
7150 PRINT AT 11,3;"XXXX"
7160 PRINT AT 12,9;"XXX"
7165 INVERSE 0
7170 RETURN
8000 REM VARIABLES
8010 LET DD=0
8020 LET DA=0
8030 LET EDD=0
8040 LET EDA=0
8050 LET GA=13
8060 LET GD=13
8070 LET EGA=13
8080 LET EGD=13
8090 LET SCORE=0
8100 REM GRAPHIC B IN 8110
8110 LET A$="➔"
8999 RETURN
```



Clearing a part of the display

This is a useful subroutine that enables you to clear any number of lines from the bottom of the screen. It enables the instructions to be kept on the top few lines of the display during the game, or the score or other special instructions may be kept on-screen while everything else is cleared. The subroutine should be called by GOSUB **8010**.

```
8010 INPUT "How many lines to be
 cleared? ";c
8020 IF c<0 OR c>21 THEN GO TO 8
010
8030 FOR f=21 TO 21-c STEP -1
8040 PRINT AT f,0;"
                  "
8050 NEXT f
8060 PRINT AT f+1,0;
8070 RETURN
```

Line **8010** asks how many lines you want to clear, starting from line 21 at the bottom of the screen and working upwards — it looks quite impressive. The INPUT statement is not idiot-proofed; you may like to experiment with this yourself. The statement in line **8060** moves the PRINT position to the start of the part of the screen you've just cleared.

Screen scrolling in BASIC

Machine code programs are easily written to block-load large chunks of memory from one location to another to enable, for instance, the screen to be block-loaded in certain directions. BASIC as a rule is usually too slow to enable this to be done. The only method that can be realistically used to scroll the

screen is to store an image of the screen in a string array and PRINT this. Here are four example programs that enable the entire screen to be scrolled up, down, left or right.

These routines are noticeably slower than the command SCROLL, especially the left and right scrolls because those are PRINTed one line at a time.

Upward scroll

```
  5 REM Upward scroll
 10 DIM a$(704)
 20 INPUT a$
 30 PRINT AT 0,0;a$
 40 LET a$=a$(33 TO )+"
50 GO TO 30
```

Downward scroll

```
  5 REM Downward scroll
 10 DIM a$(704)
 20 INPUT a$
 30 PRINT AT 0,0;a$
 40 LET a$="
50 GO TO 30
```

Leftward scroll

```
  5 REM Leftward scroll
 10 DIM a$(704)
 20 INPUT a$
 30 PRINT AT 0,0;a$
 40 FOR f=1 TO 673 STEP 32
 50 LET a$(f TO f+31)=a$(f+1 TO
f+31)+"
 60 NEXT f
 70 GO TO 30
```


Rightward scroll

```
  5 REM Rightward scroll
 10 DIM a$(704)
 20 INPUT a$
 30 PRINT AT 0,0;a$
 40 FOR f=1 TO 673 STEP 32
 50 LET a$(f TO f+31)="
TO f+30)
 60 NEXT f
 70 GO TO 30
```

Saving lines at screen edges

The easiest way to accomplish this is to use the string slicing facility to "SCROLL" certain parts of the string only. First of all, the upward scroll. L is the number of lines NOT to be scrolled.

```
 10 DIM a$(704)
 20 INPUT a$
 25 INPUT L
 30 PRINT AT 0,0;a$
 40 LET a$(L*32+1 TO )=a$((L+1)
*32+1 TO
 50 GO TO 30
```

Next the downward scroll. L is the number of lines not to be scrolled at the bottom of the screen.

```
 10 DIM a$(704)
 20 INPUT a$
 25 INPUT L
 30 PRINT AT 0,0;a$
 40 LET a$( TO 704-L*32)="
TO 704-(L+1)*32)
 50 GO TO 30
```

The same technique can be applied to the sideways scrolls, but since these are slow enough already it hardly seems

worthwhile. You can extend this idea to permit lines in any part of the screen to be kept stationary while those above and/or below are scrolled, simply by modifying the string slicing, but if you require complex arithmetic to work this out then you may slow the routine down excessively.

Another technique you can apply to these routines is "wraparound" whereby anything disappearing off any edge promptly reappears on the opposite edge. Here's how to do this with an upward scroll:-

```
  10 DIM a$(704)
  20 INPUT a$
  30 PRINT AT 0,0;a$
  40 LET a$=a$(33 TO )+a$( TO 32
)
  50 GO TO 30
```

And a downward scroll with wraparound:-

```
  10 DIM a$(704)
  20 INPUT a$
  30 PRINT AT 0,0;a$
  40 LET a$=a$(673 TO )+a$( TO 6
72)
  50 GO TO 30
```

A leftward scroll with wraparound:-

```
10 DIM a$(704)
20 INPUT a$
30 PRINT AT 0,0;a$
40 FOR f=1 TO 673 STEP 32
50 LET a$(f TO f+31)=a$(f+1 TO
f+31)+a$(f)
60 NEXT f
70 GO TO 30
```

For a rightward scroll with wraparound, change line 50 to:-

```
  50 LET a$(f TO f+31)=a$(f+31)+
a$(f TO f+30)
```

Finally, remember: instead of using PRINT to place anything on screen, use LET A$(X) = "X", since anything you PRINT is not scrolled — only the contents of the array A$ appear on screen.

Moving graphics

The theory behind moving graphics is that first we draw a character in one position for a short time, then erase it and draw it in another position. Variables are used to remember the position of the character. Let us look at an example program which we can draw up from this "theory":

```
10 LET x=0
20 PRINT AT 5,x;"■"
30 PRINT AT 5,x;""
40 LET x=x+1
50 GO TO 20
```

This is not a very good program — the black blob seems to flash on and off as it moves across the screen and the program stops with an error report when the blob reaches the right-hand side of the screen. What has happened is that X was the variable that told the computer how far across the screen it should PRINT the black blob, and if the number is greater than 31 then your computer cannot PRINT since the limits of the screen are from 0 to 31, and any attempts to use a number greater than 31 would place the blob off the screen to the right, either in your television speaker or your living room or wherever. Now the Spectrum, being a rather clever little machine, decides that this is not on, so it stops the program and tells you what went wrong, so that you can correct it. Let us do this. The easiest way is to arrange that if the value of X goes outside the permitted range then the program automatically changes it to a suitable value. One way in which this can be done is to add a line like:

```
45 IF x>31 THEN LET x=0
```

But since we already have a line saying LET X = 0 in line 10 then we could do the same thing by sending the program back to line 10, like this:

```
45 IF x>31 THEN GO TO 10
```

This technique is used rather a lot in programs — sending a program back to the beginning in order to reset some variables to their starting values. In this example both methods achieve the same results, but you may come across certain programs where only one is suitable, or one method is better than the other.

The next step is to improve the flashing display. One way to do this is to make sure the blob is displayed for longer than it is erased. Try this:

```
10 LET x=0
20 PRINT AT 5,x;"■"
30 FOR f=1 TO 10
40 NEXT f
50 PRINT AT 5,x;" "
60 LET x=x+1
70 IF x>31 THEN LET x=0
80 GO TO 20
```

Seems to work rather well, but in most programs there are other computations to be carried out which will slow down the program, and the time-wasting loop of lines **30** & **40** will slow things down unnecessarily, so this is not a very good approach. Let us try to reduce the flashing by reducing the time between the space being PRINTed and the blob being PRINTed. Try this program:

```
10 LET x=0
20 LET p=x
30 LET x=x+1
40 IF x>31 THEN LET x=0
50 PRINT AT 5,p;" ";AT 5,x;"■"
60 GO TO 20
```

See how much smoother this is.

Here, X is the variable that remembers the current position of the blob. P remembers the previous position so that it may be erased by drawing a space over it. Line **10** makes X start off with a value of **0**. Line **20** determines the value of P by making it equal to X *before* X is increased in value in line **30**. This can be any amount of increase — try changing the number after the + sign to see the effect. It appears to move

faster, and that can be advantageous or disadvantageous. Stay with 1 after the + sign for now.

The action of line **40** is to ensure that when the blob has reached the right-hand side of the screen it is sent back to the left-hand side of the screen by resetting X to **0**. This ensures a constant supply of blobs for us! Line **50** does all the PRINTing — note how two AT functions can be placed on the same line joined by a semi-colon. We can also do the same with TAB incidentally. Try writing these as two separate lines like this to see if it makes any difference to the program:

```
50 PRINT AT 5,p;" "
55 PRINT AT 5,x;"■"
```

We can shorten the program by one line by changing line **30** to:

```
30 LET x=x+1 AND x<31
```

The way in which this line works is rather complex, and is more fully explained in the sections on conditional statements. Simplified, it means "if X is less than 31 then add 1 to the value of X, but if X is not less than 31 then make X zero". Now delete line **40** (the one we've made redundant) and renumber nicely in steps of **10** and we end up with this:

```
10 LET x=0
20 LET p=x
30 LET x=x+1 AND x<31
40 PRINT AT 5,p;" ";AT 5,x;"■"
50 GO TO 20
```

You may have noticed that the space is PRINTed one column behind the blob all the time, and therefore it may be possible to simply use PRINT AT 5,X;"(space)□" and dispense with the variable P altogether. This does make the display smoother, but it causes problems when it gets to the end of the line and you need an extra line to clear this position. To see what we mean, try this program:

```
10 LET x=0
20 LET x=x+1 AND x<30
30 PRINT AT 5,x;" ■"
40 GO TO 20
```

See what we mean? Every time a new blob shoots across the screen the old one stays on the right of the screen. Let us add a facility to erase these blobs:

```
10 LET x=0
20 LET x=x+1 AND x<30
30 PRINT AT 5,x;" ■"
35 IF x=0 THEN PRINT AT 5,31;"
40 GO TO 20
```

Again we can use AND to shorten the program a little:-

```
10 LET x=0
20 LET x=x+1 AND x<30
30 PRINT AT 5,x;" ■";AT 5,31;"
" AND x=0
40 GO TO 20
```

Line 35 in the first program and the second part of line 30 in the second ensure that a space is only PRINTed if the blob has reached the end of its travel. Here is another way of doing this, using the control variable of a FOR/NEXT loop instead of the conventional variable X. This method uses slightly less memory and is slightly faster to run.

```
 10 FOR x=0 TO 30
 20 PRINT  AT 5,x;" ■";AT 5,31;"
" AND x=0
 30 NEXT x
 40 GO TO 10
```

Another way to use this method is to have PRINT AT 5,31;"(space)[ ]" outside the FOR/NEXT loop. This has the advantage that the computer does not have to examine the conditional expression so often, so the program runs considerably faster:


```
10 FOR x=0 TO 30
20 PRINT AT 5,x;" ■"
30 NEXT x
40 PRINT AT 5,31;" "
50 GO TO 10
```

So we've ended up with a routine that is quite fast running and economical in memory usage.We've also seen some of the problems of developing this kind of program. The examples we've seen so far deal with constant movement across the screen. We will also require to move characters about the screen, possibly with control of this movement from the keyboard, so that the operator may control the movement. To do this, we first need to return to INKEY$, a very useful aid for moving graphics. If you've read the earlier section, you'll recall that INKEY$ is the character which corresponds to the key being pressed on the keyboard. If you press K, then INKEY$ is "K' or INKEY$ = "k". If you aren't pressing anything on the keyboard, INKEY$ becomes the empty string, or "". (INKEY$ cannot be "(space)" because pressing SPACE acts as BREAK when a program is running and stops a BASIC program.) We will look now at some other ways in which we can use INKEY$, because it is a powerful and extremely useful function for moving graphics. Most arcade games require you to press buttons, flick switches or manipulate joysticks. Your computer does not have these (unless you buy or build an add-on board), so all control has to be done through the keyboard.

To control movement on the screen, certain keys have advantages over others because of their layout. For instance, take the Z and the M keys on the bottom row of the keyboard. They can be used to move left (Z) or right (M), to control left/right movement on screen, e.g. in an invaders type of game. Their advantage is that they are logically placed, making them convenient to use. One disadvantage is that the M is near to the BREAK key so that you may accidentally stop the program in a frenzied hurry to avoid being bombed by the Green Menace.

If you want to control left/right/up/down movement (e.g. to move the cursor in the word-processor program near the back of this book) then the 5,6,7,8 keys are a better choice

since they have arrows pointing in the four directions printed on the keys. They are next to each other for convenience of use and they are not dangerously near BREAK.

The most common way of using INKEY$ in a moving graphics program (be it games or serious applications) is to put it in a conditional statement to control the value of a variable. For example:

IF INKEY$ = "8" THEN LET X = X + 1

Here, one is added to the value of X if the 8 key is being pressed, but X stays the same if no key is being pressed, or a key other than 8 is being pressed. Having done this, the variable can be used to control, for example, where to PRINT on the screen, for example:

PRINT AT 5,X;"□"

If we adopt the convention of X being the horizontal position across the screen and Y being the vertical position on the screen, then the bigger the value of X, the further to the right across the screen the character is PRINTed and the bigger the value of Y the further down the screen the character is PRINTed.

So knowing this we can write a short program to control the movement of a character (e.g. an asterisk) on the screen using the keys 5,6,7 & 8 (the keys with arrows printing on them) like this:

```
100 IF INKEY$="5" THEN LET x=x-
1
110 IF INKEY$="8" THEN LET x=x+
1
120 IF INKEY$="7" THEN LET y=y-
1
130 IF INKEY$="6" THEN LET y=y+
1
140 PRINT AT y,x;"*"
```

Before we can RUN this program we need to define X and Y or nothing will happen. Add these lines to the program:


```
10 LET x=0
20 LET y=0
```

We can now RUN the program. What we get is an asterisk PRINTed at the top left corner of the screen, and the program stops after line 140. To prevent this happening we can add a line like:

```
200 GO TO 100
```

This ensures that the computer carries on doing the task over and over again. RUN the program and try pressing the 5,6,7,8 keys one at a time. You will see that as the asterisk moves it leaves behind a trail of asterisks. Keep going to the edge of the screen and keep pressing the keys. Strange things begin to happen. If you go off the bottom of the screen or off the right-hand side of the screen the program stops with an error message. This is because the value of X has gone greater than 31 or the value of Y has gone greater than 21, and as a consequence the computer is being asked to PRINT outside the screen boundaries, and this of course it cannot do.

However, if you try to move past the top of the screen or the left-hand edge of the screen then even stranger things begin to happen — the asterisk starts to travel in the opposite direction! It's not that we've broken the computer or anything like that, there is a simple explanation. When the value of X or Y is negative (as happens when you try to go off the top or the left-hand side of the screen) then PRINT ignores the — sign (it takes the ABS value if you like), and so it apparently causes some keys to change functions! Not very useful to say the least, since you then have to take the asterisk back to the screen boundaries to restore normal operation. What we need is a method whereby if the asterisk gets to the edge of the screen it stops and will not attempt to go outside the screen line.

The easiest method is to prevent X and Y taking values which causes these problems. The values which are permissible are X: 0 to 31 and Y: 0 to 21. Here's how to do this. Change lines 100, 110, 120 and 130 like this:

```
  100 IF INKEY$="5" AND x>0 THEN
LET x=x-1
  110 IF INKEY$="6" AND x<31 THEN
  LET x=x+1
  120 IF INKEY$="7" AND y>0 THEN
LET y=y-1
  130 IF INKEY$="6" AND y<21 THEN
  LET y=y+1
```

This makes the asterisk stay on the screen properly, but we still have the problem that a trail of asterisks is set up as the asterisk moves. This is because we have no facility to erase the old position of the asterisk when it moves. The best way to do this is to use a second set of variables to remember the old position of the character, and if it's different to the new position PRINT a space in the old position.

To do this add these lines:

```
   50 LET a=x
   60 LET b=y
  135 IF a<>x OR b<>y THEN PRINT
AT b,a;" "
  200 GO TO 50
```

The program you have in the computer should now be:

```
   10 LET x=0
   20 LET y=0
   50 LET a=x
   60 LET b=y
  100 IF INKEY$="5" AND x>0 THEN
LET x=x-1
  110 IF INKEY$="6" AND x<31 THEN
  LET x=x+1
  120 IF INKEY$="7" AND y>0 THEN
LET y=y-1
  130 IF INKEY$="6" AND y<21 THEN
  LET y=y+1
  135 IF a<>x OR b<>y THEN PRINT
AT b,a;" "
  140 PRINT AT y,x;"*"
  200 GO TO 50
```

We now have the basic design of a moving graphics program. When we get around to designing a game around this routine we may have to alter some details or change the order of statements but the principles involved will be similar. As we

did earlier we may also be able to shorten the routine somewhat, for example:

```
   10 LET x=0
   20 LET y=0
   30 LET a=x
   40 LET b=y
  100 LET x=x-(INKEY$="5" AND x>0
)+(INKEY$="6" AND x<31)
  110 LET y=y-(INKEY$="7" AND y>0
)+(INKEY$="6" AND y<21)
  135 IF a<>x OR b<>y THEN PRINT
AT b,a;" "
  140 PRINT AT y,x;"*"
  200 GO TO 30
```

This version occupies nearly fifty bytes of memory less than the previous version. You could also combine line 140 with line 135 into one conditional statement, since neither PRINT statement is required unless the asterisk has moved. This means we can delete line 140. Here is how to change line 135:

```
  135 IF a<>x OR b<>y THEN PRINT
AT b,a;" ";AT y,x;"*"
```

Remember to delete line 140. The disadvantage is that while saving 5 more bytes of memory, when the program is first RUN the asterisk does not actually appear on screen until it is moved. So it may be better not to do this unless you are desperate for memory.

SCREEN$ and scrolling

Let us now look at another facility which is useful in moving graphics programs, scrolling. Scrolling, which can be done automatically by including the line POKE 23692,−1, moves everything in the display up one line and if there was anything printed on the top line of the display then it is lost.

The next program — ROAD RUNNER — shows scrolling in action again to produce moving graphics. In this program you are attempting to drive a long line of little 'cars' down a twisting, turning track of red asterisks. Your controls are "Z" and "M" which move you left and right respectively.

Lines **80** and **90** moves the track randomly, making sure that it does not stray off the edge of the screen. Line **110** prints the 'car', which is scrolled up (as is the track), by lines **130** and **140**.

Line **160** introduces a new Spectrum function SCREEN$ which returns information as to the state of the screen at the location which is specified following the word SCREEN$. In line **160**, the computer uses SCREEN$ to look at the character cell just in front of where the last 'car' was printed. If it finds an asterisk there, it knows the car is about to crash, so sends action to line **200**, where the YOU HAVE CRASHED routine begins.

```
  10 REM ROAD RUNNER
  20 LET C=0
  30 LET T=0
  40 GO SUB 250
  50 LET A=10
  60 LET X=15
  70 LET Y=12
  80 LET K=INT (RND*2)
  90 LET A=A-(K=1 AND A>1)+(K=0
AND A<24)
 100 REM NEXT LINE CONTAINS A
       GRAPHIC C AS DOES 200
 110 PRINT AT Y,X-1; INK 1;"  "
 120 PRINT AT 20,A; INK 2;"*";TA
B A+5;"*"
 130 PRINT
 140 POKE 23692,-1: PRINT
```



```
 150 PRINT INK 6; PAPER 2;AT 0,1
0;" SCORE IS ";T;"  "
 160 IF SCREEN$ (Y+1,X-1)="*" TH
EN GO TO 200
 170 LET X=X-(INKEY$="Z")+(INKEY
$="M")
 180 LET T=T+1
 190 GO TO 80
 200 PRINT AT Y,X-1; INK 1;"  "
 210 PRINT AT 6,6; FLASH 1; BRIG
HT 1;"  You have crashed!!  "
 220 PRINT AT 8,10; FLASH 1; BRI
GHT 1; INK RND*7; PAPER 9;"  YOU
 SCORED ";T;"  "
 230 BEEP .01,RND*20-RND*20
 240 GO TO 210
 250 FOR J=0 TO 7
 260 READ Z
 270 POKE USR "C"+J,Z
 280 NEXT J
 290 RETURN
 300 DATA BIN 00110110,BIN 00110
110,BIN 00111110,BIN 00010100,B
IN 00111110,BIN 00110110,BIN 00
011100,0
```

Here are a few other programs to illustrate 'moving graphics' ideas we've discussed.

Red arrow

The object of this game is to fire, by pressing any key except BREAK, when the RED ARROW flies over your base, and be rewarded with a hit if you succeed. You have only a limited number of shots, so try to score as many hits as possible before the game stops.

Here is the listing:

```
  5 REM Line 10 contains
  7 REM graphics A B and C
  8 GO SUB 200
  9 PAPER 7: CLS
 10 PRINT AT 15,14; INK 1;"  "
;AT 4,0; PAPER 7; INK 2;"  "
;AT 16,0; INK 3;"  "
```

```
  20 LET h=0
  30 LET m=h
  40 LET y=RND*6+7
  50 FOR x=0 TO 19
  60 IF h+m=250 THEN GO TO 1000
  70 REM Graphic D in 0a
  80 PRINT AT y,x; INK 2;"»"
  90 IF INKEY$<>"" AND x=14 THEN
GO TO 140
 100 IF INKEY$<>"" THEN LET m=m+
1
 110 NEXT x
 120 PRINT AT y,20;" "
 130 GO TO 40
 140 LET h=h+1
 150 FOR r=1 TO 30
 160 PRINT AT y,x; INK 2; PAPER
4; BRIGHT 1; FLASH 1;"Hit": FOR
z=1 TO 20: BORDER RND*7: BEEP .0
1,99-3*z: PRINT AT y,x;"   "
 170 NEXT r
 180 PRINT AT 4,11; PAPER 7; INK
2;
 190 GO TO 40
 200 FOR j=0 TO 7
 210 READ q
 220 POKE USR "A"+j,q
 230 NEXT j
 240 FOR j=0 TO 7
 250 READ q
 260 POKE USR "B"+j,q
 270 NEXT j
 280 FOR j=0 TO 7
 290 READ q
 300 POKE USR "C"+j,q
 310 NEXT j
 320 FOR j=0 TO 7
 330 READ q
 340 POKE USR "D"+j,q
 350 NEXT j
 400 RETURN
 500 DATA BIN 00000001,BIN 00000
011,BIN 00000110,BIN 00001100,BI
N 00011000,BIN 00110000,BIN 0110
0000,BIN 11000000
 510 DATA BIN 11111111,BIN 11111
111,0,0,0,0,0,0
 520 DATA BIN 11100000,BIN 11100
000,BIN 00110000,BIN 00011000,BI
N 00001100,BIN 00000110,BIN 0000
0011,1
 530 DATA BIN 00001000,BIN 10000
100,BIN 01000010,BIN 00111111,BI
N 01000010,BIN 10000100,BIN 0000
1000,0
1000 FOR 9=1 TO 70
1010 BEEP .01,9/2: BEEP .01,40-9
1020 NEXT 9
1030 RUN
```


Line 10 sets up the main part of the display, the parts that do not move during the program. H is the variable that remembers the amount of hits you have scored and M the amount of misses (shots without hitting the U.F.O.) and both are initially set to zero. Line 40 sets the Y coordinate of the ARROW to a value from 7 to 13 at random. PRINT takes the value of the nearest whole number if one of the coordinates is not an integer, so RND * 6 + 7 can be from 7 to 13. This gives the position of the ARROW up the screen. The main loop for controlling the flight of the ARROW across the screen begins at line 50. 0 is the starting position, 19 the end position. Line 60 checks if you've used up your shots, then sends the program to line 1000 if you have. Line 80 PRINTs the ARROW. Note how it erases the old position by PRINTing a space one position behind the ARROW (in its previous position). Line 90 checks if a key is being pressed and if the ARROW is directly above the base, then it sends the action to the HIT routine at line 140. This adds one to the number of hits scored and provides an explosive-looking display at the point where the craft was hit, then sends the program to line 40 again to supply another one. However, if a key was pressed and the ARROW was not above the base, then one is added to the number of misses. If the ARROW reaches the end of its travel, the program falls out of the FOR/NEXT loop and line 120 erases the final position of the craft then line 130 sends the program back to line 40 to set up a new one.

Garbage gobbler

The object of this game is to clear up garbage which appears on the screen as magenta shapes. You control the 'gobbler' with the keys 5,6,7, & 8, movement being in the direction of the arrows printed on the keys. You eat the garbage by running over it. You are given a limited time, and you are told how many items of garbage you collected. The loop in lines **40** to **60** set up the garbage in random positions.

```
  10 REM GARBAGE GOBBLER
  20 GO SUB 1000
  30 REM GRAPHICS M IN 50
  40 FOR G=1 TO 50
  50 PRINT AT RND*15+3,RND*26+4;
INK 1;"M"
  60 NEXT G
  70 LET X=0
  80 LET Y=0
  85 FOR G=1 TO 500
  90 LET A=X
 100 LET B=Y
 110 LET Y=Y-(INKEY$="7" AND Y>1
)+(INKEY$="6" AND Y<20)
 120 LET X=X-(INKEY$="5" AND X>1
)+(INKEY$="8" AND X<31)
 130 IF ATTR (Y,X)=49 THEN LET S
CORE=SCORE+1: PRINT AT 0,20; INK
 RND*7; PAPER 9; FLASH 1;"Score
is ";SCORE;AT 20,0;"Time left =
";500-G;"  "
 140 IF ATTR (Y,X)=49 THEN BEEP
.01,40
 150 REM GRAPHIC D IN 160
 160 PRINT AT B,A;" ";AT Y,X; IN
K 2;"D"
 170 NEXT G
 180 PRINT AT 0,20; INK RND*7; P
APER 9; FLASH 1;"Score is ";SCOR
E;AT 20,0;"Time left = ";0;"  "
 190 BEEP .1,RND*50: GO TO 190
 900 STOP
1000 FOR A=0 TO 7
1010 READ B
1020 POKE USR "M"+A,B
1030 NEXT A
1040 DATA BIN 10100101,BIN 01011
010,BIN 10100101,BIN 01011010,BI
N 01011010,BIN 10100101,BIN 0101
1010,BIN 10100101
1050 FOR A=0 TO 7
1060 READ B
1070 POKE USR "D"+A,B
1080 NEXT A
```



```
1090 DATA BIN 00011111,BIN 01111
111,BIN 11111100,BIN 11111000,BI
N 11100000,BIN 11111000,BIN 0111
1100,BIN 00111111
1100 BORDER 2: PAPER 6: CLS
1120 LET SCORE=0
1500 RETURN
```

Now we can explain how to find out what is on the screen, so the computer knows the 'gobbler' is about to run over something. This facility will be used time and time again in programs. The vital function is ATTR which, as you can see from lines 130 and 140, is similar in form to PRINT AT. Lines 130 and 140 check the attributes of the character square Y, X before the 'gobbler' is printed there. If it finds a value of 49, it knows the square contains garbage, and so the time and score is updated. The number actually produced by ATTR depends on whether the character is FLASHING or not, whether it is BRIGHT or not, and the colours of the INK and PAPER at the position.

Because ATTR is a little difficult to use, it is best to set up a small routine to print out the results of ATTR before you finally decide what value you want to test for. The 'absence' of a value produced when the 'gobbler' moves over a blank background is not recommended; instead of this, test for the present of a particular value. We did this when writing this program, by having line 140 blank at first, and line 130 reading PRINT AT 0,0; ATTR(Y,X), then watching what happened when the 'gobbler' was about the land on a garbage square. We recommend you follow this method.

String manipulation

Finally, there is one other method of producing moving graphics in BASIC which is not used very often — through the use of strings. Strings may be PRINTed very rapidly, and the computer's comprehensive string slicing facilities (discussed earlier in the book) mean that we have at our disposal a powerful tool. The basic method is to assign a character or string array large enough to cover the area of screen used for movement and PRINT the array at a certain location. To simulate movement we can either PRINT different parts of the array or change the contents of the array.

(1) PRINT different parts of the string in order. Try this short program:

```
 10 DIM A$(32)
 20 LET A$(1)="+"
 30 FOR A=32 TO 1 STEP -1
 40 PRINT AT 20,0;A$(A TO )+A$(
TO A-1)
 50 NEXT A
 60 GO TO 30
```

Can you see why it is necessary to have line 30 count *backwards* from 32 to 1? What would happen if line 30 counted from 1 to 32 (30 FOR A = 1 TO 32)? Draw on a piece of paper (or use the printer if you have one) every step the program will take to make up the display. Note the high speed possible, and how the previous position is erased and the new position PRINTed in one go. An interesting effect may be obtained by changing line 20 to 20 INPUT A$ and entering a message of up to 32 characters. This is similar to a type of display found in shop windows for advertising purposes, although this is not the kind of effect one would normally encounter.

This method of producing moving graphics from strings is very useful because it does not alter the contents of strings/arrays, but rather only displays them in a different order, so the information may be retrieved easily at any time. It can be used for a shop window display like the one below.


```
  20 INPUT "ENTER YOUR MESSAGE";
A$
  30 IF A$="" THEN GO TO 20
  40 LET S=50
  50 FOR B=1 TO LEN A$+33
  70 PRINT AT 5,0;("
                        "+A$+"
                              ")(B T
O B+31)
  80 LET S=S-(5 AND INKEY$="F" A
ND S>1)+(5 AND INKEY$="D")
  90 IF INKEY$="A" THEN GO SUB 1
40
 100 FOR A=1 TO S
 110 NEXT A
 120 NEXT B
 130 GO TO 50
 140 FOR A=1 TO 200
 150 NEXT A
 160 RETURN
```

What it does is ask you to enter a display message, then after it has been entered, the message begins to appear from the right of the screen and moves to the left and eventually, disappears to the left, whereupon the sequence begins all over again. The rotating sequence begins at a slow speed but can be speeded up by pressing the F key or slowed down by pressing the D key. The fastest speed is very fast, the slowest is very slow. You can "freeze" the display for a short time by pressing the A key. You can stop the program at any time by pressing BREAK.

Provided that the computer has enough memory available then in theory the size of the message is limited by the size of the largest array the computer can handle in theory. In practice, however, if you fill the screen with the message when entering it (i.e. it is more than 24 rows of 32 characters long) and subsequent characters entered have to be entered blind because they will seem to be below the screen — try it to see what I mean. Warning — you'll end up with a tired finger!

The message begins to appear from the right of the screen about a quarter of the way down, and runs along the screen towards the left. Once it has disappeared past the left-hand side of the screen it reappears as before and repeats the cycle

over and over again. The speed may be varied as described. The display is frozen as described.

Line 70 is fairly complex: to prevent changing the contents of string A$, the entire contents in the first pair of brackets are treated as one long string consisting of thirty two spaces followed by the message string A$, followed in turn by another thirty two spaces. The slicer in the second pair of brackets selects which parts of this long string are PRINTed. Note that A$ still retains its own identity. Whichever parts are selected, the string PRINTed is always 32 characters long.

As it stands, the program has no facility for you to change the message once it is running — you have to use BREAK then RUN the program once again. One way of providing this facility is to add this line, so that on pressing "1" (EDIT) the program restarts automatically:

```
65 IF INKEY$="1" THEN RUN
```

(2) Move the elements of the array around. Try this program:

```
10 DIM a$(32)
20 FOR a=1 TO 32
30 LET a$(a)=a$(a+1)
40 PRINT AT 20,0;a$
50 LET a$(a)=" "
60 NEXT a
70 GO TO 20
```

This method gives us great flexibility. We can handle strings quickly and efficiently with the computer's string handling facilities. Strings are very useful for storing information because this information may be accessed quickly and conveniently compared with REM statements for instance, and the speed with which they may be PRINTed makes them an attractive method for producing displays. The main disadvantage is that it is wasteful of memory since the information is held in both the display file and the arrays involved, possibly in the program area as well. Here is a moving display program which relies on information in the strings PRINTed.

The program is called BASIC Invaders because it's a version of the arcade game written in BASIC. It is very simplified of necessity and is included for the purpose of demonstrating the use of strings for moving display purposes. A row of invaders descends the screen towards you. You can move left or right using the 5 & 8 keys respectively. You fire up at the invaders by pressing the 7 key. If you are directly under an invader it disappears and is destroyed.

There are seven waves of invaders and you have to destroy them all to win.

```
  1 REM Basic Invaders
  2 REM Graphic F in 40
  3 REM Graphic B in 120
  4 BORDER 2: PAPER 0: CLS
  5 GO SUB 230
 10 DIM a$(32)
 20 DIM b$(32)
 30 FOR d=1 TO 7
 40 LET a$=a$+"  Ж  Ж  Ж  Ж  Ж
  Ж"
 50 LET X=INT (RND*32)
 60 LET C=X
 70 PRINT AT 5,12; INK X/5; PAP
ER 9; FLASH 1; BRIGHT 1;" WAVE:
";D;" "
 80 LET C=0
 90 FOR B=D+9 TO 19 STEP 2
100 FOR A=0 TO 31
110 LET X=X+(INKEY$="8" AND X<3
1)-(INKEY$="5" AND X>0)
120 PRINT AT B,0; INK RND*8;A$;
AT 20,C;" " AND C<>X;AT 20,X; IN
K 4;"Ж"
130 IF A$=B$ THEN GO TO 200+(60
AND D=7)
140 LET C=X
150 IF INKEY$="7" THEN BEEP .01
,20: IF A$(X+1)="Ж" THEN LET A$(
X+1)="■": PRINT AT B,0; INK 6;A$
: BEEP .01,50: LET A$(X+1)=" ": L
LET SC=SC+1: PRINT AT 2,19; INK
2; PAPER 0; BRIGHT 1; FLASH 1;"S
cors ";SC*27187
160 NEXT A
165 LET A$=A$(4 TO )+A$( TO 3)
170 PRINT AT B,0;B$
180 NEXT B
190 GO TO 240
200 FOR B=1 TO 10
210 NEXT B
220 PRINT AT 20,C;" "
230 NEXT D
```

```
  240 PRINT FLASH 1; INK RND*7; P
APER 9;"THEY HAVE LANDED!"
  250 BEEP .1,-RND*30: POKE 23692
,-1: GO TO 240
  260 PRINT INK RND*7; PAPER 9;"A
LL INVADERS DESTROYED"
  265 POKE 23692,-1
  270 BEEP .01,RND*50: GO TO 260
  280 REM Make invaders
  290 FOR J=0 TO 7
  300 READ Q
  310 POKE USR "F"+J,Q
  320 NEXT J
  330 DATA BIN 00111110,BIN 00101
010,BIN 00111110,BIN 00011100,BI
N 00001000,BIN 01110111,BIN 0100
0001,BIN 01000001
  340 REM Make base
  350 FOR J=0 TO 7
  360 READ Q
  370 POKE USR "B"+J,Q
  380 NEXT J
  390 DATA BIN 01111110,BIN 01111
110,BIN 01111110,BIN 01111110,BI
N 01100110,BIN 11100011,BIN 1100
0011,BIN 10000001,BIN 10000001
  400 LET SC=0
  410 RETURN
```

The program is very wasteful of memory. No attempt has been made to conserve memory, and you may be able to speed it up with some minor modifications. The two strings of interest are A$ and B$. B$ is merely set up to 32 spaces and is used to prevent typing in "(32 spaces)" at various points in the program. A$ is the string that represents the invaders. It is initially set to 32 elements, the number of characters in one line of display. Line 40 sets the initial state of the characters and can be any combination of spaces and graphic — and should consist of 32 characters. X is the variable that controls your position and its value is altered in line 110. Line 120 performs the main PRINTing, updating the invaders display and your position.

Line 130 compares the invaders with a string of 32 spaces (B$) and if it finds A$ contains no invaders (i.e. it is all spaces) it either causes a jump to the next wave of invaders or if you're already on the last wave it causes a jump to the victory message at line 260. Line 150 is of particular interest, since it scans the string for the character above you in the invader

display, and if it finds an invader there, it converts it to a space. This is only done if the 7 key is pressed.

The rest of the program is mainly concerned with timing of loops and sorting out the different waves of invaders.

If you're storing the entire screen in a string array (or a part of the screen involving lines above or below each other), then you can use two different methods using different types of array. Consider the case of the full 22 by 32 screen. You will require a 22 by 32 element string array and this may be accomplished by either:

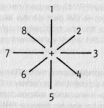
(1) using a two-dimensional array, set up with the statement DIM A$(22,32).You can then use the PRINT AT coordinates to access the elements, remembering the array elements start with 1, PRINT coordinates with 0. For example, to PRINT AT Y,X;CHR$ T you would say LET A$(Y + 1,X + 1) = CHR$ T. The problem with this method is that a lengthy PRINT statement is required to place the entire array on the screen, i.e. PRINT AT 0,0;A$(1),A$(2),A$(3). . . . A$(21),A$(22). However, since the main reason for using an array for printing is the ease of access of information, this is only necessary at the very beginning of a program since from then on we need only PRINT the parts of the array we're actually dealing with. Take the example of a game of draughts. We would need the entire board on screen at the start of the game, and we need to be able to examine the board in its entirety; however, when it comes to PRINTing moves we need only PRINT the parts of the array which are changed by a move — the part of the array from which the piece was moved, the part to which it is moved to and possibly a part of the array where an opposing piece was captured.

(2) Use a single dimension array with 704 elements for the 22 by 32 display, set up by the statement DIM A$(704). This can be treated like a memory mapped screen and may be PRINTed in one go by the statement PRINT AT 0,0;A$ and is very fast to execute. The elements are easily accessed. To move a character on screen we have to move it about in the

array in a manner which makes it move satisfactorily on screen. To understand how to do this we have to understand the layout of the array on the screen. Here is a diagram:

X / Y	0	1	2	29	30	31
0	A$(1)	A$(2)	A$(3)	A$(30)	A$(31)	A$(32)
1	A$(33)	A$(34)	A$(35)	A$(62)	A$(63)	A$(64)
2	A$(65)	A$(66)	A$(67)	A$(94)	A$(95)	A$(96)
3	A$(97)	A$(98)	A$(99)	A$(126)	A$(127)	A$(128)

The diagram shows a fragment from the top of the screen. Y and X are the coordinates of the PRINT AT Y,X; function. Can you see how the Y,X; coordinates can be related to the subscripts of A$? There are 32 elements of A$ in each line of the display. The X coordinates start off at 0 whereas the array subscripts start at 1. So Y,X corresponds to A$(Y* 32 + X + 1). When moving a character it can go to one of eight locations all around it, as shown in the diagram below:



Now suppose the invader is at A$(A). Here is a chart showing how much there difference is beween subscripts of the elements representing the possible positions on all sides (how much to add to the old subscript to make it the new one).

Direction of movement	How much to add to A
1	−32
2	−31
3	1
4	33
5	32
6	31
7	−1
8	−33

Shifted INKEY$

At this stage we have to be careful not to go beyond the limits of the array, as we would when using PRINT, to avoid crashing the program with a subscript error. We can use the cursor keys to control vertical movement and horizontal movement, and use the SHIFTed cursor keys for diagonal movement. Here is a short program which moves a blot around the screen under cursor control to illustrate how this may be done.

Pressing SHIFT-5, SHIFT-6, SHIFT-7, SHIFT-8 moves you 45° clockwise to the direction of the arrows on the keys. The 5,6,7,8 keys by themselves cause movement in the direction of the arrows. This program does not have the facility to prevent subscript errors occuring due to moving outside the boundaries of the array.

```
10 DIM a$(704)
20 LET a=INT (RND*704)+1
30 LET a$(a)=" "
40 LET a=a-(32 AND INKEY$="7")
-(31 AND INKEY$=CHR$ 11)+(1 AND
INKEY$="8")+(33 AND INKEY$=CHR$
9)+(32 AND INKEY$="6")+(31 AND I
NKEY$=CHR$ 10)-(1 AND INKEY$="5"
)-(33 AND INKEY$=CHR$ 8)
50 LET a$(a)="■"
60 PRINT AT 0,0;a$
70 GO TO 30
```

The reason why SHIFT-5, SHIFT-6, SHIFT-7, SHIFT-8 have been represented by CHR$ 8, CHR$ 10, CHR$ 11 and CHR$ 9 is that they cannot be entered directly from the keyboard (they act as cursor control if you try), so the easiest way to get them in is by means of CHR$.

It should be emphasised that the use of strings to create moving graphics is limited to those applications where speed is not of great importance, rather than applications where the access of information is important but speed of graphics is important but not of highest priority. An example is a 'board game such as draughts where pieces move occasionally but it is necessary to have rapid access to the information.

Introduction to arithmetic on the computer

Have a quick glance at this section before you read it in detail. You may well find it has no new information for you. If this is the case, feel free to turn to the next section.

The symbols for the various operations in BASIC are probably well known to you by now. They are multiply (*), divide (/), subract (−), add (+) and raise to the power (↑ — SHIFT H). The computer follows a strict priority for operations.

The term priority refers to the order in which expressions are evaluated. The computer may not evaluate everything in the order in which they are printed on the screen. For example you may have noticed that putting an expression in brackets could produce a different result compared with the same expression evaluated with the brackets omitted (indeed leaving out brackets may cause the computer to refuse to do anything because of a syntax error). We formalise this by giving each operation a *priority*, a number between 1 & 16.

The operations with highest priority are evaluated first, and operations with equal priority are evaulated in order from left to right. In effect, the computer looks at the expression, and finds the part with the highest priority and says to itself: "Wait a minute fellers, there's something with a higher priority than you over there — I'll come back to you in a minute when it's your turn."

Operation	*Priority*
()	16
SUBSCRIPTING/SLICING	12
ALL FUNCTIONS	11
↑	10
− (negation)	9
*	8
/	8
+	6
− (substraction)	6
=,>,<,<=,>=,<>	5
NOT	4
AND	3
OR	2

Note that a number is assumed to be positive unless it is preceded by a minus sign. Similarly, unless a decimal point appears with a number, the computer will assume it is an integer. Although you can use decimal points when working with the computer, commas are not allowed. The use of *scientific notation* for very large and very small numbers was explained in the section on variables. Refer back to that if you need a reminder on how this works.

The BASIC on this computer works fairly quickly, as can be seen by running the following programs. The first program in this section works out arithmetic progressions. You must enter the first term, the common difference, and the number of terms, and the computer will produce the information for you very rapidly.

```
  10 REM Arithmetic progression
  20 PRINT "I will work out for
you the"
  30 PRINT "arithmetic progressi
on from the"
  40 PRINT "information you give
 me"
  50 PRINT '"Enter the first ter
m"
  60 INPUT first
  70 PRINT ;first
  80 PRINT '"And the common diff
erence?"
  90 INPUT diff
 100 PRINT ;diff
 110 PRINT '"How many terms?"
 120 INPUT terms
 130 LET terms= INT (terms+.5)
 140 CLS
 150 POKE 23692,-1
 160 PRINT INK 6; PAPER 0;"Arith
metic progression"
 180 PRINT INK 2;"  Term number"
;TAB 13;"Value"
 190 LET count=0
 200 FOR L=0 TO terms-1
 210 LET w=L+1
 220 LET 0=first+(L*diff)
 230 LET count=count+0
 240 PRINT TAB 4;w;TAB 13;q
 250 NEXT L
 260 PRINT INK 2;TAB 4;'"The sum
 is ";count
```

```
I will work out for you the
arithmetic progression from the
information you give me

Enter the first term
            234

And the common difference?
            13.5

How many terms?
```



```
Arithmetic progression
 Term number  Value
     1          234
     2          247.5
     3          261
     4          274.5
     5          288
     6          301.5
     7          315
     8          328.5
     9          342
     10         355.5
     11         369
     12         382.5
The sum is 3699
```

As you can see, the program also works out the sum of the terms.

Prime numbers

Prime numbers are very easy to determine.

```
  10 REM prime numbers
  20 PRINT "Enter the value of t
he"
  30 PRINT " maximum prime numbe
r"
  40 PRINT TAB 4;"that you want"
  50 INPUT a: PRINT a
  60 DIM z(a): LET k=2
  70 FOR j=1 TO a: LET z(j)=j
  80 NEXT j: IF a<4 THEN GO TO 2
00
  90 LET z(4)=5: LET k=4
 100 LET iz=5
 110 LET iz=iz+2
 120 IF iz>a THEN GO TO 200
 125 LET jo=3
 130 LET ex=iz/z(jo)
 140 IF ex=INT ex THEN GO TO 110
 150 IF ex<z(jo) THEN GO TO 180
 160 LET jo=jo+1
 170 GO TO 130
 180 LET k=k+1: LET z(k)=iz
```

```
100 GO TO 110
200 POKE 23692,-1
210 PRINT "The prime numbers up
to ";a
230 PRINT TAB 4; INK 2; PAPER 6
;"Prime no.","Prime"
240 FOR c=1 TO k1
250 PRINT TAB 4;c,z(c)
260 NEXT c

Enter the value of the
 maximum prime number
   that you want
19
The prime numbers up to 19
    Prime no.     Prime
        1             1
        2             3
        3             5
        4             7
        5            11
        6            13
        8            17
        9            19
```

The mathematical ability of the computer can also, of course, be turned to produce other kinds of information.

Statistics

This next program, a series of statistical routines, can easily be broken down into four shorter programs.

The four programs are:

Arithmetic mean - This is simply the average of a set of numbers.

Geometric mean - The geometric mean is the nth root of the product of the numbers, where n is the total number of numbers entered.

Harmonic mean - The harmonic mean is derived from the reciprocals of the entered numbers.

Factorial - A factorial is the progression A*(A-1)*(A-2)*(A-3) ..., down to *(2)*(1), where A is the integer entered in line 4030. As this only works with integers, line 4040 changes any non-integer entry into an integer.

The routine from line 9000 presents a menu of choices. Note the use of GOTO A*1000 in line 9130. This is a shorthand way of saying

```
IF A = 1 THEN GOTO 1000
IF A = 2 THEN GOTO 2000
IF A = 3 THEN GOTO 3000
IF A = 4 THEN GOTO 4000
```

You can often make use of this technique in menu-driven programs.

```
  20 GO TO 9000
  30 REM QUOTE MARKS WITHIN
  LINES 1040, 2040, 3040
    ARE FROM THE A KEY
 900 REM ************************
1000 REM ARITHMETIC MEAN
1010 PRINT "ARITHMETIC MEAN"
1020 PRINT "ENTER THE NUMBERS YO
U WISH ME"
1030 PRINT TAB 5;"TO AVERAGE FOR
 YOU"
1040 PRINT "ENTER "E" TO END YOU
R INPUT"
1060 INPUT Q$: IF Q$="" THEN GO
TO 1060
1070 IF Q$="E" THEN GO TO 1120
1080 POKE 23692,-1: PRINT Q$
1090 LET TOTAL=TOTAL+VAL Q$
1100 LET COUNT=COUNT+1
1110 GO TO 1060
1120 PRINT '"THE ARITHMETIC MEA
N IS ";TOTAL/COUNT
1130 GO TO 9000
1900 REM ************************
2000 REM GEOMETRIC MEAN
2010 PRINT "GEOMETRIC MEAN"
2020 PRINT "ENTER THE NUMBERS YO
U WISH ME"
```

```
2030 PRINT "TO USE TO FIND GEOME
TRIC MEAN"
2040 PRINT "ENTER "E" TO END YOU
R INPUT"
2050 LET TOTAL=1
2060 INPUT 0$: IF 0$="" THEN GO
TO 2060
2070 IF 0$="E" THEN GO TO 3120
2080 LET COUNT=COUNT+1
2090 LET TOTAL=TOTAL*VAL 0$
2100 POKE 23692,-1: PRINT 0$
2110 GO TO 2060
2120 PRINT '"THE GEOMETRIC MEAN
IS "(TOTAL↑(1/COUNT)
2130 GO TO 9000
2900 REM **********************
3000 REM HARMONIC MEAN
3010 PRINT "HARMONIC MEAN"
3020 PRINT "ENTER THE NUMBERS YO
U WISH ME
3030 PRINT "TO USE TO FIND HARMO
NIC MEAN"
3040 PRINT "ENTER "E" TO END YOU
R INPUT"
3050 INPUT 0$: IF 0$="" THEN GO
TO 3050
3060 IF 0$="E" THEN GO TO 3120
3070 POKE 23692,-1: PRINT 0$
3080 LET TOTAL=TOTAL+(1/VAL 0$)
3100 LET COUNT=COUNT+1
3110 GO TO 3050
3120 PRINT '"THE HARMONIC MEAN I
S ";1/(TOTAL/COUNT)
3130 GO TO 9000
3900 REM **********************
4000 REM FACTORIAL
4010 PRINT "FACTORIAL"
4020 PRINT "ENTER AN INTEGER ",,
"LESS THAN 34"
4030 INPUT NUM: IF NUM>=34 THEN
GO TO 4030
4040 LET NUM=INT (NUM)
4050 LET A=1
4060 FOR B=1 TO NUM
4070 LET A=A*B
4080 NEXT B
4100 PRINT '"THE FACTORIAL OF ",
NUM;" IS ";A
8900 REM **********************
9000 PRINT '"SELECT THE PROGRAM
YOU WANT"
9010 PRINT '"1 - ARITHMETIC MEAN

9020 PRINT "2 - GEOMETRIC MEAN"
9030 PRINT "3 - HARMONIC MEAN"
9040 PRINT "4 - FACTORIAL"
9050 PRINT "5 - TO END": PRINT
9060 LET A$=INKEY$
```



```
9070 IF A$<"1" OR A$>"5" THEN GO
TO 9060
9080 LET A=VAL A$
9090 IF A=5 THEN STOP
9100 LET TOTAL=0
9110 LET COUNT=0
9120 PRINT "**********************
**********"
9130 GO TO A*1000
```

It can perform far more complex tasks. This program, for example, calculates the mean, standard deviation, standard error of the mean and variance of up to 20 frequency distributions with up to 240 items in each. It can also calculate any significant difference between any two frequency distributions with the help of Student's T-test.

```
  10 REM T-TEST
  12 REM CONVERTED FROM A
PROGRAM BY ALLAN NORLIN
  13 REM BY ANDERS ØLUND
  20 DIM X(240)
  22 DIM N(20)
  24 DIM M(20)
  26 DIM S(20)
  28 DIM E(20)
  30 DIM V(20)
  32 DIM G(20)
 110 PRINT "THIS PROGRAM CALCULA
TES THE MEAN"
 115 PRINT ",STANDARD DEVIATION,
"
 118 PRINT "STANDARD ERROR AND"
 120 PRINT "VARIANCE FOR A NUMBE
R OF"
 123 PRINT "FREQUENCY DISTRIBUTI
ONS."
 124 PRINT "(MAXIMUM 20 DISTRIBU
TIONS)"
 130 PRINT "AND STUDENTS T-TEST
TO"
 135 PRINT "EVALUATE IF THERE IS
"
 140 PRINT "A SIGNIFICANT DIFFER
ENCE"
 141 PRINT "BETWEEN ANY TWO OF T
HE"
 142 PRINT "FREQUENCY DISTRIBUTI
ONS"
 144 PRINT
 145 PRINT "DO YOU WANT THE RESU
LTS ONLY ON"
 147 PRINT "THE DISPLAY (D) OR A
LSO ON "
 148 PRINT "THE PRINTER (P)"
```

```
 140 INPUT U$
 150 CLS
 160 LET G=0
 161 LET G=0
 162 LET S=0
 170 PRINT "HOW MANY DISTRIBUTIO
NS?"
 180 INPUT M
 200 LET G=G+1
 205 PRINT
 220 PRINT "FREQUENCY DISTRIBUTI
ON NO.";G
 245 PRINT "HOW MANY ITEMS OF DA
TA?"
 250 INPUT N
 255 PRINT
 265 LET N(G)=N
 268 LET S1=0
 269 LET S2=0
 285 PRINT "INPUT DATA WITH NEW
LINE AFTER"
 270 FOR I=1 TO N
 271 CLS
 280 PRINT "NO ";I
 290 INPUT X
 295 CLS
 300 LET S1=S1+X
 310 LET S2=S2+X*X
 320 LET X(I)=X
 330 NEXT I
 340 LET M(G)=S1/N
 345 LET V(G)=((S2-S1*S1/N)/(N-1
))
 350 LET S(G)=SQR V(G)
 360 LET E(G)=S(G)/SQR (N)
 380 PRINT "DO YOU WANT THE DATA
PRINTED?    Y/N"
 390 INPUT O$
 395 CLS
 400 IF O$="N" THEN GOTO 550
 405 PRINT "FREQUENCY DISTRIBUTI
ON NO.";G
 410 LET O=O+1
 415 LET R=1
 417 LET K=0
 420 FOR I=1 TO N
 425 PRINT AT R,K;I;".";X(I)
 430 IF I/20=INT (I/20) THEN LET
 K=K+10
 431 IF K>21 AND U$="P" THEN COP
Y
 432 IF K>21 THEN GOSUB 3000
 435 IF K>21 THEN LET K=K-30
 450 IF I/20=INT (I/20) THEN LET
 R=R+20
 455 LET R=R+1
 480 NEXT I
 490 IF U$="P" THEN COPY
 500 GOSUB 3000
 550 IF G<M THEN GOTO 200
```



```
 740 FOR I=1 TO M
 750 PRINT "FREQUENCY DISTRIBUTI
ON NO:";
 752 PRINT
 755 PRINT "NUMBER OF DATA = ";N
(I)
 758 PRINT
 760 PRINT "MEAN = ";M(I)
 762 PRINT
 765 PRINT "STANDARD DEVIATION =
 ";S(I)
 767 PRINT
 770 PRINT "STANDARD ERROR = ";E
(I)
 772 PRINT
 775 PRINT "VARIANCE = ";V(I)
 776 IF U$="P" THEN COPY
 777 GOSUB 3000
 780 NEXT I
 790 PRINT
 900 PRINT "DO YOU WANT T-TEST?
Y/N"
 810 INPUT O$
 815 CLS
 820 IF O$="N" THEN GOTO 1000
 830 PRINT "T-TEST BETWEEN TWO D
ISTRIBUTIONS"
 835 PRINT
 840 PRINT "INPUT NO OF FIRST DI
STRIBUTION"
 845 INPUT S1
 847 PRINT "NO:";S1
 849 PRINT
 850 PRINT "INPUT NO OF SECOND D
ISTRIBUTION"
 852 INPUT S2
 854 PRINT "NO:";S2
 890 LET A=N(S1)+N(S2)
 900 LET B=N(S1)*N(S2)
 910 LET D=A-2
 920 LET T=ABS (M(S1)-M(S2))/SQR
 (((N(S1)-1)*S(S1)↑ 2+(N(S2)-1)*
S(S2)↑ 2)*A/(B*D))
 930 GOSUB 2000
 935 CLS
 940 PRINT "T-TEST FOR DISTRIBUT
IONS          NO:";S1;" AND NO:";
S2
 945 PRINT
 950 PRINT "T=";T
 955 PRINT
 960 PRINT "DF=";D
 965 PRINT
 970 PRINT "P=";P
 975 PRINT
 980 IF P>0.001 THEN GOTO 990
 982 PRINT "P(0.001    ***"
 984 GOTO 1050
 990 IF P>0.01 THEN GOTO 1000
 992 PRINT "P(0.01    **"
```

```
  994 GOTO 1050
 1000 IF P>0.05 THEN GOTO 1010
 1002 PRINT "P<0.05    *"
 1004 GOTO 1050
 1010 PRINT "P>0.05  NS."
 1050 IF U$="P" THEN COPY
 1055 PRINT
 1056 PRINT
 1057 PRINT
 1058 PRINT
 1060 PRINT "DO YOU WANT TO TEST
MORE"
 1065 PRINT "DISTRIBUTIONS?  Y/N"
 1070 INPUT Q$
 1075 CLS
 1080 IF Q$="N" THEN GOTO 1200
 1090 GOTO 840
 1200 PRINT "DO YOU WANT TO ANALY
SE MORE" "FREQUENCY DISTRIBUTION
S?  Y/N"
 1210 INPUT Q$
 1215 CLS
 1220 IF Q$="Y" THEN GOTO 120
 1240 PRINT "###*THE END*###"
 1245 GOTO 9999
 2000 REM CALCULATES T-VALUE, P-
VALUE (PROBABILITY) AND DF-VALUE
(DEGREES OF FREEDOM)
 2020 LET P=1
 2025 LET M=1
 2030 IF T=0 THEN GOTO 2210
 2040 LET U=T*T
 2050 IF U<1 THEN GOTO 2100
 2060 LET I=M
 2070 LET J=D
 2080 LET T1=U
 2090 GOTO 2130
 2100 LET I=D
 2110 LET J=M
 2120 LET T1=1/U
 2130 LET A1=2/9/I
 2140 LET B1=2/9/J
 2150 LET Z=ABS ((1-B1)*T1↑(1/3)
-1+A1/SQR (B1*T1↑(2/3)+A1)
 2160 IF J>3 THEN GOTO 2180
 2170 LET Z=Z*(1+.08*Z↑4/J↑3)
 2180 LET P=.5*(1+Z*(.196854+Z*(.
115194+Z*(.000344+Z*.019527))))
 ↑4
 2190 IF U>1 THEN GOTO 2210
 2200 LET P=1-P
 2210 RETURN
 3000 PRINT AT 21,0;"TO CONTINUE
PRESS C"
 3004 IF INKEY$<>"C" THEN GOTO 30
04
 3007 CLS
 3010 RETURN
 9999 STOP
```


Species

The final program in this section uses the computer to simulate the life cycles of two species, one of which preys upon the other, and to graph their relative populations. The relationship between the two species is controlled by a differential equation. You enter the starting populations, as numbers between one and five. Fractions are acceptable, and it is fascinating to enter a very low population for one of the animals, and a high one for the other, and watch the two evolve. When the program has run through a specified number of generations, it will generate another starting population for the two species. The development of this relationship will then be graphed, on top of the existing graph, so you can build up a number of graphs showing the effects of different starting populations for the predator and its prey.

```
   5 BORDER 2: PAPER 5: CLS
  10 REM Species
  20 PRINT "HOW MANY OF SPECIES
ONE";"(1 TO 5)?"
  30 INPUT X: IF X<1 OR X>5 THEN
 GO TO 30
  40 PRINT '"AND OF SPECIES TWO
?";"(1 TO 5)?"
  50 INPUT Y: IF Y<1 OR Y>5 THEN
 GO TO 50
  55 LET X=X+RND: CLS
  60 FOR Z=1 TO 20
  70 FOR T=1 TO 7 STEP .5
  80 PRINT AT 1,1; "Species one
"; FLASH 1; BRIGHT 1;"  ";INT (X
*10000)
  90 PRINT AT 2,1; INK 1;"Specie
s two: "; FLASH 1; BRIGHT 1; INK
 2;"  ";INT (Y*10000);"  "
 100 LET X=X+(4*X-2*X*Y)*0.01
 110 LET Y=Y+(Y*Y-3*Y)*0.01
 120 PLOT 20*X,30*Y
 130 NEXT T
 135 BEEP .05,Z+T
 140 NEXT Z
 150 LET X=RND*5
 160 LET Y=RND*5
 180 GO TO 60
```

Functions

The computer's dialect of BASIC, in common with other BASICs, contains a number of pre-programmed functions which you can use in a program, or in the direct mode. The following discussion includes a program which uses a 'defined function' to draw a picture of a bat!

General functions:

ABS — This function, ABSolute, gives the value of X, ignoring the sign, so that if X was −10, ABS(X) would be 10. Similarly, if X was 10, ABS(X) is still 10.

INT — The INT function gives the whole number, or INTeger part of a number, giving the largest number which is not greater than X. If X was 2.42, INT(X) would be 2.

INT rounds off numbers to the next lowest whole number, e.g. INT 2.2 is 2, INT 2.9 is 2, INT 2 is 2 and so on. A frequent requirement is to round off numbers to the nearest whole number, so that 2.6 becomes 3, etc. (some commands do this automatically, e.g. PRINT AT, POKE). This is quite easy to do. Suppose the number to be rounded off is A. If we first add 0.5 to A, then apply INT, the answer will be the nearest whole number. If as an example, A was 2.6 and we wished to round off to the nearest whole number: PRINT INT (2.6+0.5) would give 3 whereas PRINT INT (2.3+0.5) would give 2. PRINT INT (2.5+0.5) is rounded *up* to 3.

It is often necessary when doing monetary calculations to have answers to two decimal places to resemble dollars and cents values. This routine does this. It also inserts a 0 before the decimal point if the answer is less than $1.00. Enter the amount as A in line 10, in units of dollars, but do not enter the $ symbol — that will be added by the routine in line 50.


```
  10 INPUT A
  20 LET A$=STR$ (INT (A*100+0.5
)/100)
  25 IF A$(1)="." THEN LET A$="0.
"+A$
  30 LET B=LEN A$-LEN STR$ INT V
AL A$
  40 LET A$=A$+(".00" AND B=0)+(
"0" AND B=2)
  50 PRINT "$";A$
  60 GO TO 10
```

```
11.245      $11.25
33          $33.00
33.1        $33.10
333.888     $333.89
0.9999      $1.00
22.009      $22.01
```

RND — This is used to generate a RaNDom number. RND gives a random number between zero and one.

SGN — This function returns the SiGN of the variable in brackets, the SiGN of the *argument* as this variable is known. If X equals 20, that is, X is a positive number, SGN(X) = 1. SGN(−20) = −1. SGN(0) = 0.

TAB — As pointed out earlier in the book this is the TABulating function, which moves the PRINT position across the line the number of spaces indicated by the argument of the function. Thus, PRINT TAB(6); "$" will print the $ at the seventh position across from the left-hand edge, while PRINT TAB(13);"$" will print it 14 spaces across. The direction down the screen can also be specified, by adding a second argument after a comma within the brackets. Thus, PRINT TAB(4,9);"$" will print a dollar sign five spaces down, and ten across. TAB reduces a number modulo 32, meaning that the argument of (number after) TAB can be larger than 31; it will be reduced to a number in the range 0 to 31 and the PRINT position moves on the same line unless this would involve backspacing in which case it moves onto the next line. This modulo business means is that the argument of TAB is divided by

32 (the number of columns per line on a screen) and the remainder taken. You may be able to take advantage of this when the PRINT spacing is determined by calculation since you do not have to ensure that the number falls in the range 0 to 31.

EXP — This function gives the value of e raised to the power of the argument, so PRINT EXP 5 will give 148.41316.

LN — LN X yields the natural logarithm to base e, so PRINT LN 5 gives 1.6094379.

SQR — This function yields the SQuare Root of a number, so when X is five, PRINT SQR X gives 2.236068.

Trigonometrical functions:

SIN — This gives the SINE of an angle in radians. SIN 5 yields −0.95892428.

COS — Yields the COSine of an angle in radians. PRINT COS X where X equals five gives 0.28366219.

TAN — Produces the TANgent of angle X in radians.

(The computer measures angles in radians. PI radians equals 180 degrees).

The RANDOMIZE function works as follows:

The number you place after the word RANDOMIZE is stored in the system variables after being rounded off to the nearest whole number. If you just enter RANDOMIZE or RANDOMIZE0 then it is given the value of the frame counter. This value is *not* affected by CLEAR or RUN, but is reset to 0 by NEW, as it is at switch-on. It changes every time you use RND.

Converting other Basics

A wealth of computer programs written in BASIC can be found in a variety of books and computing magazines, but as all versions of BASIC differ to some extent, it is unlikely that a program written to run on another computer will work on your computer without some change. The extent and nature of these changes will depend greatly on the structure of the particular program and how it handles data, but it is possible to give some general guidance on things to look for when approaching the task of converting a "foreign" program to run on your computer.

Integer Arithmetic

In general, always add the function INT before a division in a program designed for a computer with integer arithmetic. You may require brackets around the division so that INT works only on the result of that division.

DIM

Some BASICs allow you to write several DIM statements on one line such as DIM A$(9), B$(8), C$(7). You will have to replace this by individual DIM statements on separate program lines. If the program calls for arrays with names that are more than one letter long, then these have to be replaced by single letter names like A$ or B. If you do not have enough letters available then you may be able to declare additional dimensions to the existing ones for a certain array and use the extra dimension to replace an array. Beware of zero subscripts.

GET, GET$

This is a function that reads characters or values from keys pressed on the keyboard. It takes various forms on various computers, but in general it waits until a key is pressed before it goes on, assigning either the character corresponding to the key pressed or the code of that character to a variable. For example, GET A$ or LET A$=GET$. You could do this on your computer:

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

This would return the character corresponding to the key pressed on the keyboard. If the function was to return the CODE (which will probably be written as ASC) of the character then use this routine:

```
1000 LET a$=INKEY$
1010 IF a$="" THEN GO TO 1000
1020 LET a=CODE a$
1030 PRINT a
```

Slightly different is the version that returns a numeric value rather than a character code. It is necessary to ensure that the character read from the keyboard is in the range "0" to "9" so that we can apply VAL to convert the character to a number. Here's one way:

```
1000 LET a$=INKEY$
1010 IF a$<"0" OR a$>"9" THEN GO
TO 1000
1020 LET a=VAL a$
1030 PRINT a
```

You may also come across a version of INKEY$ which allows a time limit to be specified for an user response, e.g.

```
100 LET A$ = INKEY$(X)
```


where X specifies the time limit. This can be converted in two ways, either as:

```
 10 LET x=50
100 PAUSE x
110 LET a$=INKEY$
```

or as follows, demonstrated in a simple game:

```
  5 REM ENGAGE CAPS LOCK FIRST
 10 LET b$=CHR$ (INT (RND*26)+C
ODE "A")
 20 PRINT AT 10,0;"QUICKLY, PRE
SS ";b$
 100 FOR a=0 TO 100
 110 LET a$=INKEY$
 120 IF a$<>"" THEN GO TO 140
 125 NEXT a
 130 PRINT "Time is up": STOP
 140 IF a$=b$ THEN PRINT "You we
re right": STOP
 150 PRINT "You were wrong"
```

VAL

If the argument of VAL does not form a valid numerical argument, you get an error report. Other BASICs return 0.

SET, RESET

These are used to make a particular screen point white or black. Replace with a PLOT/OVER/PRINT AT.

ELSE

This is an extension to the IF...THEN conditional statement and allows more than one outcome depending on whether the conditional statement is true or false. It may be replaced by two conditional expressions on your computer. For example:

```
20 IF X=1 THEN LET Y=7 ELSE GOTO 80
```

may be replaced by:

```
20 IF X=1 THEN LET Y=7
21 IF X<> 1 THEN GOTO 80
```

If the action of ELSE is to assign one of several alternative values to a variable then it can be replaced on one line, e.g.

```
50 IF X=1 THEN LET Y=7 ELSE LET Y=8
```

may be replaced by:

```
50 LET Y=(7 AND X=1) + (8 AND X<>1)
```

Certain expressions such as the one above may be replaced by even shorter forms such as:

```
50 LET Y=7 + (1 AND X <>1)
```

No general guideline can be given since the method used will vary from example to example — the examples above give an idea of what to expect.

You may come across a statement where the action performed by ELSE is itself conditional:

```
10 IF X=1 THEN LET Y=1 ELSE IF X=5 THEN GOTO
100
```

This will need to be rewritten as either:-

```
10 IF X =1 THEN LET Y=1
11 IF X<> 1 THEN IF X=5 THEN GOTO 100
```


or:-

```
10 IF X=1 THEN LET Y=1'
11 IF X<> 1 AND X=5 THEN GOTO 100
```

Again you may meet all sorts of conditional ELSEs, and the Spectrum versions will depend on the variation encountered.

REPEAT....UNTIL

This is a loop that performs an operation continuously ending only when a specified condition is met. Its use is so wide it is difficult to specify a universal method of conversion to ZX BASIC, probably the best being the IF...THEN GOTO conditional statement. Here is an example:

```
10 PRINT "ENTER YES OR NO"
20 REPEAT
30 INPUT A$
40 UNTIL A$="YES" OR A$="NO"
```

may be replaced by:

```
10 INPUT "Enter yes or no ";a$
20 IF a$<>"yes" AND a$<>"YES"
AND a$<>"no" AND a$<>"NO" THEN G
O TO 10
```

REPEAT....UNTIL structures are generally far more complex than this example, and it may be necessary to find a means of conversion other than IF...THEN GOTO. For example, where the value of a variable is the determining factor, a FOR/NEXT loop may sometimes be used. However, the possibility of using an IF...THEN GOTO conditional statement should always be considered and is sometimes the only acceptable method of conversion.

Undefined variables

If you attempt to use a variable before it has been defined or assigned in a program, then some computers will return a value of 0. You get an error report on your computer if the variable has not previously been assigned. So all variables must have been assigned when using programs on your computer which use variables.

Matrices

Some BASICs have matrix functions which perform operations on arrays. Your computer does not have these functions, so you will have to perform the operations on array elements individually, possibly by means of a loop.

```
10 DIM X(Y)
20 DIM P(Y)
30 MAT X=P
```

This particular example can be replaced by:

```
10 LET n=0
20 DIM x(y)
30 DIM p(y)
40 LET n=n+1
50 IF n<y THEN GO TO 40
```


PROC, ENDPROC

This is a method of using subroutines to do certain procedures in such a way that among other things makes programs and listings easier to understand and read (it is called structured programming by some). It enables subroutines to be used specifically to do certain things and it is like a subroutine in many ways, but with the important exception that it is called by a name rather than by its line numbers. Take this example, which prints the score on the screen: —

```
100 PROCscore
.....
1000 DEF PROCscore
1010 PRINT "SCORE =";S
1020 ENDPROC
```

ENDPROC is in a way similar to RETURN in that the procedure comes to an end and the program resumes from the line after the one which called the procedure, in this case the line after line 100. The name of the procedure is not used in your computer's version, although it can be adapted for the purpose as the second example Sinclair version will show. The simplest method of conversion to ZX BASIC is for line 100 to GOSUB line 1000, possibly with a REM statement somewhere in the subroutine to identify it, and end the subroutine with a RETURN command.

```
 100 GOSUB 1000
.....
1000 REM SCORE SUBROUTINE
1010 PRINT "SCORE=";S
1020 RETURN
```

If you want to retain the procedure/subroutine naming facility you can use a variable of the same name as the PROC name assigned during the course of the program before the subroutine is called, and use this variable as the destination for the GOSUB command. You could include a REM statement in the subroutine to identify the subroutine and tie it up with the variable name used. It is useful to use inverse

characters in these REM statements so that they stand out from the rest of the listing text. So you can make your programs seem fairly structured.

```
  50 LET SCORE=1000
 100 GOSUB SCORE
.....
1000 REM SCORE SUBROUTINE
1010 PRINT "SCORE=";S
1020 RETURN
```

Although PROCs may be complex, an ordinary subroutine is the best method of conversion to your BASIC using GOSUB/RETURN.

INSTR(A$,B$)

This is a function that looks to see if there is a copy of B$ in A$, and if there is it tells you where the copy starts. For instance, if B$ was "PUT" and A$ was "COMPUTER" then the value of INSTR(A$,B$) would be 4 because the part of A$ which held the letters "PUT" started at the fourth element of A$. If the function does not find a copy of B$ in A$, the INSTR(A$,B$) has a value of 0. A special routine has to be written to provide this function on your computer.

Here is one method of converting this function to run on the Spectrum:

```
HAPPY P
INSTR(A$,B$)=3
HAPPY M
INSTR(A$,B$)=1
HAPPY SAD
INSTR(A$,B$)=0
SINCLAIR BASIC
INSTR(A$,B$)=0
SINCLAIR SIN
INSTR(A$,B$)=1
```



```
  10 REM -- LET Y=INSTR(A$,B$) --
  20 INPUT A$
  30 INPUT B$
  40 PRINT A$;" ";B$
  50 GO SUB 1000
  60 PRINT "INSTR(A$,B$)=";Y
  70 GO TO 20
1000 REM SUBROUTINE FOR 'INSTR'
1010 LET Y=0
1020 IF LEN A$=0 OR LEN B$=0 OR
LEN B$>LEN A$ THEN RETURN
1030 FOR Y=1 TO LEN A$-LEN B$+1
1040 IF A$(Y TO Y+LEN B$-1)=B$ T
HEN RETURN
1050 NEXT Y
1060 LET Y=0
1070 RETURN
```

Note that if you want to detect whole words rather than just strings then you will have to examine A$ for space or punctuation marks that signify the start and end of words. The routine above just finds matching strings, so that if you wanted to find the word CAT in a phrase containing the word CATASTROPHIC, this would trigger on the first three letters of CATASTROPHIC. However, users of INSTR usually have this problem so the program will cater for this anyway!

DIV

DIV gives the whole number part of the result of a division, for example, 17 DIV 5 gives 3. INT can be applied to the result of the division on your computer. So A DIV B on the Spectrum would be INT (A/B).

MOD

MOD gives the *remainder* of a division, e.g. 17 MOD 5 is 2. A MOD B is A — (INT (A/B*B) on the Spectrum. Note that TAB carries out its own MOD action (modulo 32) on your computer.

TAB

Some computers may have two arguments to the TAB function, which is used to space out along the screen. This use of TAB conforms to your computer's use of AT. For example, TAB (X,Y) on some computers would correspond to AT Y,X; on your machine. The X and Y coordinates may be in reverse order on some computers.

Degrees and radians

Your Spectrum deals with trigonometrical functions in radians by this expression:

```
LET RADIANS = (PI * DEGREES)/180
```

and radians may be converted to degrees by:

```
LET DEGREES = (180 * RADIANS)/PI
```

Base 10 logarithms

As your computer works in natural logs, to base e, if you need logs base 10 for any reason, these may be found using the expression:

```
LET LOGBASE10 (X) = (LN(X)/(LN(10))
```

You could use this to find logs (any base), suppose you wanted the log base B of X:

```
LET LOGBASEB (X) = (LN(X))/(LN(B))
```


%

The percentage symbol is generally used to specify an integer variable, e.g. A%. These are usually used to save memory or because they can be processed faster than conventional variables. In general, there is no harm to using an ordinary variable, although you should be wary of these integer variables being assigned as the result of a division as they automatically truncate the quotient to its integer value. In this case use LET A = INT (A/2) for example to "integerise" the result of the division.

?

On most computers the symbol ? is used as an abbreviation for the command PRINT.

PEEK and POKE

These two commands are very powerful instructions that enable you to do things with your computer that you might not be able to do otherwise. Let us start by defining the two terms PEEK and POKE.

(1) PEEK m gives us the numbers stored at address m in memory.
(2) POKE m,n puts number n into memory at address m. When accepted, it erases what used to be there.

The term address needs explaining. A computer like yours thinks and remembers in numbers, not words as people do. Certain patterns of numbers make parts of the computer do specific things. This is called a program. Now the computer

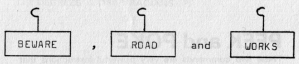
needs a way to hold all these numbers so that they are remembered until needed and can then be looked at, and once their values and patterns are known, the computer can decide what it's going to do.

Certain patterns of numbers may make the computer PRINT something on the screen, add two numbers together or maybe crash if the pattern of numbers makes it try to do something it can't or shouldn't.

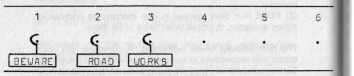
The computer can't just stuff the numbers anywhere — this would cause chaos if it didn't know where to look next. So there is a method used whereby everything can be neatly organised.

Imagine you wanted to display a message in public and you had the words written on little placards with hooks on, ready for all occurrences, so that you could display any message simply by hanging up the right set of placards. For example, if we wanted to display this message:

"BEWARE ROAD WORKS" we need these placards:



We need a board on which to hang up these words. If we start with the first hook by hanging the first placard there and proceed along the board, we end up with a fairly neat sign:



The pegs on the board tell us where each word is hung. This makes a good comparison with your computer's memory. There are 651536 places where we can "hang" numbers on the Spectrum, but these are split up for various uses, and you or the computer can do various things with these. These "pegs" or locations or whatever you want to call them are actually referred to as addresses (the home of each number if you like). However, if the Spectrum has a number it wants to store, it can't just stuff it anywhere because it might upset what's already there.

One way you've already used POKE is to create user-defined graphics. These are POKEd into addresses starting at 32600 on a 16K Spectrum.

If we look at the first peg (PEEK 1) we find the word BEWARE there. If we look at the second peg (PEEK 2) we find the word ROAD there, and so on. Can you see the analogy? Remember that the computer would use numbers rather than words, but the idea is still the same. Similarly, we can change the words on the pegs quite easily by using POKE to stuff a new number where another number used to be. We could do something like POKE peg 2, BUILDING which would put the word BUILDING on the second peg of the notice, and so change its entire meaning. The great secret about PEEK and POKE is not what they do — it's how to use them. It's all very well finding what number is in which address or stuffing a new number into a particular address, but how can you make use of this in a program and how do you know where to PEEK and POKE. The answer is, mainly by experience and reading through other people's programs, although you will find that as your knowledge of computers increases, you will find you think up new ways to use PEEK and POKE. Before we look at examples, a brief reminder of how to write/type PEEK & POKE statements.

PEEK m. m is the address which we're looking into. m is a number from 0 to 65560 or m can be the result of a calculation. POKE m,n. m is an address to which the new number is to be placed, as with PEEK. It is written between the word POKE and the comma. The number after the

comma, n, is the number to be placed in address m, and can be a number from 0 to 255 or the result of a calculation provided it is a number from 0 to 255. You can actually make n a negative value from 0 to −255, but this is rarely done and is not very useful anyway.

Let us now look at some examples of PEEK and POKE in use.

(1) REM statements

Many programs rely on information held in REM statements in the first line of a BASIC program on the computer. This is because it is easy to access and is very economical on memory. The important point is that the address of the first character after the word REM in the first line of a program is 23760. So if you had the program:

```
1 REM ABCDEF
2 PRINT PEEK 23760
```

it would print the number 65 on the screen. This is the CODE of the character A, so address 23760 has the value of 65.

You can easily change this value by POKEing a new value into 23760. For example, to change A to a Z, look up the CODE of Z, which is 90, and POKE this into 23760:

```
POKE 23760, 90
```

Or it could have been written as POKE 23760, CODE "Z" which works just as well. The next address, 23761 stores the B, 23762 stores the C and so on.

The technique of PEEKing/POKEing into REM statements was of great importance for the storage of machine code programs on the ZX81.

Although it can still be used, the manual (in the section called 'Using Machine Code') explains a way of reserving memory, and POKEing machine code into the reserved area.

(2) Using the timer

The timer is contained in addresses 23672, 23673 and 23674 and it simply counts the number of frames sent to the T.V. You can PEEK and POKE into these addresses.

Reset the counter to zero by these statements:

```
POKE 23674, 255
POKE 23673, 255
POKE 23672, 255
```

And to read its value, use this expression:

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

This gives us an answer in frames, and since 50 frames are sent to the T.V. every second (60 in the US), we need to divide by 50 (or 60) to get an answer in seconds, like this:

```
LET TIME = (65536*PEEK 23674 + PEEK 23672 + 256* PEEK 23673)/50
```

Here is a program to provide a stopwatch:

```
  5 POKE 23674,255
 10 POKE 23673,255
 20 POKE 23672,255
 30 LET time=(65536*PEEK 23674+
PEEK 23672+256*PEEK 23673)/50
 40 PRINT AT 11,14;INT (time*10
)/10;
 50 GO TO 30
```

The INT in line 40 is added to prevent fractions of a second less than a tenth being printed. This stopwatch keeps fairly accurate time because the frame counter is controlled by special hardware, so unless the program deliberately forces it to do otherwise it is independent of how fast the program runs and keeps time fairly well. The counting range of the frame counter allows timing for nearly four days. If you want a readout in minutes and seconds then use this routine:

```
  5 POKE 23674,255
  10 POKE 23673,255
  20 POKE 23672,255
  30 LET time=65536*PEEK 23674+
PEEK 23672+256*PEEK 23673)/50
  40 PRINT AT 11,12;INT (time/60
);":";INT (time-INT (time/60)*60
);"  "
  50 GO TO 30
```

Business Uses

The computer can be used for a number of small business applications. A wide variety of programs are commercially available to exploit the large potential of the computer. In this section, we'll look at some simple practical application programs for your Sinclair machine.

The first one is for money manipulation. James Walsh has written a program to calculate compound interest. The user prompts are clear, and easy to follow.

```
  90 LET A$="Year     Interest
Total"
 100 INPUT "Number of years? ";Y
 110 PRINT "Compound Interest"
 120 PRINT '"Over ";Y;" years"
 130 INPUT "Amount? ";A: LET T=A
 140 PRINT '"Principal is $";A"
 150 INPUT "Interest per year?
";I
 160 CLS : PRINT AT 1,0;
 170 FOR N=1 TO Y
 180 POKE 23692,-1
 200 GO SUB 340
 210 PRINT N;TAB 6;"$";INT (V+.5
);TAB 19;"$";INT (T+.5)
 250 NEXT N
 270 PRINT '"Total=$";INT (T+.5)
 290 PRINT '"Interest= ";IN;"%"
 310 PRINT '"Original amount=$";
A
 320 PRINT AT 0,0;A$
 330 STOP
 340 LET V=1+(IN/100)*T
 350 LET T=(IN/100)*T+T
 360 RETURN
```


Word processing

This word processor program will make text neat and tidy before you print it — and gives you the chance to correct mistakes, using a free-moving cursor. You enter your text (up to 17 lines deep) as a single string, X$. When you have the text in, you press ENTER, and the computer will shuffle the words to ensure that none of them are split at the end of a line.

A menu appears with three options: 1 — correct the text; 2 — LPRINT the text; and 3 — to start again. If you decide you wish to correct the text, it will reappear on the screen, with the words "ENTER 1 TO RETURN TO MENU" above it. You use the 5, 6, 7 and 8 keys to move the cursor in the direction indicated by the arrows on those keys, and the cursor moves along the line of text, putting a black blob over the letter it is passing over. Once you find a letter which is wrong you press "A" and the words ENTER LETTER appear at the bottom of the screen. You enter your letter, and press ENTER, and the incorrect letter will be altered to the letter you've chosen. Pressing "1" at any time will return you from the 'correction phase' to the original menu, and from this menu you can choose "2" to LPRINT the text.

After LPRINTing, you are shown a further menu, which allows you to run the whole program again from scratch, LPRINT again, or to terminate the run.

```
  10 REM WORD PROCESSOR
  20 PRINT "ENTER TEXT"
  30 INPUT X$
  32 LET X$=X$+"
                "
  35 CLS
  40 GO SUB 1000
  45 GO SUB 1000
  50 PRINT X$
  70 PRINT '"ENTER 1 TO CORRECT
TEXT, 2 TO LPRINT, 3 TO S
TART AGAIN"
  80 INPUT 0
 100 IF 0=3 THEN RUN
 110 IF 0=2 THEN GO TO 4000
 120 IF 0=1 THEN GO TO 2000
 130 GO TO 80
```

```
1000 REM STOPS WORDS SPLITTING
1010 LET N=1
1020 GO SUB 1180
1030 LET N=N+33
1040 IF N>=LEN X$ THEN RETURN
1045 REM SINGLE SPACE IN
       NEXT LINE
1050 IF X$(N)=" " THEN GO TO 116
0
1065 REM SINGLE SPACE IN
       NEXT LINE
1070 IF X$(N)=" " THEN GO TO 103
0
1080 LET J=0
1090 GO SUB 1180
1100 LET J=J+1
1105 REM SINGLE SPACE IN
       NEXT LINE
1110 IF X$(N)>" " THEN GO TO 109
0
1120 FOR N=N TO N+J-1
1125 REM SINGLE SPACE IN
       NEXT LINE
1130 LET X$=X$(1 TO N)+" "+X$(N+
1 TO )
1140 NEXT N
1150 GO TO 1030
1160 LET X$=X$(1 TO N-1)+X$(N+1
TO )
1170 GO TO 1020
1180 LET N=N-1
1190 RETURN
2000 REM # CORRECTIONS #
2010 CLS
2020 PRINT "ENTER 1 TO RETURN TO
 MENU"
2030 LET A=1: LET L=LEN X$
2035 LET Z$=A$(1)
2035 PRINT AT 2,0;X$
2037 LET X$(A)=Z$
2050 IF INKEY$="6" AND A<L THEN
LET A=A+1
2055 IF INKEY$="6" AND A<L+32 TH
EN LET A=A+32
2060 IF INKEY$="5" AND A>1 THEN
LET A=A-1
2065 IF INKEY$="7" AND A>32 THEN
 LET A=A-32
2070 IF INKEY$="1" THEN GO TO 70
2080 IF INKEY$="A" THEN GO SUB 3
000
2090 PRINT AT 1,0;A;" ";X$(A);"
  "
2100 LET Z$=X$(A)
2110 LET X$(A)="■"
2120 GO TO 2035
3000 INPUT INK 2; FLASH 1; BRIGH
T 1;" Enter letter ";H$
3010 LET X$(A)=H$
```



```
3030 RETURN
4000 LPRINT X$
4010 PRINT "ENTER 1 TO PRINT AGA
IN"
4020 PRINT TAB 5;"2 TO RUN AGAIN
"
4030 PRINT TAB 5;"3 TO STOP"
4040 INPUT U
4050 IF U=1 THEN GO TO 4000
4060 IF U=2 THEN RUN
4070 IF U=3 THEN STOP
4080 GO TO 4040
```

The final program in this section is designed to place entries and page references in alphabetical order and will enable indexes to be constructed (e.g. to books or articles in magazines) or can be easily adapted to accommodate stock lists or levels.

The program is in three parts. The first (to line 100) accepts the entries, the second (lines 200 to 300) sorts them and the third (lines 310 to 530) displays the data.

The program asks you to enter a title (T$) and its author (N$) and then enter the subjects and pages, one by one, entering an "E" to end the entry process. The program will accept up to 400 entries (line 20). This runs on a 16K Spectrum. You can have 2000 entries or more on a 48K machine. At the end, you can choose to have them printed to the screen or to the printer.

Here is a sample run:

```
BRAIN GAMES

FISHER R B

CONSCIOUSNESS - 189
INTELLIGENCE - 7
LEARNING - 114
MACHINERY - 32
MEMORY - 92
PERCEPTION - 187
PERSONALITY - 9
```

```
  10 REM Book index
  20 DIM A$(400,16)
  30 INPUT "ENTER TITLE ";T$
  40 INPUT "ENTER AUTHOR'S NAME
";N$
  50 FOR G=1 TO 500
  60 INPUT "ENTER WORD AND PAGE
NUMBER      ENTER 'E' TO END ENT
RY";A$(G)
  70 REM 16 SPACES IN NEXT LINE
  80 IF A$(G)="E
THEN GO TO 200
  90 PRINT A$(G)
 100 NEXT G
 200 CLS
 210 PRINT "STAND BY, SORTING"
 220 FOR B=1 TO G-1
 230 FOR C=B+1 TO G-1
 240 IF A$(B)<=A$(C) THEN GO TO
260
 250 LET D$=A$(B)
 260 LET A$(B)=A$(C)
 270 LET A$(C)=D$
 280 NEXT C
 290 NEXT B
 300 PRINT ''"READY"
 310 PRINT "ENTER 1 TO LPRINT LI
ST"
 320 PRINT ''"ENTER 2 TO PRINT ON
 SCREEN"
 330 IF INKEY$="2" THEN CLS : GO
TO 410
 340 IF INKEY$="1" THEN GO TO 36
0
 350 GO TO 330
 360 LPRINT T$
 370 LPRINT
 380 LPRINT N$
 390 LPRINT
 400 FOR A=1 TO G-1
 410 LPRINT A$(A)
 420 NEXT A
 430 STOP
 440 PRINT 'T$
 450 PRINT 'N$
 460 PRINT
 500 FOR A=1 TO G-1
 520 PRINT A$(A)
 530 NEXT A
```


Improving your programs

You've probably gone through several stages as you develop your programming skills. After the first, brief struggle with BASIC, you suddenly discovered you could, after a fashion, write programs which ran. They may have looked pretty convoluted when you looked at their listings, and friends may have needed a detailed explanation from you before they knew what to do when running the programs, but at least they worked.

There comes a stage when you decide you're going to have to do better than that. But while you may be vaguely dissatisfied with your programs, you may not have much idea of how to go about becoming a better programmer. Here are a few guidelines which may help.

First, have a look at a printout of your listing. Programs linked by REM statements look better, and are easier to understand when you return to them after a break. Of course, shortage of memory may preclude the luxury of REM statements, but if you have the memory, you should include them. REM statements filled just with a line of asterisks can prove quite useful in separating each major section of the program. Examine any unconditional GOTO critically. Too many GOTOs leapfrogging over other parts of the program show a lack of directed thinking, make programs run more slowly, and can make them almost impossible to decipher.

It is very good programming practice, to have each of the main sections of the program (like the one which assigns the variables at the beginning of a run, the one which prints out the board, the one which works out who has won, and so on) in separate subroutines. The beginning of your program could well look like this:

```
10  REM *NAME OF PROGRAM*
20  REM ASSIGN VARIABLES
30  GOSUB 9000
```

```
40  REM PRINT BOARD
50  GOSUB 8000
60  REM HUMANS MOVE
70  GOSUB 7000
80  REM COMPUTERS MOVE
90  GOSUB 6000
100 REM CHECK IF GAME OVER
110 GOSUB 5000
120 GOTO 50
```

As you can see, this ensures that the program actually cycles through a continuous loop over and over again, until the program terminates within the "CHECK IF GAME OVER" subroutine. You can actually write a series of lines like these before you start writing anything else, and even before you know how you are going to actually perform some of the tasks within the subroutine.

Then you can write the program module by module, making sure that each module works before going onto the next. It is relatively easy to debug a program like this, and far simpler to keep an image of 'where everything is' when you do this, than when you just allow a program to, more or less, write itself.

The listing should be, then, as transparent as you can make it, both for your own present debugging, and for future understanding of what bit carries out what task. The output of the program should also look good. Again, if memory is not a problem, make sure the display is clear and uncluttered. Use blank PRINT lines to space it out, use rules of graphic symbols or whatever to break the screen up into logical sections and so on. Once you have a program working satisfactorily, it is worth spending extra time on the subroutine which controls the display. Here you'll appreciate again the advantage of having all the display handling in one subroutine, as it will be easy to know where to go to enhance the display.

Of course, as we live in a far from ideal world, it is unlikely that every single display command can be contained within

one subroutine, but if you aim towards that end, it will make subsequent working upon the program much easier than it might be otherwise.

The 'structured' approach outlined also helps you realise another aim of a good program — to do what you expected it to, every time you run it. You should write a program so that, even if you are not present when a friend decides to run it for the first time, it performs as expected. This means not only, of course, that it is properly debugged, but that the instructions (which can be contained within the ASSIGN VARIABLES subroutine) are clear and complete.

The user prompts should be clear, so the human operator knows whether to enter a number, a series of numbers, a word, a date, a mixture of letters and numbers, and so on. The program has to assume that the operator is a complete fool, and that no matter how clearly the instructions and/or user prompts are stated, he or she will attempt to do things the wrong way. A classic example of this is the entering of dates. 'Mug traps', as the routines to reject erroneous input from the operator are called, should be set up to reject a date being entered in a form which the computer cannot understand (such as the month before the day) or which is clearly wrong (such as entering the 32nd of February). You should ensure that, no matter what the operator does, the program does not crash or otherwise misbehave. This can happen if the program was expecting a numerical input, and the operator tried to enter a letter or a word, or hit ENTER without entering anything at all. You can get around this by always allowing a string input, going back for another input if the empty string is entered, and taking the VAL or CODE of the input to turn it into numerical form.

Documentation is an area of programming which is often neglected. It is virtually essential for a program which is intended for publication, and most advisable for long programs which you've written for yourself. At the least, the documentation should include a list of variables, an explanation of the program structure (which should be easy to do if you've followed the 'modular' approach advised), and

brief instructions, especially if the program itself does not contain instructions. A sample run showing the kind of inputs, and the nature and layout of the program outputs, is also useful.

Your program should run as quickly as possible. Every time there is a subroutine or GOTO call, the computer must search through the whole program, line by line, to find the specified line number, so placing often used subroutines near the beginning of the program will speed them up fractionally. That is why the instructions are often placed right at the end. You do not want the computer to have to wade through the initialisation and instruction lines every time it has been told to GOTO or GOSUB looking for the destination, or return line number.

Define often-used variables first, so they will occupy the early slots in the variables store. The computer will search the store only until it finds the variable it wants, so there is no point in getting it to look at more entries than absolutely necessary.

Finally, and this is by far the best way to test a program you've written, call in a friend and sit him or her in front of the TV, and tell them to press RUN, without you saying anything, and just sit back and watch. If there is any hesitation, or the program hiccups, you have more work to do.

In summary then:

- Use REM statements
- Make program listings neat and logical
- Use structured programming techniques, controlling the program through a loop of subroutine calls
- Examine unconditional GOTO commands critically
- Make output display attractive and clear
- Ensure all user prompts are clear
- Add 'mugtraps' on all user input
- Document your programs, even if you just make a list of variables

- Make your program run as quickly as possible
- Test programs by allowing someone unfamiliar with the program to run it

Programs, programs, programs

Finally in this book are some programs you may enjoy entering and running.

```
  1 REM Greyhound
  2 REM © Gourlay,Hartnell 1982
  5 RANDOMIZE
  7 GO SUB 200
 10 FOR x=1 TO 22
 20 PRINT INK 4;TAB 30;"▟▙"
 30 NEXT x
 35 PRINT AT 0,6;"You bet on nu
mber ";w
 40 DIM a(9)
 50 FOR x=1 TO 9
 60 PRINT AT 2*x,a(x);" "
 70 LET a(x)=a(x)+RND*2
 80 PRINT AT 2*x,a(x); INK x/2;
x
 85 BEEP .01,3*x
 90 IF a(x)>30 THEN GO TO 115
100 NEXT x
110 GO TO 50
115 FOR z=1 TO 50 STEP 2
120 PRINT AT 18,6; INK RND*7;x;
" is the winner!"
122 BEEP .02,z
125 IF w=x THEN PRINT AT 20,3;
INK RND*7;"And you are the winne
r !!!";z
130 FLASH RND
140 NEXT z
145 FLASH 0
150 RUN
200 BORDER 0
205 PRINT AT 3,1; INK 2;"Welcom
e to the greyhound tract"
210 PRINT AT 5,6; INK 4;"There
are nine dogs."
220 INPUT ( INK 2;"Place your b
et for a win     ");w
```

```
  230 IF w<1 OR w>9 THEN GO TO 22
0
  235 BORDER 2
  240 CLS : RETURN


   10 REM Star bouncer
   15 REM @ Hartnell, 1982
   20 BORDER 0
   30 LET a=1: LET b=1: LET c=RND
+20: LET d=RND*20
   40 PRINT AT c,d;"*";AT c,d; IN
K RND*7;"*"
   50 IF c+b>21 OR c+b<-1 THEN LE
T b=-b: BEEP 0.2,-RND*15
   60 IF d+a>31 OR d+a<0 OR RND>.
96 THEN LET a=-a: BEEP 0.1,RND*2
5
   70 LET c=c+b: LET d=d+a
   80 GO TO 40
```

Colourthello

Challenge your Spectrum to a game of Reversi with this program COLOURTHELLO, written by Graham Charlton. COLOURTHELLO is intended to highlight the sound and colour potential of the Spectrum. You'll see, when you run the program, how effective the Spectrum's features can be. You move by entering the number down the side, followed by the number across the top or bottom, as a single, two-digit number. For example, if you wanted to place a piece where the bottom "O" is on the board, you'd enter 64.

```
    1 REM ____Colourthello____
    5 PRINT AT 0,12; INK 2;"C"; I
NK 1;"o"; INK 6;"l"; INK 5;"o"; I
NK 4;"u"; INK 6;"r"; INK 2;"t"; I
NK 1;"h"; INK 6;"e"; INK 5;"l"(
; INK 4;"l"; INK 2;"o"
   10 DIM a(10,10): FOR b=1 TO 10
: FOR c=a TO 10
   20 BEEP .01,b*c/10
   40 IF b<>1 AND c<11 AND b<>10
AND c<>10 THEN LET a(b,c)=CODE "
*"
```



```
   50 NEXT c: NEXT b: LET p=0: LE
T r=0
   70 LET a(5,5)=CODE "x": LET a(
6,6)=CODE "x": LET a(5,6)=CODE "
o": LET a(6,5)=CODE "o"
  120 INPUT ( INK 2;"Do you want
to go first? ")a$
  125 CLS : GO SUB 3000
  127 PRINT AT 0,12; INK 2;"C"; I
NK 1;"o"; INK 6;"l"; INK 5;"o"; I
NK 4;"u"; INK 6;"r"; INK 2;"t"; I
NK 1;"h"; INK 6;"e"; INK 3;"l"(
; INK 4;"l"; INK 6;"o"
  130 IF CODE a$<>CODE "n" AND CO
DE a$<>CODE "N" THEN GO TO 2000
1000 PRINT INK 2;AT 10,16;"MY GO
"
1010 LET s=CODE "o": LET t=CODE
" ": LET h=0
1040 FOR a=2 TO 9: FOR b=2 TO 9
1050 IF a(a,b)<>CODE " " THEN GO
 TO 1070
1070 LET q=0: FOR c=-1 TO 1: FOR
 d=-1 TO 1: LET k=0: LET f=a: LE
T g=b
1150 IF a(f+c,g+d)<>s THEN GO TO
 1180
1140 LET k=k+1: LET f=f+c: LET g
=g+d: GO TO 1130
1180 IF a(f+c,g+d)<>t THEN GO TO
 1200
1190 LET q=q+k
1200 NEXT d
1210 NEXT c
1220 IF f=2 OR f=9 OR g=2 OR g=9
 THEN LET q=q*2
1230 IF f=3 OR f=8 OR g=3 OR g=8
 THEN LET q=q/2
1250 IF (f=2 OR f=9) AND (g=3 OR
 g=8) OR (f=3 OR f=8) AND (g=2 O
R g=9) THEN LET q=q/2
1280 IF q<h OR q=0 OR (RND)<.3 AN
D q<>h+1 THEN GO TO 1320
1290 LET h=q: LET m=a: LET n=b
1300 NEXT b
1310 NEXT a
1340 IF h=0 AND r=0 THEN GO TO 5
000
1350 IF h=0 THEN GO TO 1370
1360 GO SUB 4000
1370 GO SUB 3000
2000 PRINT INK 1;AT 10,16;"YOUR
GO"
2010 LET s=CODE "x": LET t=CODE
 "o"
2030 INPUT r
2040 IF r=0 THEN GO TO 2090
2050 IF r<11 OR r>88 THEN GO TO
 2030
2060 LET m=INT (r/10)+1: LET n=r
-10*INT (r/10)+1
```

```
2050 GO SUB 4000
3000 GO SUB 3000: GO TO 1000
3000 PRINT AT 5,0,: BEEP .25,RND
*50
3010 LET c=0: LET h=0
3030 PRINT INK 4;" 123456789 "
3040 FOR b=2 TO 9: PRINT INK 4;b
-1;
3060 FOR d=2 TO 9
3070 IF a(b,d)=CODE "x" THEN PRI
NT INK 2;"x";
3075 IF a(b,d)=CODE "o" THEN PRI
NT INK 1;"o";
3077 IF a(b,d)=CODE "." THEN PRI
NT INK 5;".";
3080 IF a(b,d)=CODE "x" THEN LET
c=c+1
3090 IF a(b,d)=CODE "o" THEN LET
h=h+1
3100 NEXT d
3110 PRINT INK 4;b-1
3120 NEXT b
3130 PRINT INK 4;" 123456789 "
3150 PRINT '' INK 3;"I have ";h; I
NK 2;c; INK 3;"  You have "; IN
K 1;h-c; INK 3;"."
3170 RETURN
4000 FOR c=-1 TO 1
4010 FOR d=-1 TO 1
4020 LET f=b: LET g=d
4040 IF a(f+c,g+d)<>s THEN GO TO
 4060
4050 LET f=f+c: LET g=g+d: GO TO
 4040
4060 IF a(f+c,g+d)<>t THEN GO TO
 4140
4090 LET a(f,g)=t: IF f=f AND b=
g THEN GO TO 4140
4110 LET f=f-c: LET g=g-d: GO TO
 4090
4140 NEXT d: NEXT c: RETURN
5000 IF c>h THEN PRINT "I won, "
;c;" - ";h
5010 IF h>c THEN PRINT "You won,
";h;" - ";c
5050 © Charlton 1982
```



```
 123456789
1....x....1
2....x....2
3...xox...3
4...oox...4
5..xox...5
6...o.....6
7.........7
8.........8
 123456789
```

Your go

I have 7 You have 4

Life

Here are two versions of John Conway's game of LIFE, the game which simulates the birth, growth and death of a cell colony. The cells evolve according to the following rules:

- Each cell on the grid has eight neighbours
- Every cell with two or three neighbours survives to the next generation
- If there are three, and only three, neighbouring cells, a new cell is born
- Any cell with four or more neighbours dies from over population

```
  5 REM LIFE - © ANNE MARSHALL
 10 DIM A(145): DIM L(145): DIM
B(8)
 15 LET G=0
 20 FOR T=1 TO 8
 25 READ Z: LET B(T)=Z: NEXT T
 30 LET Z=CODE "o": LET Z=128
 35 BORDER 1: PAPER 0: CLS
 40 FOR B=1 TO 12
 50 FOR D=1 TO 12
 60 LET A(B+10*D)=Z
 70 IF RND>.45 THEN LET A(B+10*
D)=0
 80 LET L(B+10*D)=A(B+10*D)
```

```
  30 NEXT D: NEXT B
 100 LET G=G+1
 120 FOR U=1 TO 12
 130 FOR B=1 TO 12
 140 LET F=U+10*B
 150 IF G=1 THEN GO TO 250
 190 LET H=0
 200 FOR T=1 TO 8
 210 IF A(F+E(T))=C THEN LET H
=H+1
 220 NEXT T
 230 IF A(F)=C AND H<>3 AND H<>2
 THEN LET L(F)=Z
 235 IF A(F)=Z AND H=3 THEN LET
L(F)=C
 240 NEXT B: BORDER RND*7: NEXT
U
 245 BORDER 1
 250 FOR M=11 TO 144: LET A(M)=L
(M): BEEP .005,M/3: NEXT M
 255 PRINT AT 5,0;
 260 FOR U=1 TO 12: PRINT TAB 4;
 270 FOR B=1 TO 12: LET F=U+10*B
 280 PRINT INK 6;CHR$ A(F);" ";:
NEXT B: PRINT : NEXT U
 285 PRINT AT 3,10; PAPER 8; INK
 5;"Generation ";G: BEEP .3,50
 290 GO TO 100
 300 DATA 11,10,9,1,-1,-9,-10,-1
1

  10 REM Conway's Colony
  15 REM © Hartnell, 1982
  20 GO SUB 90
  30 LET print colony=200:
  40 LET generation update=320
  45 REM ******************
  50 GO SUB print colony
  60 GO SUB generation update
  70 GO TO 50
  80 REM ********************
  90 REM initialise
 100 CLS
 110 LET cells=0
 120 DIM a(11,11): DIM b(11,11)
 130 FOR x=2 TO 10: FOR y=2 TO 1
0
 135 BORDER RND*7
 140 IF RND>.35 THEN LET a(x,y)=
1: BEEP .02,x+y/2: LET cells=cel
ls+1
 150 LET b(x,y)=a(x,y)
 160 NEXT y: NEXT x
 170 LET year=0
 175 BORDER 7: CLS
 180 RETURN
 190 REM ******************
 200 REM Print colony
 210 LET year=year+1
```



```
 220 BEEP .02,RND*20
 250 PRINT AT 1,0; INK RND*6;"Ga
neration ";year;AT 3,8;
 255 LET cells=0
 260 FOR x=2 TO 10: FOR y=2 TO 1
0
 270 LET a(x,y)=b(x,y)
 280 IF a(x,y)=0 THEN PRINT "  "
;
 285 IF a(x,y)=1 THEN PRINT INK
RND*6;"■ ";: LET cells=cells+1
 290 NEXT y: PRINT : PRINT : PRI
NT TAB 8;: NEXT x
 292 PRINT AT 21,0; INK RND*6;"C
ells ";cells;"
 293 IF cells<6 THEN RUN
 295 RETURN
 300 REM ******************
 310 REM Update
 340 FOR x=2 TO 10: FOR y=2 TO 1
0
 345 BORDER RND*6
 350 LET c=0
 360 IF a(x-1,y-1)=1 THEN LET c=
c+1
 370 IF a(x-1,y)=1 THEN LET c=c+
1
 380 IF a(x-1,y+1)=1 THEN LET c=
c+1
 390 IF a(x,y-1)=1 THEN LET c=c+
1
 400 IF a(x,y+1)=1 THEN LET c=c+
1
 410 IF a(x+1,y-1)=1 THEN LET c=
c+1
 420 IF a(x+1,y)=1 THEN LET c=c+
1
 430 IF a(x+1,y+1)=1 THEN LET c=
c+1
 440 IF a(x,y)=1 AND c<>2 AND c<
>3 THEN LET b(x,y)=0
 450 IF a(x,y)=0 AND c=3 THEN LE
T b(x,y)=1
 460 NEXT y: NEXT x
 470 RETURN
```

Matchsticks

This game is based on one which was played in the film "Last Year at Marienbad". There are a certain number of 'matches' at the start of the game, and you and the computer take it in turns to take one or more away. The maximum number you can take is shown at the top of the screen. The player who takes the last match loses. The computer is not infallible.

```
  5 REM * MATCHSTICKS *
 10 REM WHITE TEXT ON BLUE
 15 PAPER 1: INK 7: BORDER 1: C
LS
 20 LET E=0: LET Z=16+INT (RND*
9)
 30 IF 2*(Z/2)=Z THEN LET Z=Z+1
 40 LET H=INT (RND*4)+2
 50 PRINT PAPER RND*5+2; INK 0;
AT 0,6;"MAXIMUM TO TAKE IS ";H
 60 IF E>0 THEN PRINT AT 7,2;"Y
OU TOOK ";E;TAB 20;"I TOOK ";0
 70 FOR K=1 TO 9: BEEP .01,K
 80 PRINT INK RND*5+2;K;" ";
 90 IF RND>.85 THEN PRINT : PRI
NT
100 NEXT K
105 LET K=7: IF RND>.5 THEN LET
K=4
110 INPUT INK K;"HOW MANY WILL
YOU TAKE? ";E
120 IF E>H OR E<1 THEN GO TO 11
0
130 CLS : LET Z=Z-E
140 IF Z=0 THEN BORDER RND*7: P
RINT PAPER RND*5;AT 10,12;"I WIN
": BEEP .05,RND*30+30: GO TO 140
150 LET O=Z-1-INT ((Z-1)/(H+1))
*(H+1)+INT (RND*3)-1
160 IF O>Z OR O<1 OR O>H THEN G
O TO 150
170 LET Z=Z-O
180 IF Z=0 THEN BORDER RND*7: P
RINT ,,,PAPER RND*6;AT 10,2;"I TO
OK ";O;" SO YOU WIN!": BEEP .05
,RND*10: GO TO 100
190 GO TO 50
```



```
      MAXIMUM TO TAKE IS 5


  YOU TOOK 3          I TOOK 1
1  2
3
4  5  6  7  8
9
10  11
12  13  14
```

Fruit machine

The next program costs you an inflationary $1.50 a spin. From time to time the HOLD option will come up. You can hold all four reels if you like. When HOLD comes up, you just enter each number you wish to hold, pressing ENTER after each one. When you have held enough, or if you don't want to hold any, enter 5, then press ENTER which gets you back to the next roll.

```
 20 POKE 23609,100
 30 GO SUB 9000
 40 POKE 23692,-1
 50 PAPER 0: CLS : BORDER 0: IN
K 7
 60 PRINT '''' PAPER 2;TAB 2;"TH
IS IS ROUND ";ROUND';TAB 2;"YOU
HAVE $";MONEY';TAB 2;"PRESS ANY
KEY TO ROLL"
 70 IF INKEY$<>"" THEN GO TO 70
 80 IF INKEY$="" THEN GO TO 80
 85 POKE 23692,-1
 90 FOR G=1 TO 50: BORDER RND*7
: BEEP .01,50-G: NEXT G: BORDER
0
100 FOR J=1 TO 4
```

```
110 IF M(J)=J THEN GO TO 150
120 LET A(J)=INT (RND*4)+1
140 BEEP .1,50/J
150 NEXT J
155 LET ROUND=ROUND+1
160 GO SUB 5000
165 GO SUB 4000
170 IF RND>.7 THEN GO SUB 6000
175 FOR T=1 TO 45: PRINT AT 1,2
5; INK RND*7;"■ ■ ■";AT 1,25; INK
RND*7;"■ ■ ■";: NEXT T
177 FOR T=1 TO 25: PRINT : NEXT
T
180 IF MONEY>0 THEN GO TO 60
190 PRINT '''TAB 5;"YOU SURVIVE
D ";ROUND;" ROUNDS"
195 BORDER RND*7
200 PRINT "BUT NOW YOU ARE BROK
E AND THE"
205 BORDER RND*7
210 PRINT "C A S I N O I S C
L O S E D !"
215 BORDER RND*7
220 POKE 23692,-1
230 PAUSE 10
240 GO TO 130
4000 REM ** MONEY **
4005 PRINT '''': POKE 23692,-1
4010 LET MONEY=MONEY-1.5
4020 IF A(1)=A(2) AND A(2)=A(3)
AND A(3)=A(4) THEN PRINT INK 5;"
£££££ JACKPOT!!!! £££££££££";
BEEP 2,10: PRINT '''YOU WIN $10
";: LET MONEY=MONEY+10: GO TO 4
100
4030 IF (A(1)=A(2) AND (A(3)=A(2
) OR A(4)=A(2))) OR A(1)=A(2) AN
D A(2)=A(4) OR A(2)=A(3) AND A(3
)=A(4) THEN PRINT INK 6; PAPER 2
;"$$$$$$$ THREE OF A KIND! $$$$
$";: BEEP 2,20: PRINT '''YOU WIN
$5";: LET MONEY=MONEY+5: GO TO 41
00
4040 IF A(3)=A(2) AND A(3)=A(4)
THEN PRINT INK 6; PAPER 2;"$$$$$
$$$ TRIO $$ TRIO $$ $$$$$";: BEEP
.2,40: PRINT '''YOU WIN $7.50";
LET MONEY=MONEY+7.5: GO TO 4100
4050 IF A(1)=A(2)+A(3)+A(4)=10 T
HEN PRINT PAPER 2;"››››››››››";PA
PER 0;" SMASHERO!!";: BEEP 2,-30
: PRINT '''YOU WIN $7.50!!";: LET
 MONEY=MONEY+7.5
4100 FOR T=1 TO 20: BORDER RND*7
: BEEP .01,T: NEXT T: BORDER 0
4105 PRINT
4110 FOR T=1 TO 64: PRINT INK RN
D*7;"■";: NEXT T
```



```
4120 PRINT '''TAB 8;"YOU NOW HAVE
 $";MONEY''
4130 FOR T=1 TO 64: PRINT INK RN
D*7;"■";: NEXT T
4140 PRINT
4150 POKE 23692,-1: PRINT : PRIN
T
4160 DIM M(4)
4170 RETURN
5000 REM ** SPIN **
5010 FOR T=1 TO 50: BORDER RND*7
: BEEP .01,50/T/2: NEXT T: BORDE
R 0
5020 PRINT '''TAB 4;
5030 FOR J=1 TO 4
5040 IF A(J)=1 THEN PRINT INK 2;
"■ ■";: BEEP .1,10
5050 IF A(J)=2 THEN PRINT INK 7;
"■$$■";: BEEP .1,20
5060 IF A(J)=3 THEN PRINT INK 4;
"■$$■";: BEEP .1,30
5070 IF A(J)=4 THEN PRINT INK 5;
"■■■";: BEEP .1,40
5080 PAUSE 70
5090 NEXT J
5100 RETURN
6000 REM ** HOLD **
6010 DIM M(5)
6020 BEEP .5,1
6025 POKE 23692,-1
6030 PRINT '': INK 6;"ENTER ANY N
UMBER(S) YOU"
6040 PRINT ' INK 6;"WISH TO HOLD
 ENTER'S."
6050 PRINT ' INK 6;"WHEN YOU HAV
E FINISHED"
6060 INPUT 0
6070 IF 0<>5 THEN PRINT INK 2;0
6080 LET M(0)=0
6090 IF 0<>5 THEN GO TO 6060
6100 RETURN
9000 REM ** ASSIGN VARIABLES **
9010 DIM A(5): DIM M(5)
9020 LET MONEY=7.5
9030 LET ROUND=1
9040 FOR T=1 TO 20
9050 BEEP .2,2*T
9060 NEXT T
9070 BORDER 7: PAPER 7: CLS
9080 BORDER 0: PAPER 0: CLS
9090 RETURN
```

Final circuit

FINAL CIRCUIT was adapted from a ZX80 program (2K RACETRACK) first published in the National ZX Users' CLUB monthly magazine, INTERFACE. The original version was written by Alan Gunnell.

It is easy to play, and because it ends up giving you a score after each 'race', acts as a challenge to play it over and over again, trying to increase your score. There are three 'racetracks' on which you can drive, of varying degrees of difficulty.

Throughout the race, you are asked to enter your choice of acceleration and gear setting. You'll soon learn the effects these have. Your score is shown at all times (line 220), and a final score is given at the end. Your feedback (including such lines as 'Driver behind is hooting, hurry up' if you're dragging your heels) is in words, and comes throughout the race. You'll find there is a great tendency to crash, and your vehicle manages somehow to survive an infinite number of crashes. Of particular interest is line 290, which takes the place of five IF/THEN statements of the type IF H = 5 THEN LET B$ = "oily straight" and so on.

```
  5 REM Final Circuit
 10 REM Adapted from ZX80
 12 REM Program by Alan Gunnell
 14 REM First published in
 16 REM      INTERFACE
 18 REM
 20 LET a=5: LET g=1: LET b=3
 22 BORDER 1: PAPER 7: INK 0
 25 INPUT "Which track (3 TO 5)
?";v
 27 IF v<3 OR v>5 THEN GO TO 25
 30 LET x=0
 40 LET l=100+v*v
 50 LET s=0
 60 IF x=10 THEN STOP
 80 LET x=x+1
 90 IF x=10 THEN PRINT INK RND*
6;TAB 5;"THE RACE IS OVER";TAB 4
;"Score is ";l;" out of ";100+v*
v: POKE 23692,-1: BORDER RND*7:
BEEP .02,RND*30: GO TO 90
```



```
110 GO SUB 180
112 FOR t=1 TO 50: BEEP .02,t:
NEXT t
115 GO SUB 270
120 PRINT INK RND*6;b$
125 GO SUB 145
130 GO SUB 350
135 PAUSE 50
140 GO SUB 270
142 GO TO 60
145 FOR t=1 TO 50: BORDER RND*7
: NEXT t: BORDER 1
149 LET s=ABS (s+(a*a)-(b*15)+(
2*g))
150 PRINT '''' PAPER 2; INK 6;"
Gear ";g;"  Speed ";s
160 GO SUB h*1000
180 BORDER 1: INPUT INK 7;"Sele
ct a gear (1 TO 10)";g
190 IF g<1 OR g>10 THEN GO TO 1
80
200 INPUT INK 7;"Enter accelera
tion (0 TO 10)";a
210 IF a<1 OR a>10 THEN GO TO 2
00
220 PRINT INK RND*6;"Current sc
ore is ";l: INK 9;'
240 INPUT PAPER 2; INK 6;"Enter
 braking (0 TO 10)";b
250 IF b<0 OR b>10 THEN GO TO 2
40
260 RETURN
270 LET h=INT (RND*v)+1
290 LET b$="oily straight" AND
h=5)+("hairpin" AND h=4)+("corn
er" AND h=3)+("bend" AND h=2)+("
straight" AND h=1)
310 RETURN
350 IF a=0 THEN LET a=1
360 LET s=ABS (s+(a*a)-(b*15)+(
2*g))
370 IF s<10 THEN LET s=10
380 IF s<15 THEN PRINT '''' IN
K 2;"Driver behind is hooting":
PRINT ' INK 1;"Hurry up"
400 RETURN
1000 IF s>90 THEN PRINT INK 2;"Y
ou're speeding...slow down!": LE
T l=l-5
1010 RETURN
2000 IF s>40 THEN BEEP 3,50: FOR
 t=1 TO 20: BORDER RND*6: NEXT t
: PRINT INK 2;"Crash
2010 IF b>5 THEN PRINT INK 2;"Cr
ash": BEEP .5,20: BORDER RND*6
PAUSE 20: LET l=l-9+INT (RND*10)
3020 RETURN
3000 IF s>25 THEN FOR r=1 TO 10:
 PRINT INK RND*6;"Crash!!!!!!!!!
": NEXT r: LET l=l-10
```

```
3010 RETURN
4000 IF s>35 THEN PRINT INK 2;"*
*****Crash*********************":
 LET l=l-10
4010 RETURN
5000 IF s>20 THEN PRINT INK 2;"C
CCCCrrrrraaaassssshhh!!!!!": LE
T l=l-10
5005 FOR t=1 TO 50: PAPER RND*7:
 CLS : NEXT t: PAPER 7
5010 IF b>3 THEN PRINT "Crash!!"
: LET l=l-10
5020 RETURN
```

Breakout

In this game, based on one written by Eric Thompson, you control the action of the little slide at the bottom with the "1" and "Ø" keys. Your ball is a small letter "o".

```
  3 POKE 23609,100
  5 REM BREAKOUT
  6 REM BASED ON ZX81 PROGRAM
  7 REM BY ERIC THOMPSON
  9 GO SUB 500
 15 PAPER 6: CLS : BORDER 2
 20 FOR L=0 TO 7
 30 PRINT AT L,8; INK RND*3;"▚▚
▚▚▚▚▚▚▚▚▚▚▚▚"
 40 NEXT L
 55 LET S=0
 60 LET X=16
 70 LET B=INT (RND*10)+8
 80 LET D=1-INT (RND*3)
 90 IF D=0 THEN GO TO 80
100 FOR P=8 TO -8 STEP -1
110 IF ABS B>18 THEN LET B=18
115 LET X=X+(INKEY$="0")-(INKEY
$="1")
120 PRINT AT 9,X-2; INK 2;" ▛
 ";AT P,B; INK 1;"o"
140 LET B1=B
150 IF B=8 OR B=18 THEN LET D=-
D: BEEP .005,30
160 LET B=B+D
170 LET X=X+(INKEY$="0")-(INKEY
$="1")
175 FOR G=1 TO Z: NEXT G
180 PRINT AT P,B1;" "
190 NEXT P
```



```
195 LET X=X+(INKEY$="0")-(INKEY
$="1")
200 IF ABS (B-X)>2 THEN GO TO 2
30
205 LET S=S+1
210 PRINT AT 16,0; INK 1;"Your
score is "; INK 2;S: BEEP .25,1
220 GO TO 80
230 FOR G=1 TO 400: NEXT G
240 LET S=0
250 GO TO 15
500 INPUT "DEGREE OF DIFFICULTY
 (1 TO 20)?";Z
510 IF Z<1 OR Z>20 THEN GO TO 5
00
520 LET Z=2*Z
530 FOR S=1 TO Z
540 BEEP .01,S
550 NEXT S
560 RETURN
```

Galxian

This program was adapted by Tim Hartnell from a ZX81 program written by James Walsh and Paul Holmes and was first published in the magazine DATABUS.

```
  1 REM      Galxian -
    Holmes, Walsh, Hartnell
  4 GO SUB 1000
  5 BORDER 0
  6 PAPER 0: CLS
  7 INK 6
 10 LET X=10
 20 LET S=0
 30 LET P=1
 40 LET S=S+P
 50 CLS
 60 PRINT AT 0,0;S
 70 LET P=0
 80 LET N=10
 90 LET Y=13
100 LET C=0
110 PRINT AT 15,X;"   "
120 IF INKEY$="5" THEN LET X=X-
1
130 IF INKEY$="8" THEN LET X=X+
1
140 PRINT AT 15,X; INK 2;A$
145 BEEP .004,10
```

```
150 PRINT AT P,N;" "
160 IF RND>.65 THEN LET N=N+INT
(RND*3-1)
170 IF RND>.6 THEN LET P=P+1
190 PRINT AT P,N; INK 4;"B"
200 IF P>14 THEN GO TO 5
210 IF C=0 AND INKEY$="1" THEN
LET C=X+1
220 IF C=0 THEN GO TO 110
230 PRINT AT Y,C;" "
240 LET Y=Y-1
250 IF Y<4 THEN GO TO 90
260 PRINT AT Y,C;"A": BEEP .009
,50
270 IF C=N AND Y=P THEN FOR U=1
TO 9: PRINT AT Y,C; INK RND*5;"
": BEEP .15,15: BORDER RND*7: B
EEP .15,20: BORDER RND*7: BEEP .
15,30: NEXT U: GO TO 40
280 GO TO 110
1000 LET A$="◆◆◆"
1005 FOR J=0 TO 7
1010 READ N
1020 POKE USR "A"+J,N
1030 NEXT J
1040 FOR J=0 TO 7
1050 READ N
1060 POKE USR "B"+J,N
1070 NEXT J
1080 FOR J=0 TO 7
1090 READ N
1100 POKE USR "C"+J,N
1110 NEXT J
1120 FOR J=0 TO 7
1130 READ N
1140 POKE USR "M"+J,N
1150 NEXT J
1160 FOR J=0 TO 7
1170 READ N
1180 POKE USR "E"+J,N
1190 NEXT J
2000 DATA BIN 11111110,BIN 11111
100,BIN 11111000,BIN 11110001,BI
N 01100001,BIN 10010001,BIN 1011
1000,BIN 10111100
2010 DATA BIN 01111110,BIN 10111
101,BIN 11000011,BIN 01000010,BI
N 00000000,BIN 01000010,BIN 0011
1100,BIN 00111100
2020 DATA BIN 01111111,BIN 00111
111,BIN 00011111,BIN 10001111,BI
N 10000110,BIN 10000001,BIN 0001
1101,BIN 00111101
2030 DATA BIN 01111110,BIN 10011
001,BIN 11000011,BIN 01100110,BI
N 01100110,BIN 10000001,BIN 1100
0011,BIN 11100111
```



```
2040 DATA BIN 11110111,BIN 11100
011,BIN 11010101,BIN 10110110,BI
N 11110111,BIN 11110111,BIN 1110
0011,BIN 11001001
5000 RETURN
```

Appendices

Microdrive

This is a miniature microfloppy disc memory system. Each Microdrive holds up to 100K of program or data, and up to eight can be connected to the Spectrum at once. The transfer rate of information from the Microdrive to the Spectrum is 16K per second, and the total time to scan the entire microfloppy to find a particular point is seven seconds although many access times will be less than that. Some commands which have not been mentioned in this book are designed entirely for the Microdrive and the RS232 interface.

RS232 interface

The RS232 interface allows the Spectrum to be connected to any of peripherals (such as printers, terminals, other computers) which are RS232 compatible. The RS232 is an industry standard, so there is a wide range of uses for a Spectrum with the interface. The interface operating system is in the Spectrum monitor.

Other commands

OPEN#, CLOSE#, MOVE, ERASE, CAT and FORMAT are designed for use with the interface and Microdrive. The Microdrive supports not only SAVE, VERIFY, LOAD and MERGE, but also PRINT, LIST, INPUT and INKEY$.

IN and OUT are commands to trigger the computer's input and output ports, and are used for controlling or obtaining information from such things as the keyboard or printer.

Binary to decimal converter

You may prefer to use decimal, rather than binary, numbers in your data statements for user-defined graphics. If this is so, this list should help you; in case you're interested, the program used to print out the list is given at the end.

00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
00000101	5
00000110	6
00000111	7
00001000	8
00001001	9
00001010	10
00001011	11
00001100	12
00001101	13
00001110	14
00001111	15
00010000	16
00010001	17
00010010	18
00010011	19
00010100	20
00010101	21
00010110	22
00010111	23
00011000	24
00011001	25
00011010	26
00011011	27
00011100	28
00011101	29
00011110	30
00011111	31
00100000	32
00100001	33
00100010	34
00100011	35
00100100	36
00100101	37
00100110	38
00100111	39
00101000	40
00101001	41
00101010	42
00101011	43
00101100	44
00101101	45

00101110	46
00101111	47
00110000	48
00110001	49
00110010	50
00110011	51
00110100	52
00110101	53
00110110	54
00110111	55
00111000	56
00111001	57
00111010	58
00111011	59
00111100	60
00111101	61
00111110	62
00111111	63
01000000	64
01000001	65
01000010	66
01000011	67
01000100	68
01000101	69
01000110	70
01000111	71
01001000	72
01001001	73
01001010	74
01001011	75
01001100	76
01001101	77
01001110	78
01001111	79
01010000	80
01010001	81
01010010	82
01010011	83
01010100	84
01010101	85
01010110	86
01010111	87
01011000	88
01011001	89
01011010	90
01011011	91
01011100	92
01011101	93
01011110	94
01011111	95
01100000	96
01100001	97
01100010	98
01100011	99
01100100	100
01100101	101
01100110	102
01100111	103

01101000	104
01101001	105
01101010	106
01101011	107
01101100	108
01101101	109
01101110	110
01101111	111
01110000	112
01110001	113
01110010	114
01110011	115
01110100	116
01110101	117
01110110	118
01110111	119
01111000	120
01111001	121
01111010	122
01111011	123
01111100	124
01111101	125
01111110	126
01111111	127
10000000	128
10000001	129
10000010	130
10000011	131
10000100	132
10000101	133
10000110	134
10000111	135
10001000	136
10001001	137
10001010	138
10001011	139
10001100	140
10001101	141
10001110	142
10001111	143
10010000	144
10010001	145
10010010	146
10010011	147
10010100	148
10010101	149
10010110	150
10010111	151
10011000	152
10011001	153
10011010	154
10011011	155
10011100	156
10011101	157
10011110	158
10011111	159
10100000	160
10100001	161

10100010	162
10100011	163
10100100	164
10100101	165
10100110	166
10100111	167
10101000	168
10101001	169
10101010	170
10101011	171
10101100	172
10101101	173
10101110	174
10101111	175
10110000	176
10110001	177
10110010	178
10110011	179
10110100	180
10110101	181
10110110	182
10110111	183
10111000	184
10111001	185
10111010	186
10111011	187
10111100	188
10111101	189
10111110	190
10111111	191
11000000	192
11000001	193
11000010	194
11000011	195
11000100	196
11000101	197
11000110	198
11000111	199
11001000	200
11001001	201
11001010	202
11001011	203
11001100	204
11001101	205
11001110	206
11001111	207
11010000	208
11010001	209
11010010	210
11010011	211
11010100	212
11010101	213
11010110	214
11010111	215
11011000	216
11011001	217
11011010	218
11011011	219

```
11011100        220
11011101        221
11011110        222
11011111        223
11100000        224
11100001        225
11100010        226
11100011        227
11100100        228
11100101        229
11100110        230
11100111        231
11101000        232
11101001        233
11101010        234
11101011        235
11101100        236
11101101        237
11101110        238
11101111        239
11110000        240
11110001        241
11110010        242
11110011        243
11110100        244
11110101        245
11110110        246
11110111        247
11111000        248
11111001        249
11111010        250
11111011        251
11111100        252
11111101        253
11111110        254
11111111        255
```

```
 10 FOR B=0 TO 255
 20 LET J=B
 30 LET A$=""
 40 FOR N=0 TO 7
 50 LET T=J-INT (J/2)*2
 60 IF T=0 THEN LET A$="0"+A$
 70 IF T<>0 THEN LET A$="1"+A$
 80 LET J=INT (J/2)
 90 NEXT N
100 PRINT A$,B
110 NEXT B
```


Report codes

These appear at the bottom of the screen whenever the computer stops executing some BASIC, and explain why it stopped, whether for a natural reason, or because an error occurred.

CONTINUE generally goes to the line and statement specified in the last report, but there are exceptions with reports **0**, 9 and D.

Code Meaning

0 OK
Successful completion, or jump to a line number bigger than any existing.

1 NEXT without FOR

2 Variable not found

3 Subscript wrong
A subscript is beyond the dimension of the array, or there are the wrong number of subscripts.

4 Out of memory

5 Out of screen
An INPUT statement has tried to generate more than 23 lines in the lower half of the screen. Also occurs with PRINT AT 22,...

6 Number: too big
Calculations have led to a number greater than about 10^{38}.

7 RETURN without GO SUB

8 End of file (Microdrive, etc, operations)

9 STOP statement

A Invalid argument

B Integer out of range

C Nonsense in BASIC

D BREAK — CONT repeats.
The behaviour of CONTINUE after this report is normal in that it repeats the statement.

E Out of DATA

F Invalid file name

G No room for line

H STOP in INPUT

I FOR without NEXT

J Invalid I/O device. (Microdrive, etc, operations)

K Invalid colour

L BREAK into program
BREAK pressed, this is detected between two statements.

M RAMTOP no good.
The number specified for RAMTOP is either too big or too small.

N Statement lost.
Jump to a statement that no longer exists.

O Invalid stream. (Microdrive, etc, operations.)

P FN without DEF

Q Parameter error,
Wrong number of arguments, or one of them is the wrong type (string instead of number or vice versa).

R Tape loading error

CONTENTS:

Your ZX Spectrum is a powerful computer and this book will help you make the most of it. From first principles, right through to quite complex programming techniques, this book leads you step by simple step through the art of programming your new computer.

The book contains more than 100 programs and routines, all guaranteed to run, designed to get your computer up and running with interesting and worthwhile programs from the moment you switch it on.

The book is by Tim Hartnell, one of the UK's leading experts on small computer systems, and best-selling author of a number of ZX books, including, 'Getting Acquainted with your ZX81', '49 Explosive Games for the ZX81' and 'Making the Most of Your ZX80'. He is also author of the authorative 'Personal Computer Guide', published by Virgin Books; 'The Book of Listings' (co-author Jeremy Ruston), published by the BBC; and 'Let Your BBC Micro Teach You to Program', published by Interface. He is a regular contributor to the monthly computer magazines, and answers readers' queries each month in Your Computer. Tim is editor of the bi-monthly magazine 'ZX Computing', Britain's biggest magazine for the Sinclair user.

In this book, Tim is joined by Dilwyn Jones, an experienced computer programmer of many years standing. Dilwyn has made a special study of ways of getting the maximum use out of ZX computers, and shares his findings with you in this book.

Interface books are designed to make the art of computer programming simple, and inviting, and this book follows strongly in this tradition.

The computer press have welcomed previous books:

"If, in any sense you are a beginner to programming or computers, this is undoubtedly the book to read. Full of insight, witty, sensible and extremely funny, it eases you into programming practically from the word go..." Personal Computer World.

"Tim Hartnell has certainly provided the reader with many varied programs but in the text, linked to most of the listings, is a well thought out 'hands on' approach... As you work your way through the book, not only does your library of programs grow but also your understanding of the BASIC commands which make them possible..." Computing Today.

Another great book from

INTERFACE PUBLICATIONS

ISBN 0-907563-19-8

PROGRAMMING YOUR **ZX SPECTRUM** by **TIM HARTNELL** and DILWYN JONES

IP