CHAPTER 17

Graphics

Summary
PLOT, DRAW, CIRCLE
POINT
pixels

In this chapter we shall see how to draw pictures on the ZX Spectrum. The part of the screen you can use has 22 lines and 32 columns, making 22*32=704 character positions. As you may remember from Chapter 16, each of these character positions is made of an 8 by 8 square of dots, and these are called pixels (picture elements).

A pixel is specified by two numbers, its *coordinates*. The first, its *x coordinate*, says how far it is across from the extreme left-hand column. (Remember, x is a cross), the second, its *y coordinate*, says how far it is up from the bottom (wise up). These coordinates are usually written as a pair in brackets, so (0,0), (255,0), (0,175) and (255,175) are the bottom left-, bottom right-, top left- and top right-corners.

The statement

PLOT x coordinate, y coordinate

inks in the pixel with these coordinates, so this measles program

```
10 PLOT INT (RND*256), INT (RND*176): INPUT a$: GO TO 10
```

plots a random point each time you press **ENTER**.

Here is a rather more interesting program. It plots a graph of the function **SIN** (a sine wave) for values between 0 and 2π.

```
10 FOR n=0 TO 255
20 PLOT n,88+80*SIN (n/128*PI)
30 NEXT n
```

This next program plots a graph of **SQR** (part of a parabola) between 0 and 4:

```
10 FOR n=0 TO 255
20 PLOT n,80*SQR (n/64)
30 NEXT n
```

Notice that pixel coordinates are rather different from the line and column in an **AT** item. You may find the diagram in Chapter 15 useful when working out pixel coordinates and line and column numbers.

To help you with your pictures, the computer will draw straight lines, circles and parts of circles for you, using the **DRAW** and **CIRCLE** statements.

The statement **DRAW** to draw a straight line takes the form

DRAW x,y

The starting place of the line is the pixel where the last **PLOT**, **DRAW** or **CIRCLE** statement left off (this is called the **PLOT** position; **RUN**, **CLEAR**, **CLS** and **NEW** reset it to the bottom left hand corner, at (0,0)), and the finishing place is **x** pixels to the right of that and **y** pixels up. The **DRAW** statement on its own determines the length and direction of the line, but not its starting point.

Experiment with a few **PLOT** and **DRAW** commands, for instance

```
PLOT 0,100: DRAW 80,-35
PLOT 90,150: DRAW 80,-35
```

Notice that the numbers in a **DRAW** statement can be negative, although those in a **PLOT** statement can't.

You can also plot and draw in colour, although you have to bear in mind that colours always cover the whole of a character position and cannot be specified for individual pixels. When a pixel is plotted, it is set to show the full ink colour, and the whole of the character position containing it is given the current ink colour. This program demonstrates this:

```
10 BORDER 0: PAPER 0: INK 7: CLS : REM black out screen
20 LET x1=0: LET y1=0: REM start of line
30 LET c=1: REM for ink colour, starting blue
40 LET x2=INT (RND*256): LET y2=INT (RND*176): REM
   random finish of line
50 DRAW INK c;x2-x1,y2-y1
60 LET x1=x2: LET y1=y2: REM next line starts where last one
   finished
70 LET c=c+1: IF c=8 THEN LET c=1: REM new colour
80 GO TO 40
```

The lines seem to get broader as the program goes on, and this is because a line changes the colours of all the inked in pixels of all the character positions that it passes through. Note that you can embed **PAPER**, **INK**, **FLASH**, **BRIGHT**, **INVERSE** and **OVER** items in a **PLOT** or **DRAW** statement just as you could with **PRINT** and **INPUT**. They go between the key word and the coordinates, and are terminated by either semicolons or commas.

An extra frill with **DRAW** is that you can use it to draw parts of circles instead of straight lines, by using an extra number to specify an angle to be turned through: the form is

```
DRAW x,y,a
```

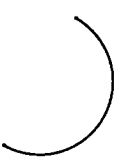
x and **y** are used to specify the finishing point of the line just as before and **a** is the number of radians that it must turn through as it goes – if **a** is a positive it turns to the left, while if **a** is a negative it turns to the right. Another way of seeing **a** is as showing the fraction of a complete circle that will be drawn: a complete circle is 2π radians, so

if **a**=π it will draw a semicircle, if **a**=0.5*π a quarter of a circle, and so on.

For instance suppose **a**=π. Then whatever values **x** and **y** take, a semicircle will be drawn. Run

```
10 PLOT 100,100: DRAW 50,50, PI
```

which will draw this:



The drawing starts off in a south-easterly direction, but by the time it stops it is going north-west: in between it has turned round through 180 degrees, or π radians (the value of **a**).

Run the program several times, with **PI** replaced by various other expressions – e.g. **−PI**, **PI/2**, **3*PI/2**, **PI/4**, **1,0**.

The last statement in this chapter is the **CIRCLE** statement, which draws an entire circle. You specify the coordinates of the centre and the radius of the circle using

CIRCLE x coordinate, y coordinate, radius

Just as with **PLOT** and **DRAW**, you can put the various sorts of colour items in at the beginning of a **CIRCLE** statement.

The **POINT** function tells you whether a pixel is ink or paper colour. It has two arguments, the coordinates of the pixel (and they must be enclosed in brackets); and its result is 0 if the pixel is paper colour, 1 if it is ink colour. Try

```
CLS : PRINT POINT (0,0): PLOT 0,0: PRINT POINT (0,0)
```

Type

```
PAPER 7: INK 0
```

and let us investigate how **INVERSE** and **OVER** work inside a **PLOT** statement. These two affect just the relevant pixel, and not the rest of the character positions. They are normally off (0) in a **PLOT** statement, so you only need to mention them to turn them on (1).

Here is a list of the possibilities for reference:

PLOT; – this is the usual form. It plots an ink dot, i.e. sets the pixel to show the ink colour.

PLOT INVERSE 1; – this plots a dot of ink eradicator, i.e. it sets the pixel to show the paper colour.

PLOT OVER 1; – this changes the pixel over from whatever it was before: so if it was ink colour it becomes paper colour, and vice versa.

PLOT INVERSE 1; **OVER** 1; – this leaves the pixel exactly as it was before; but note that it also changes the **PLOT** position, so you might use it simply to do that.

As another example of using the **OVER** statement fill the screen up with writing using black on white, and then type

```
PLOT 0,0: DRAW OVER 1;255,175
```

This will draw a fairly decent line, even though it has gaps in it wherever it hits some writing. Now do exactly the same command again. The line will vanish without leaving any traces whatsoever. This is the great advantage of **OVER 1**. If you had drawn the line using

```
PLOT 0,0: DRAW 255,175
```

and erased it using

```
PLOT 0,0: DRAW INVERSE 1;255,175
```

then you would also have erased some of the writing.

Now try

```
PLOT 0,0: DRAW OVER 1;250,175
```

and try to undraw it by

```
DRAW OVER 1;-250,-175
```

This doesn't quite work, because the pixels the line uses on the way back are not quite the same as the ones that it used on the way down. You must undraw a line in exactly the same direction as you drew it.

One way to get unusual colours is to speckle two normal ones together in a single square, using a user-defined graphic. Run this program:

```
1000 FOR n=0 TO 6 STEP 2
1010 POKE USR "a"+n, BIN 01010101: POKE USR "a"+n+1,
     BIN 10101010
1020 NEXT n
```

which gives the user-defined graphic corresponding to a chessboard pattern. If you print this character (graphics mode, then **a**) in red ink on yellow paper, you will find it

gives a reasonably acceptable orange.

Exercises

1. Play about with **PAPER**, **INK**, **FLASH** and **BRIGHT** items in a **PLOT** statement. These are the parts that affect the whole of the character position containing the pixel. Normally it is as though the **PLOT** statement had started off

```
PLOT PAPER 8; FLASH 8; BRIGHT 8; . . .
```

and only the ink colour of a character position is altered when something is plotted there, but you can change this if you want.

Be especially careful when using colours with **INVERSE 1**, because this sets the pixel to show the paper colour, but changes the ink colour and this might not be what you expect.

2. Try to draw circles using **SIN** and **COS** (if you have read Chapter 10, try to work out how). Run this:

```
10 FOR n=0 TO 2*PI STEP PI /180
20 PLOT 100+80*COS n,87+80*SIN n
30 NEXT n
40 CIRCLE 150,87,80
```

You can see that the **CIRCLE** statement is much quicker, even if less accurate.

3. Try

```
CIRCLE 100,87,80: DRAW 50,50
```

You can see from this that the **CIRCLE** statement leaves the **PLOT** position at a rather indeterminate place – it is always somewhere about half way up the right hand side of the circle. You will usually need to follow the **CIRCLE** statement with a **PLOT** statement before you do any more drawing.

4. Here is a program to draw the graph of almost any function. It first asks you for a number **n**; it will plot the graph for values from **−n** to **+n**. It then asks you for the function itself, input as a string. The string should be an expression using **x** as the argument of the function.

```
10 PLOT 0,87: DRAW 255,0
20 PLOT 127,0: DRAW 0,175
30 INPUT s,e$
35 LET t=0
40 FOR f=0 TO 255
50 LET x=(f-128)*s/128: LET y=VAL e$
60 IF ABS y>87 THEN LET t=0: GO TO 100
70 IF NOT t THEN PLOT f,y+88: LET t=1: GO TO 100
80 DRAW 1,y-old y
100 LET old y=INT (y+.5)
110 NEXT f
```

Run it, and, as an example, type in **10** for the number **n** and **10*TAN x** for the function. It will plot a graph of tan *x* as *x* ranges from −10 to +10.

CHAPTER 18

Motion

Summary
PAUSE, INKEY$, PEEK

Quite often you will want to make the program take a specified length of time, and for this you will find the **PAUSE** statement useful.

PAUSE n

stops computing and displays the picture for *n* frames of the television (at 50 frames per second in Europe or 60 in America). *n* can be up to 65535, which gives you just under 22 minutes; if *n*=0 then it means '**PAUSE** for ever'.

A pause can always be cut short by pressing a key (note that a **CAPS SHIFT**ed space will cause a break as well). You have to press the key down after the pause has started.

This program works the second hand of a clock:

```
10 REM First we draw the clock face
20 FOR n=1 TO 12
30 PRINT AT 10-10*COS (n/6*PI),16+10*SIN (n/6*PI);n
40 NEXT n
50 REM Now we start the clock
60 FOR t=0 TO 200000: REM t is the time in seconds
70 LET a=t/30*PI : REM a is the angle of the second hand in
   radians
80 LET sx=80*SIN a: LET sy=80*COS a
200 PLOT 128,88: DRAW OVER 1;sx,sy: REM draw second hand
210 PAUSE 42
220 PLOT 128,88: DRAW OVER 1;sx,sy: REM erase second hand
400 NEXT t
```

This clock will run down after about 55.5 hours because of line 60, but you can easily make it run longer. Note how the timing is controlled by line 210. You might expect **PAUSE 50** to make it tick one a second, but the computing takes a bit of time as well and has to be allowed for. This is best done by trial and error, timing the computer clock against a real one, and adjusting line 210 until they agree. (You can't do this very accurately; an adjustment of one frame in one second is 2% or half an hour in a day.)

There is a much more accurate way of measuring time. This uses the contents of certain memory locations. The data stored is retrieved by using **PEEK**. Chapter 25 explains what we're looking at in detail. The expression used is

(65536*PEEK 23674+256*PEEK 23673+PEEK 23672)/50

This gives the number of seconds since the computer was turned on (up to about 3 days and 21 hours, when it goes back to 0).

Here is a revised clock program to make use of this:

```
10 REM First we draw the clock face
20 FOR n=1 TO 12
30 PRINT AT 10-10*COS (n/6*PI),16+10*SIN (n/6*PI);n
40 NEXT n
50 DEF FN t()=INT ((65536*PEEK 23674+256*PEEK 23673+
   PEEK 23672)/50): REM number of seconds since start
100 REM Now we start the clock
110 LET t1=FN t()
120 LET a=t1/30*PI : REM a is the angle of the second hand in
   radians
130 LET sx=72*SIN a: LET sy=72*COS a
140 PLOT 131,91: DRAW OVER 1;sx,sy: REM draw hand
200 LET t=FN t()
210 IF t<=t1 THEN GO TO 200: REM wait until time for next
   hand
220 PLOT 131,91: DRAW OVER 1;sx,sy: REM rub out old hand
230 LET t1=t: GO TO 120
```

The internal clock that this method uses should be accurate to about .01% as long as the computer is just running its program, or 10 seconds per day; but it stops temporarily whenever you do **BEEP**, or a cassette tape operation, or use the printer or any of the other extra pieces of equipment you can use with the computer. All these will make it lose time.

The numbers **PEEK 23674**, **PEEK 23673** and **PEEK 23672** are held inside the computer and used for counting in 50ths of a second. Each is between 0 and 255, and they gradually increase through all the numbers from 0 to 255; after 255 they drop straight back to 0.

The one that increases most often is **PEEK 23672**. Every 1/50 second it increases by 1. When it is at 255, the next increase takes it to 0, and at the same time it nudges **PEEK 23673** by up to 1. When (every 256/50 seconds) **PEEK 23673** is nudged from 255 to 0, it in turn nudges **PEEK 23674** up by 1. This should be enough to explain why the expression above works.

Now, consider carefully: suppose our three numbers are 0 (for **PEEK 23674**), 255 (for **PEEK 23673**) and 255 (for **PEEK 23672**). This means that it is about 21 minutes after switch-on – our expression ought to yield

(65536*0+256*255+255)/50=1310.7

But there is a hidden danger. The next time there is a 1/50 second count, the three numbers will change to 1, 0 and 0. Every so often, this will happen when you are half way through evaluating the expression: the computer would evaluate **PEEK 23674** as

0, but then change the other two to 0 before it can peek them. The answer would then be

(65536*0+256*0+0)/50=0

which is hopelessly wrong.

A simple rule to avoid this problem is *evaluate the expression twice in succession and take the larger answer.*

If you look carefully at the program above you can see that it does this implicitly.

Here is a trick to apply the rule. Define functions

```
10 DEF FN m(x,y)=(x+y+ABS (x-y))/2: REM the larger of x and y
20 DEF FN u()=(65536*PEEK 23674+256*PEEK 23673+PEEK
   23672)/50: REM time, may be wrong
30 DEF FN t()=FN m(FN u(), FN u()): REM time, right
```

You can change the three counter numbers so that they give the real time instead of the time since the computer was switched on. For instance, to set the time at 10.00am, you work out that this is 10*60*60*50=1800000 fiftieths of a second, and that

1800000=65536*27+256*119+64

To set the three numbers to 27, 119 and 64, you do

```
POKE 23674,27: POKE 23673,119: POKE 23672,64
```

In countries with mains frequencies of 60 Hertz these programs must replace '50' by '60' where appropriate.

The function **INKEY$** (which has no argument) reads the keyboard. If you are pressing exactly one key (or a **SHIFT** key and just one other key) then the result is the character that that key gives in L mode; otherwise the result is the empty string.

Try this program, which works like a typewriter.

```
10 IF INKEY$ <>"" THEN GO TO 10
20 IF INKEY$ ="" THEN GO TO 20
30 PRINT INKEY$;
40 GO TO 10
```

Here line 10 waits for you to lift your finger off the keyboard and line 20 waits for you to press a new key.

Remember that unlike **INPUT**, **INKEY$** doesn't wait for you. So you don't type **ENTER**, but on the other hand if you don't type anything at all then you've missed your chance.

Exercises

1. What happens if you miss out line 10 in the typewriter program?

2. Another way of using **INKEY$** is in conjunction with **PAUSE**, as in this alternative typewriter program.

```
10 PAUSE 0
20 PRINT INKEY$;
30 GO TO 10
```

To make this work, why is it essential that a pause should not finish if it finds you already pressing a key when it starts?

3. Adapt the second hand program so that it also shows minute and hour hands, drawing them every minute. If you're feeling ambitious, arrange so that every quarter of an hour it puts on some kind of show – you could produce the Big Ben chimes with **BEEP**. (See next chapter.)

4. (For sadists.) Try this:

```
10 IF INKEY$ ="" THEN GO TO 10
20 PRINT AT 11,14;"OUCH!"
30 IF INKEY$ <>"" THEN GO TO 30
40 PRINT AT 11,14;"     "
50 GO TO 10
```

CHAPTER 19

BEEP

Summary

BEEP

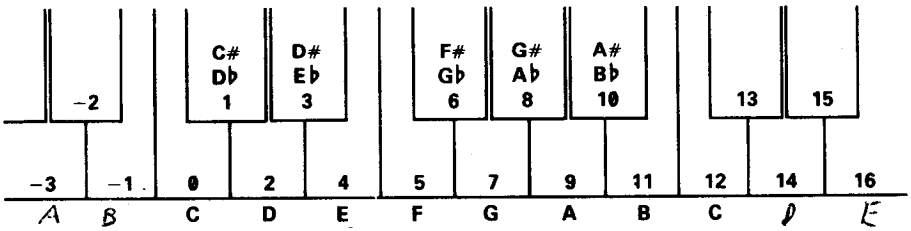
If you haven't already discovered that the ZX Spectrum has a loudspeaker built into it, read the Introductory booklet before carrying on.

The loudspeaker is sounded by using the **BEEP** statement,

BEEP duration, pitch

where, as usual, 'duration' and 'pitch' represent any numerical expressions. The duration is given in seconds, and the pitch is given in semitones above middle C – using negative numbers for notes below middle C.

Here is a diagram to show the pitch values of all the notes in one octave on the piano:



To get higher or lower notes, you have to add or subtract 12 for each octave that you go up or down.

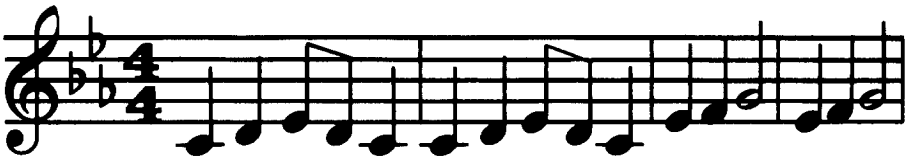
If you have a piano in front of you when you are programming a tune, this diagram will probably be all that you need to work out the pitch values. If, however, you are transcribing straight from some written music, then we suggest that you draw a diagram of the stave with the pitch value written against each line and space, taking the key into account.

For example, type:

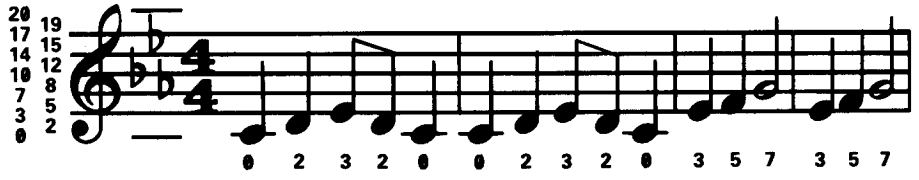
```
10 PRINT "Frere Gustav"
20 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP .5,2: BEEP 1,0
30 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP .5,2: BEEP 1,0
40 BEEP 1,3: BEEP 1,5: BEEP 2,7
50 BEEP 1,3: BEEP 1,5: BEEP 2,7
60 BEEP .75,7: BEEP .25,8: BEEP .5,7: BEEP .5,5: BEEP .5,3:
   BEEP .5,2: BEEP 1,0
70 BEEP .75,7: BEEP .25,8: BEEP .5,7: BEEP .5,5: BEEP .5,3:
   BEEP .5,2: BEEP 1,0
80 BEEP 1,0: BEEP 1,-5: BEEP 2,0
90 BEEP 1,0: BEEP 1,-5: BEEP 2,0
```

When you run this, you should get the funeral march from Mahler's first symphony, the bit where the goblins bury the US Cavalry man.

Suppose for example that your tune is written in the key of C minor, like the Mahler above. The beginning looks like this:



and you can write in the pitch values of the notes like this:



We have put in two ledger lines, just for good measure. Note how the E flat in the key signature affects not only the E in the top space, flattening it from 16 to 15, but also the E on the bottom line, flattening it from 4 to 3. It should now be quite easy to find the pitch value of any note on the stave.

If you want to change the key of the piece, the best thing is to set up a variable **key** and insert **key+** before each pitch value: thus the second line becomes

```
20 BEEP 1,key+0: BEEP 1,key+2: BEEP .5,key+3: BEEP .5,key+2:
   BEEP 1,key+0
```

Before you run a program you must give **key** the appropriate value – 0 for C minor, 2 for D minor, 12 for C minor an octave up, and so on. You can get the computer in tune with another instrument by adjusting **key**, using fractional values.

You also have to work out the durations of all the notes. Since this is a fairly slow piece, we have allowed one second for a crotchet and based the rest on that, half a second for a quaver and so on.

More flexible is to set up a variable **crotchet** to store the length of a crotchet and specify the durations in terms of this. Then line 20 would become

```
20 BEEP crotchet,key+0: BEEP crotchet,key+2: BEEP crotchet/
   2,key+3: BEEP crotchet/2,key+2: BEEP crotchet,key+0
```

(You will probably want to give **crotchet** and **key** shorter names.)

By giving **crotchet** appropriate values, you can easily vary the speed of the piece.

Remember that because there is only one loudspeaker in the computer you can only play one note at a time, so you are restricted to unharmonized tunes. If you want any more you must sing it yourself.

Try programming tunes in for yourself – start off with fairly simple ones like 'Three Blind Mice'. If you have neither piano nor written music, get hold of a very simple instrument like a tin whistle or a recorder, and work the tunes out on that. You could make a chart showing the pitch value for each note that you can play on this instrument.

Type:

```
FOR n=0 TO 1000: BEEP .5,n: NEXT n
```

This will play notes as high as it can, and then stop with error report **B integer out of range**. You can print out n to find out how high it did actually get.

Try the same thing, but going down into the low notes. The very lowest notes will just sound like clicks; in fact the higher notes are also made of clicks in the same way, but faster, so that the ear cannot distinguish them.

Only the middle range of notes are really any good for music; the low notes sound too much like clicks, and the high notes are thin and tend to warble a bit.

Type in this program line:

```
10 BEEP .5,0: BEEP .5,2: BEEP .5,4: BEEP.5,5: BEEP .5,7:
   BEEP .5,9: BEEP .5,11: BEEP .5,12: STOP
```

This plays the scale of C major, which uses all the white notes on the piano from middle C to the next C up. The way this scale is tuned is exactly the same as on a piano, and is called even-tempered tuning because the pitch interval of a semitone is the same all the way up the scale. A violinist, however, would play the scale very slightly differently, adjusting all the notes to make them sound more pleasing to the ear. He can do this just by moving his fingers very slightly up or down the string in a way that a pianist can't.

The natural scale, which is what the violinist plays, comes out like this:

```
20 BEEP .5,0: BEEP .5,2.039: BEEP .5,3.86: BEEP .5,4.98:
   BEEP .5,7.02: BEEP .5,8.84: BEEP .5,10.88: BEEP .5,12: STOP
```

You may or may not be able to detect any difference between these two; some people can. The first noticeable difference is that the third note is slightly flatter in the naturally tempered scale.If you are a real perfectionist, you might like to program your tunes to use this natural scale instead of the even-tempered one. The disadvantage is that although it works perfectly in the key of C, in other keys it works less well – they all have their own natural scales – and in some keys it works very badly indeed. The even-tempered scale is only slightly off, and works equally well in all keys.

This is less of a problem on the computer, of course, because you can use the trick of adding on a variable **key**,

Some music – notably Indian music – uses intervals of pitch smaller than a semitone. You can program these into the **BEEP** statement without any trouble; for instance the quartertone above middle C has a pitch value of .5.

You can make the keyboard beep instead of clicking by

POKE 23609,255

The second number in this determines the length of the beep (try various values between 0 and 255). When it is 0, the beep is so short that it sounds like a soft click.

If you are interested in doing more with sound from the Spectrum, like hearing the sound that **BEEP** makes on something other the internal speaker, you will find that the signal is present on both the 'MIC' and the 'EAR' sockets. It will be at a higher level on the 'EAR' socket, but otherwise they are the same. You may use this to connect an earphone or a pair of headphones to your Spectrum. This will not cut out the internal loudspeaker. If you are really keen to make a lot of noise you could connect it up to an amplifier – the 'MIC' socket will probably give about the right level – or you could record the sound onto tape and get the Spectrum to play along with itself.

You will not damage the Spectrum even if you short-circuit the 'MIC' or 'EAR' sockets, so experiment to find which gives the best output for what you want to do.

Exercise

1. Rewrite the Mahler program so that it uses **FOR** loops to repeat the bars.

Program the computer so that it plays not only the funeral march, but also the rest of Mahler's first symphony.

CHAPTER 20

Tape storage

Summary
LOAD, SAVE, VERIFY, MERGE

The basic methods for using the cassette recorder to **SAVE**, **LOAD** and **VERIFY** programs are given in the Introductory booklet. This section should be read, and the procedures tried out before reading any further here.

We have seen that **LOAD** deletes the old program and variables in the computer before loading in the new ones from tape; there is another statement, **MERGE**, that does not. **MERGE** only deletes an old program line or variable if it has to because there is a new one with the same line number or name. Type in the 'dice' program in Chapter 11 and save it on tape, as **"dice"**. Now enter and run the following:

```
 1 PRINT 1
 2 PRINT 2
10 PRINT 10
20 LET x=20
```

and then proceed as for the verification, but replacing **VERIFY "dice"** with

```
MERGE "dice"
```

If you list the program you can see that lines 1 and 2 have survived, but lines 10 and 20 have been replaced by those from the dice program. **x** has also survived (try **PRINT x**).

You have now seen simple forms of the four statements used with the cassette tape:

SAVE records the program and variables on to cassette.

VERIFY checks the program and variables on cassette against those already in the computer.

LOAD clears the computer of all its program and variables, and replaces them with new ones read in from cassette.

MERGE is like **LOAD** except that it does not clear out an old program line or variable unless it has to because its line number or name is the same as that of a new one from cassette.

In each of these, the keyword is followed by a string: for **SAVE** this provides a name for the program on tape, while for the other three it tells the computer which program to search for. While it is searching, it prints up the name of each program it comes across. There are a couple of twists to all this.

For **VERIFY**, **LOAD** and **MERGE** you can provide the empty string as the name to search for: then the computer does not care about the name, but takes the first program it comes across.

A variant on **SAVE** takes the form

SAVE string **LINE** number

A program saved using this is recorded in such a way that when it is read back by **LOAD** (but not **MERGE**) it automatically jumps to the line with the given number, thus running itself.

So far, the only kinds of information we have stored on cassette have been programs together with their variables. There are two other kinds as well, called *arrays* and *bytes*.

Arrays are dealt with slightly differently:

You can save arrays on tape using **DATA** in a **SAVE** statement by

SAVE string **DATA** array name()

String is the name that the information will have on tape and works in exactly the same way as when you save a program or plain bytes.

The array name specifies the array you want to save, so it is just a letter or a letter followed by $. Remember the brackets afterwards; you might think they are logically unnecessary but you still have to put them in to make it easier for the computer.

Be clear about the separate roles of *string* and *array name*. If you say (for instance)

SAVE "Bloggs" DATA b()

then **SAVE** takes the array *b* from the computer and stores it on tape under the name "Bloggs". When you type

VERIFY "Bloggs" DATA b()

the computer will look for a number array stored on tape under the name "Bloggs" (when it finds it it will write up 'Number array: Bloggs') and check it against the array *b* in the computer.

LOAD "Bloggs" DATA b()

finds the array on tape, and then – if there is room for it in the computer – delete any array already existing called *b* and loads in the new array from tape, calling it *b*.

You cannot use **MERGE** with saved arrays.

You can save character (string) arrays in exactly the same way. When the computer is searching the tape and finds one of these it writes up 'Character array:' followed by the name. When you load in a character array, it will delete not only any previous character array with the same name, but also any string with the same name.

Byte storage is used for pieces of information without any reference to what the information is used for – it could be television picture, or user-defined graphics, or

something you have made up for yourself. It is shown using the word **CODE**, as in

SAVE "picture" CODE 16384,6912

The unit of storage in memory is the *byte* (a number between 0 and 255), and each byte has an *address* (which is a number between 0 and 65535). The first number after **CODE** is the address of the first byte to be stored on tape, and the second is the number of bytes to be stored. In our case, 16384 is the address of the first byte in the display file (which contains the television picture) and 6912 is the number of bytes in it, so we are saving a copy of the television screen – try it. The name **"picture"** works just like the names for programs.

To load it back, use

LOAD "picture" CODE

You can put numbers after **CODE** in the form

LOAD name **CODE** start, length

Here *length* is just a safety measure; when the computer has found the bytes on tape with the right name, it will still refuse to load them in if there are more than *length* of them – since there is obviously more data than you expected it could otherwise overwrite something you had not intended to be overwritten. It gives the error report **R Tape loading error**. You can miss out *length*, and then the computer will read in the bytes however many there are.

Start shows the address where the first byte is to be loaded back to – this can be different from the address it was saved from, although if they are the same you can miss out *start* in the **LOAD** statement.

CODE 16384,6912 is so useful for saving and loading the picture that you can replace it with **SCREEN$** – for instance,

SAVE "picture" SCREEN$
LOAD "picture" SCREEN$

This is a rare case for which **VERIFY** will not work – **VERIFY** writes up the names of what it finds on tape, so that by the time it gets round to the verification the display file has been changed and the verification fails. In all other cases you should use **VERIFY** whenever you use **SAVE**.

Below, is a complete summary of the four statements used in this chapter.

Name stands for any string expression, and refers to the name under which the information is stored on cassette. It should consist of ASCII printing characters, of which only the first 10 are used.

There are four sorts of information that can be stored on tape: program and variables (together), number arrays, character arrays, and straight bytes.

When **VERIFY**, **LOAD** and **MERGE** are searching the tape for information with a given name and of a given sort, they print up on the screen the sort and name of all the information they find. The sort is shown by 'Program:', 'Number array:', 'Character array:' or 'Bytes:'. If *name* was the empty string, they take the first lot of information of the right sort, regardless of its name.

SAVE

Saves information on tape under the given name. Error F occurs when name is empty or has 11 or more characters.

SAVE always puts up a message **Start tape, then press any key**, and waits for a key to be pressed before saving anything.

1. Program and variables:

SAVE name **LINE** line number

saves the program and variables in such a way that **LOAD** automatically follows with

GO TO line number

2. Bytes:

SAVE name **CODE** start, length

saves *length* bytes starting at address *start*.

SAVE name **SCREEN$**

is equivalent to

SAVE name **CODE 16384,6912**

and saves the television picture.

3. Arrays:

SAVE name **DATA** letter ()

or

SAVE name **DATA** letter $ ()

saves the array whose name is *letter* or *letter $* (this need bear no relation to *name*).

VERIFY

Checks the information on tape against the information already in memory. Failure to verify gives error **R Tape loading error**.

1. Program and variables:

VERIFY name

2. Bytes:

VERIFY name **CODE** start, length

If the bytes *name* on tape are more than *length* in number, then gives error R. Otherwise, checks them against the bytes in memory starting at address *start*.

VERIFY name **CODE** start

checks the bytes *name* on tape against those in memory starting at address *start*.

VERIFY name **CODE**

checks the bytes *name* on tape against those in memory starting at the address from which the first cassette byte was saved.

VERIFY name **SCREEN$**

is equivalent to

VERIFY name **CODE 16384,6912**

but will almost certainly fail to verify.

3. Arrays:

VERIFY name **DATA** letter ()

or

VERIFY name **DATA** letter $ ()

checks the array *name* on tape against the array *letter* or *letter $* in memory.

LOAD

Loads new information from tape, deleting the old information from memory.

1. Program and variables:

LOAD name

deletes the old program and variables and loads in program and variables *name* from cassette; if the program was saved using **SAVE** name **LINE** it performs an automatic jump.

Error **4 Out of memory** occurs if there is no room for the new program and variables. In this case the old program and variables are not deleted.

2. Bytes:

LOAD name **CODE** start, length

If the bytes *name* from tape are more than *length* in number then gives error R. Otherwise, loads them into memory starting at address *start*, and overwriting whatever was there previously.

LOAD name **CODE** start

loads the bytes *name* from tape into memory, starting at address *start* and overwriting whatever was there previously.

LOAD name **CODE**

loads the bytes *name* from tape into memory starting at the address from which the first tape byte was saved and overwriting the bytes that were there in memory before.

3. Arrays:

LOAD name **DATA** letter ()

or

LOAD name **DATA** letter $ ()

deletes any array already called *letter* or *letter $* (as appropriate) and forms a new one from the array stored on cassette.

Error **4 Out of memory** occurs if no room for new arrays. Old arrays are not deleted.

MERGE

Loads new information from cassette without deleting old information from memory.

1. Program and variables:

MERGE name

merges the program *name* in with the one already in memory, overwriting any program lines or variables in the old program whose line numbers or names conflict with ones on the new program.

Error **4 Out of memory** occurs unless there is enough room in memory for all of the old program and variables *and* all of the new program and variables being loaded from tape.

2. Bytes:

Not possible

3. Arrays:

Not possible

Exercises

1. Make a cassette on which the first program, when loaded, prints a *menu* (a list of the other programs on the cassette), asks you to choose a program, and then loads it.

2. Get the chess piece graphics from Chapter 14, and then type **NEW**: they will survive this. However, they will not survive having the computer turned off: if you want to keep them, you must save them on tape, using **SAVE** with **CODE**. The easiest way is to save all twenty-one user-defined graphics by

SAVE "chess" CODE USR "a",21*8

followed by

VERIFY "chess" CODE

This is the system of *bytes* saving that was used for saving the picture. The address of the first byte to be saved is **USR "a"**, the address of the first of the eight

bytes that determine the pattern of the first user-defined graphic, and the number of bytes to be saved is 21*8 – eight bytes for each of 21 graphics.

To load back you would normally use

LOAD "chess" CODE

However, if you are loading back into a Spectrum with a different amount of memory, or if you have moved the user-defined graphics to a different address (you have to do this deliberately using more advanced techniques), you have to be more careful and use

LOAD "chess" CODE USR "a"

USR allows for the fact that the graphics must be loaded back to a different address.

CHAPTER 21

The ZX printer

Summary
LPRINT, LLIST, COPY

Note: None of these statements is standard BASIC, although **LPRINT** is used by some other computers.

If you have a ZX printer, you will have some operating instructions with it. This chapter covers the BASIC statements needed to make it work.

The first two, **LPRINT** and **LLIST**, are just like **PRINT** and **LIST**, except that they use the printer instead of the television. (The L is an historical accident. When BASIC was invented it usually used an electric typewriter instead of a television, so **PRINT** really did mean print. If you wanted masses of output you would use a very fast line printer attached to the computer, and an **LPRINT** statement meaning 'Line printer **PRINT**'.)

Try this program for example.

```
10 LPRINT "This program".'
20 LLIST
30 LPRINT '"prints out the character set."'
40 FOR n=32 TO 255
50 LPRINT CHR$ n;
60 NEXT n
```

The third statement, **COPY**, prints out a copy of the television screen. For instance, type **LIST** to get a listing on the screen of the program above, and type

```
COPY
```

Note that **COPY** doesn't work with one of the listings that the computer puts up automatically, because that is cleared whenever a command is obeyed. You must either use **LIST** first, or use **LLIST** and forget about **COPY**.

You can always stop the printer when it is running by pressing the **BREAK** key (**CAPS SHIFT** and **SPACE**).

If you execute these statements without the printer attached, it should lose all the output and carry on with the next statement.

Try this:

```
10 FOR n=31 TO 0 STEP -1
20 PRINT AT 31-n,n; CHR$ (CODE "0"+n);
30 NEXT n
```

You will see a pattern of characters working down diagonally from the top right-hand corner until it reaches the bottom of the screen, when the program asks if

you want to scroll.

Now change **AT 31−n,n** in line 20 to **TAB n**. The program will have exactly the same effect as before.

Now change **PRINT** in line 20 to **LPRINT**. This time there will be no **scroll?**, which should not occur with the printer, and the pattern will carry straight on with the letters F to O.

Now change **TAB n** to **AT 31−n,n** still using **LPRINT**. This time you will get just a single line of symbols. The reason for the difference is that the output from **LPRINT** is not printed straight away, but arranges in a buffer store a picture one line long of what the computer will send to the printer when it gets round to it. The printing takes place

(i) when the buffer is full,
(ii) after an **LPRINT** statement that does not end in a comma or semicolon,
(iii) when a comma, apostrophe or **TAB** item requires a new line, or
(iv) at the end of a program, if there is anything left unprinted.

(iii) explains why our program with **TAB** works the way it does. As for **AT**, the line number is ignored and the **LPRINT** position (like the **PRINT** position, but for the printer instead of the television) is changed to the column number. An **AT** item can never cause a line to be sent to the printer.

Exercise

1. Make a printed graph of **SIN** by running the program in Chapter 17 and then using **COPY**.

CHAPTER 22

Other Equipment

There is other equipment that you will be able to attach to the Spectrum.

The ZX Microdrive is a high speed mass storage device, and is much more flexible in the way it can be used than a cassette recorder. It will operate not only with **SAVE**, **VERIFY**, **LOAD** and **MERGE**, but also with **PRINT**, **LIST**, **INPUT** and **INKEY$**.

The network is used for connecting several Spectrums so that they can talk to each other – one of the uses of this is that you then need only one Microdrive to serve several computers.

The RS232 interface is a standard connection that allows you to link a Spectrum with keyboards, printers, computers and various other machines even if they were not designed specifically for the Spectrum.

These will use some extra keywords that are on the keyboard, but cannot be used without the extra attachments: they are **OPEN#**, **CLOSE#**, **MOVE**, **ERASE**, **CAT** and **FORMAT**.

IN and OUT

Summary
OUT
IN

The processor can read from and (at least with RAM) write to memory by using **PEEK** and **POKE**. The processor itself does not really care whether memory is ROM, RAM or even nothing at all; it just knows that there are 65536 memory addresses, and it can read a byte from each one (even if it's nonsense), and write a byte to each one (even if it gets lost). In a completely analogous way there are 65536 of what are called *I/O ports* (standing for Input/Output ports). These are used by the processor for communicating with things like the keyboard or the printer, and they can be controlled from the BASIC by using the **IN** function and the **OUT** statement.

IN is a function like **PEEK**.

IN address

It has one argument, the port address, and its result is a byte read from that port.

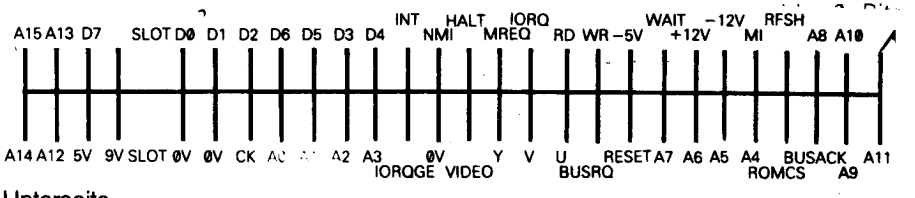
OUT is a statement like **POKE**.

OUT address, value

writes the given value to the port with the given address. How the address is interpreted depends very much on the rest of the computer; quite often, many different addresses will mean the same. On the Spectrum it is most sensible to imagine the address being written in binary, because the individual bits tend to work independently. There are 16 bits, which we shall call (using A for *address*)

A15, A14, A13, A12,....., A2, A1, A0

Here A0 is the 1s bit, A1 the 2s bit, A2 the 4s bit and so on. Bits A0, A1, A2, A3 and A4 are the important ones. They are normally 1, but if any one of them is 0 this tells the computer to do something specific. The computer cannot cope with more than



The keyboard is divided up into 8 half rows of 5 keys each.

IN 65278 reads the half row **CAPS SHIFT** to **V**
IN 65022 reads the half row **A** to **G**
IN 64510 reads the half row **Q** to **T**
IN 63486 reads the half row **1** to **5**
IN 61438 reads the half row **0** to **6**
IN 57342 reads the half row **P** to **7**
IN 49150 reads the half row **ENTER** to **H**
IN 32766 reads the half row **SPACE** to **B**

(These addresses are 254+256*(255−2 ↑ n) as *n* goes from 0 to 7.)

In the byte read in, bits D0 to D4 stand for the five keys in the given half row – D0 for the outside key, D4 for the one nearest the middle. The bit is 0 if the key is pressed, 1 if it is not. D6 is the value at the EAR socket.

Port address 254 in output drives the loudspeaker (D4) and the MIC socket (D3), and also sets the border colour (D2, D1 and D0).

Port address 251 runs the printer, both in reading and writing: reading finds out whether the printer is ready for more, and writing sends out dots to be printed.

Port addresses 254, 247 and 239 are used for the extra devices mentioned in Chapter 22.

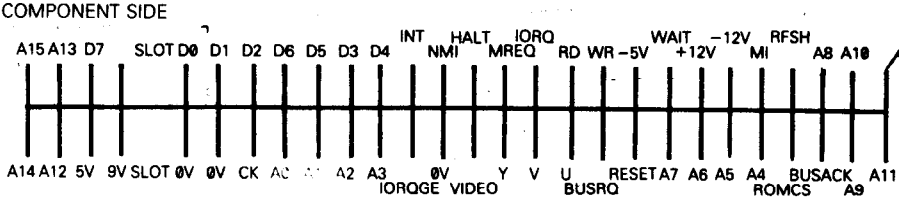
Run this program

```
10 FOR n=0 TO 7: REM half-row number
20 LET a=254+256*(255−2 ↑ n)
30 PRINT AT 0,0; IN a: GO TO 30
```

and play around by pressing keys. When you get bored with each half-row, press **BREAK** and then type

```
NEXT n
```

The control, data and address busses are all exposed at the back of the Spectrum, so you can do almost anything with a Spectrum that you can with a Z80. Sometimes, though, the Spectrum hardware might get in the way. Here is a diagram of the exposed connections at the back:



Unterseite

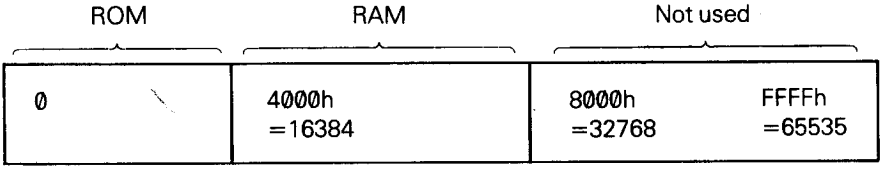
CHAPTER 24

The memory

Summary
CLEAR

Deep inside the computer, everything is stored as bytes, i.e. numbers between 0 and 255. You may think you have stored away the price of wool or the address of your fertilizer suppliers, but it has all been converted into collections of bytes and bytes are what the computer sees.

Each place where a byte can be stored has an address, which is a number between 0 and FFFFh (so an address can be stored as two bytes), so you might think of the memory as a long row of numbered boxes, each of which can contain a byte. Not all the boxes are the same, however. In the standard 16K RAM machine, the boxes from 8000h to FFFFh are simply missing altogether. The boxes from 4000h to 7FFFh are RAM boxes, which means you can open the lid and alter the contents, and those from 0 to 3FFFh are ROM boxes, which have glass tops but cannot be opened. You just have to read whatever was put in them when the computer was made.



To inspect the contents of a box, we use the **PEEK** function: its argument is the address of the box, and its result is the contents. For instance, this program prints out the first 21 bytes in ROM (and their addresses):

```
10 PRINT "Address"; TAB 8; "Byte"
20 FOR a=0 TO 20
30 PRINT a; TAB 8; PEEK a
40 NEXT a
```

All these bytes will probably be quite meaningless to you, but the processor chip understands them to be instuctions telling it what to do.

To change the contents of a box (if it is RAM), we use the **POKE** statement. It has the form

POKE address, new contents

where 'address' and 'new contents' stand for numeric expressions. For instance, if you say

POKE 31000,57

the byte at address 31000 is given the new value 57 – type

PRINT PEEK 31000

to prove this. (Try poking in other values, to show that there is no cheating.) The new value must be between −255 and +255, and if it is negative then 256 is added to it.

The ability to poke gives you immense power over the computer if you know how to wield it; and immense destructive possibilities if you don't. It is very easy, by poking the wrong value in the wrong address, to lose vast programs that took you hours to type in. Fortunately, you won't do the computer any permanent damage.

We shall now take a more detailed look at how the RAM is used but don't bother to read this unless you're interested.

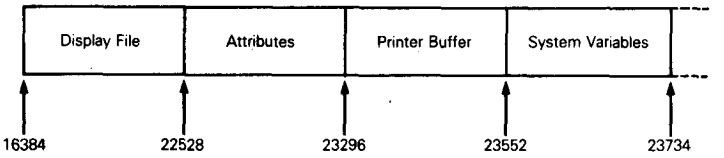
The memory is divided into different areas (shown on the big diagram) for storing different kinds of information. The areas are only large enough for the information that they actually contain, and if you insert some more at a given point (for instance by adding a program line or variable) space is made by shifting up everything above that point. Conversely, if you delete information then everything is shifted down.

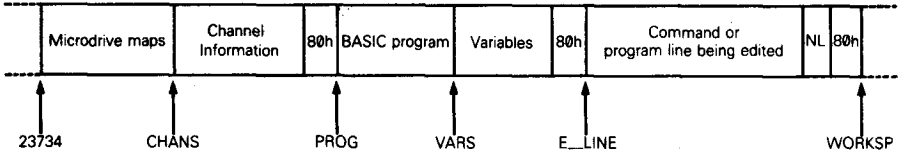
The display file stores the television picture. It is rather curiously laid out, so you probably won't want to **PEEK** or **POKE** in it. Each character position on the screen has an 8×8 square of dots, and each dot can be either 0 (paper) or 1 (ink): and by using binary notation we can store the pattern as 8 bytes, one for each row. However, these 8 bytes are not stored together. The corresponding rows in the 32 characters of a single line are stored together as a scan of 32 bytes, because this is what the electron beam in the television needs as it scans from the left hand side of the screen to the other. Since the complete picture has 24 lines of 8 scans each, you might expect the total of 172 scans to be stored in order, one after the other; you'd be wrong. First come the top scans of lines 0 to 7, then the next scans of lines 0 to 7, and so on to the bottom scans of lines 0 to 7; then the same for lines 8 to 15; and then the same for lines 16 to 23. The upshot of all this is that if you're used to a computer that uses **PEEK** and **POKE** on the screen, you'll have to start using **SCREEN$** and **PRINT AT** instead, or **PLOT** and **POINT**.

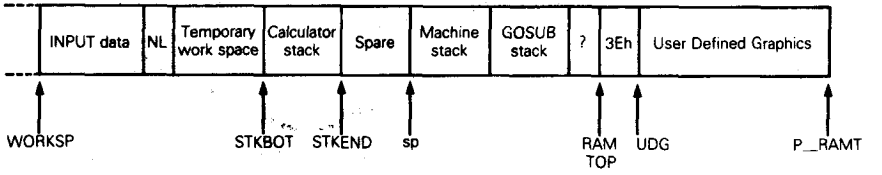
The attributes are the colours and so on for each character position, using the format of **ATTR**. These are stored line by line in the order you'd expect.

The printer buffer stores the characters destined for the printer.

The system variables contain various pieces of information that tell the computer what sort of state the computer is in. They are listed fully in the next chapter, but for the moment note that there are some (called CHANS, PROG, VARS, E_LINE and so on) that contain the addresses of the boundaries between the various areas in memory. These are not BASIC variables, and their names will not be recognized by the computer.


Display File
Attributes
Printer Buffer
System Variables
16384
22528
23296
23552
23734

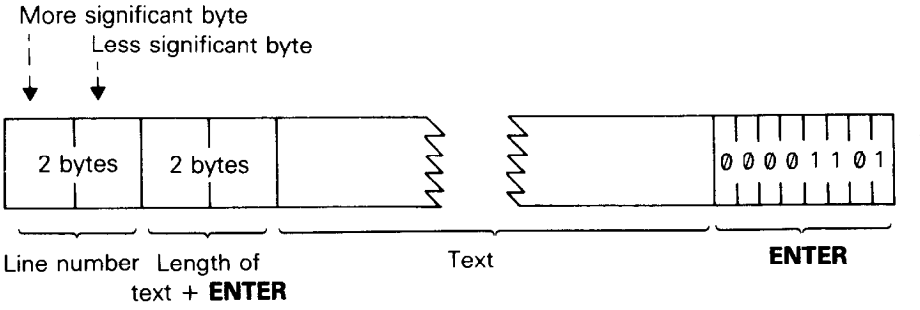

Microdrive maps
Channel Information
80h
BASIC program
Variables
80h
Command or program line being edited
NL
80h
23734
CHANS
PROG
VARS
E_LINE
WORKSP


INPUT data
NL
Temporary work space
Calculator stack
Spare
Machine stack
GOSUB stack
?
3Eh
User Defined Graphics
WORKSP
STKBOT
STKEND
sp
RAM TOP
UDG
P_RAMT

The Microdrive maps are only used with the Microdrive. Normally there is nothing there.

The channel information contains information about the input and output devices, namely the keyboard (with the lower half of the screen), the upper half of the screen, and the printer.

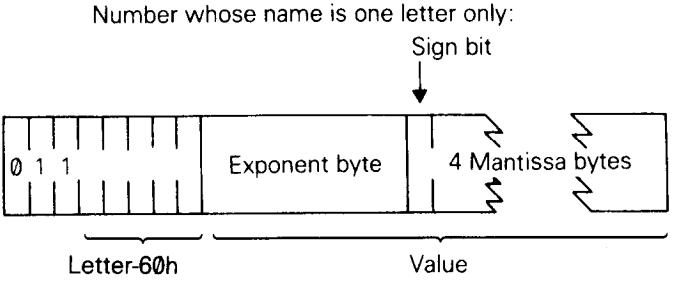
Each line of BASIC program has the form:



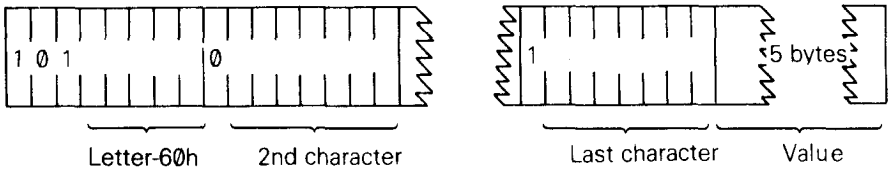
Note that, in contrast with all other cases of two-byte numbers in the Z80, the line number here is stored with its more significant byte first: that is to say, in the order that you write them down in.

A numerical constant in the program is followed by its binary form, using the character **CHR$ 14** followed by five bytes for the number itself.

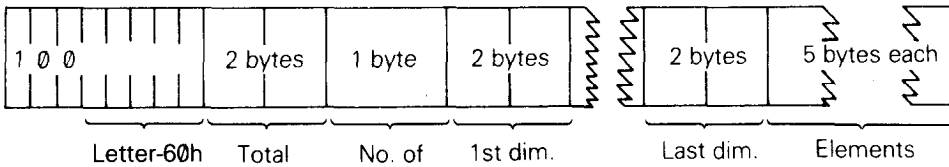
The variables have different formats according to their different ıatures. The letters in the names should be imagined as starting off in lower case.



Number whose name is longer than one letter:



Array of numbers:

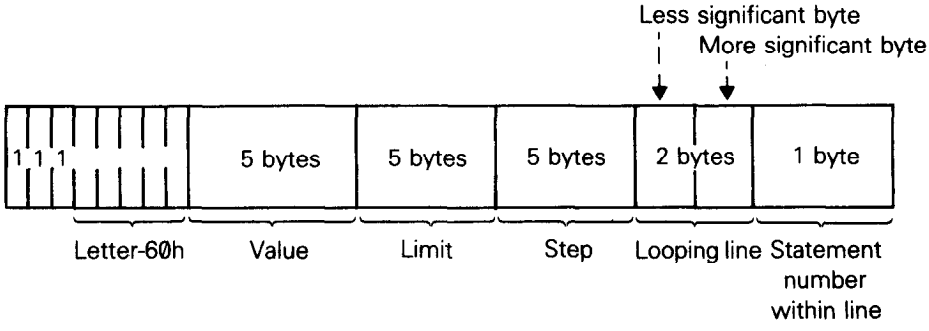


The order of the elements is:
first, the elements for which the first subscript is 1
next, the elements for which the first subscript is 2
next, the elements for which the first subscript is 3
and so on for all possible values of the first subscript.

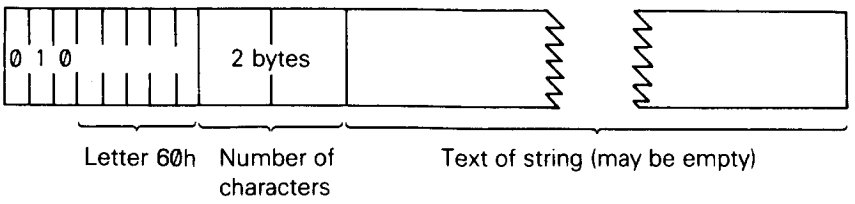
The elements with a given first subscript are ordered in the same way using the second subscript, and so on down to the last.

As an example, the elements of the 3*6 array b in Chapter 12 are stored in the order b(1,1) b(1,2) b(1,3) b(1,4) b(1,5) b(1,6) b(2,1) b(2,2) .... b(2,6) b(3,1) b(3,2) ... b(3,6).

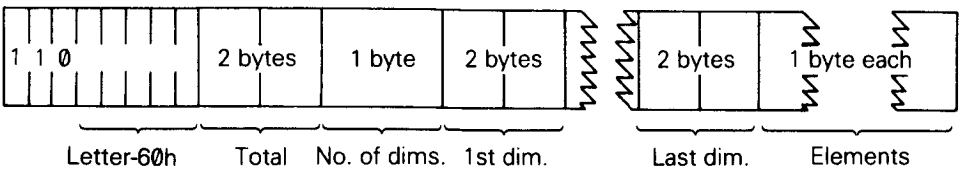
Control variable of a **FOR–NEXT** loop:



String:



Array of characters:



The calculator is the part of the BASIC system that deals with arithmetic, and the numbers on which it is operating are held mostly in the calculator stack.

The spare part contains the space so far unused.

The machine stack is the stack used by the Z80 processor to hold return addresses and so on.

The **GOSUB** stack was mentioned in Chapter 5.

The byte pointed to by RAMTOP has the highest address used by the BASIC system. Even **NEW**, which clears the RAM out, only does so as far as this – so it doesn't change the user-defined graphics. You can change the address RAMTOP by putting a number in a clear statement:

CLEAR new RAMTOP

This

(i) clears out all the variables
(ii) clears the display file (like **CLS**)
(iii) resets the **PLOT** position to the bottom left-hand corner
(iv) does **RESTORE**
(v) clears the **GOSUB** stack and puts it at the new RAMTOP – assuming that this

lies between the calculator stack and the physical end of RAM; otherwise it leaves RAMTOP as it was.

RUN also does **CLEAR**, although it never changes RAMTOP.

Using **CLEAR** in this way, you can either move RAMTOP up to make more room for the BASIC by overwriting the user-defined graphics, or you can move it down to make more RAM that is preserved from **NEW**.

Type **NEW**, then **CLEAR 23800** to get some idea of what happens to the machine when it fills up.

One of the first things you will notice if you start typing in a program is that after a while the computer stops accepting any more and buzzes at you. It means the computer is chock-a-block and you will have to empty it slightly. There are also two error messages with roughly the same meaning, **4 Memory full** and **G No room for line**.

The buzz also occurs when you type in a line longer than 23 lines – then your typing is not being ignored, though you cannot see it; but the buzz sounds to discourage you from doing any more.

You can adjust the length of the buzz by poking a number into address 23608. The usual length has number 64.

Any number (except 0) can be written uniquely as

$$\pm m \times 2^e$$

where $\pm$ is the sign,

m is the *mantissa*, and lies between $\frac{1}{2}$ and 1 (it cannot be 1),

and e is the *exponent*, a whole number (possibly negative).

Suppose you write m in the binary scale. Because it is a fraction, it will have a *binary point* (like the decimal point in the scale of ten) and then a binary fraction (like a decimal fraction): so in binary,

a half is written .1

a quarter is written .01

three quarters is written .11

a tenth is written .0001100110011001100110011 . . . and so on. With our number m, because it is less than 1, there are no bits before the binary point, and because it is at least $\frac{1}{2}$, the bit immediately after the binary point is a 1.

To store the number in the computer, we use five bytes, as follows:

(i) write the first eight bits of the mantissa in the second byte (we know that the first bit is 1), the second eight bits in the third byte, the third eight bits in the fourth byte and the fourth eight bits in the fifth byte,

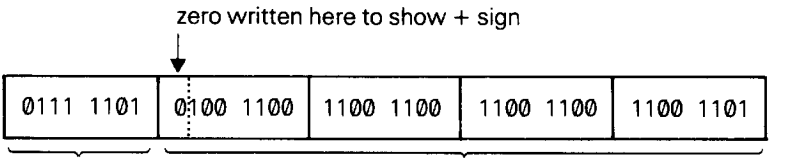
(ii) replace the first bit in the second byte – which we know is 1 – by the sign: 0 for plus, 1 for minus,

(iii) write the exponent +128 in the first byte. For instance, suppose our number is 1/10

$$1/10 = 4/5 \times 2^{-3}$$

Thus the mantissa m is .11001100110011001100110011001100 in binary (since

the 33rd bit is 1, we shall round the 32nd up from Ø to 1), and the exponent e is −3. Applying our three rules gives the five bytes



There is an alternative way of storing whole numbers between −65535 and +65535:

(i) the first byte is Ø,
(ii) the second byte is Ø for a positive number, FFh for a negative one,
(iii) the third and fourth bytes are the less and more significant bytes of the number (or the number +131072 if it is negative),
(iv) the fifth byte is Ø.

Reports

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.

The report has a code number or letter so that you can refer to the table here, a brief message explaining what happened and the line number and statement number within that line where it stopped. (A command is shown as line 0. Within a line, statement 1 is at the beginning, statement 2 comes after the first colon or **THEN**, and so on.)

The behaviour of **CONTINUE** depends very much on the reports. Normally, **CONTINUE** goes to the line and statement specified in the last report, but there are exceptions with reports 0, 9 and D (also see Appendix C).

Here is a table showing all the reports. It also tells you in what circumstances the report can occur, and this refers you to Appendix C. For instance error **A Invalid argument** can occur with **SQR**, **IN**, **ACS** and **ASN** and the entries for these in Appendix C tell you exactly what arguments are invalid.

Code	*Meaning*	*Situations*
0	OK Successful completion, or jump to a line number bigger than any existing. This report does not change the line and statement jumped to by **CONTINUE**.	Any
1	NEXT without FOR The control variable does not exist (it has not been set up by a **FOR** statement), but there is an ordinary variable with the same name.	**NEXT**
2	Variable not found For a simple variable this will happen if the variable is used before it has been assigned to in a **LET**, **READ** or **INPUT** statement or loaded from tape or set up in a **FOR** statement. For a subscripted variable it will happen if the variable is used before it has been dimensioned in a **DIM** statement or loaded from tape.	Any
3	Subscript wrong A subscript is beyond the dimension of the array, or there are the wrong number of subscripts. If the subscript is negative or bigger than 65535, then error B will result.	Subscripted variables, Substrings
4	Out of memory There is not enough room in the computer for what	**LET**, **INPUT**, **FOR**, **DIM**, **GO SUB**, **LOAD**,

Code	*Meaning*	*Situations*
	you are trying to do. If the computer really seems to be stuck in this state, you may have to clear out the command line using **DELETE** and then delete a program line or two (with the intention of putting them back afterwards) to give yourself room to manoeuvre with – say – **CLEAR**.	**MERGE**. Sometimes during expression evaluation
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, . . .	**INPUT, PRINT AT**
6	Number too big Calculations have led to a number greater than about 10^{38}.	Any arithmetic
7	RETURN without GO SUB There has been one more **RETURN** than there were **GO SUB**s.	**RETURN**
8	End of file	Microdrive, etc, operations
9	STOP statement After this, **CONTINUE** will not repeat the **STOP**, but carries on with the statement after.	**STOP**
A	Invalid argument The argument for a function is no good for some reason.	**SQR, LN, ASN, ACS, USR** (with string argument)
B	Integer out of range When an integer is required, the floating point argument is rounded to the nearest integer. If this is outside a suitable range then error B results.	**RUN, RANDOMIZE, POKE, DIM, GO TO, GO SUB, LIST, LLIST, PAUSE, PLOT, CHR$, PEEK, USR** (with numeric argument)
	For array access, see also Error 3.	Array access
C	Nonsense in BASIC The text of the (string) argument does not form a valid expression.	**VAL, VAL$**
D	BREAK – CONT repeats **BREAK** was pressed during some peripheral operation. The behaviour of **CONTINUE** after this report is normal in that it repeats the statement. Compare with report L.	**LOAD, SAVE, VERIFY, MERGE, LPRINT, LLIST, COPY**. Also when the computer asks **scroll?** and you type **N, SPACE** or **STOP**

Code	*Meaning*	*Situations*
E	Out of DATA You have tried to **READ** past the end of the **DATA** list.	**READ**
F	Invalid file name **SAVE** with name empty or longer than 10 characters.	**SAVE**
G	No room for line There is not enough room left in memory to accommodate the new program line.	Entering a line into the program
H	STOP in INPUT Some **INPUT** data started with **STOP**, or – for **INPUT LINE** – was pressed. Unlike the case with report 9, after report H **CONTINUE** will behave normally, by repeating the **INPUT** statement.	**INPUT**
I	FOR without NEXT There was a **FOR** loop to be executed no times (e.g. **FOR** n=1 **TO** 0) and the corresponding **NEXT** statement could not be found.	**FOR**
J	Invalid I/O device	Microdrive, etc, operations
K	Invalid colour The number specified is not an appropriate value.	**INK**, **PAPER**, **BORDER**, **FLASH**, **BRIGHT**, **INVERSE**, **OVER**; also after one of the corresponding control characters
L	BREAK into program **BREAK** pressed, this is detected between two statements. The line and statement number in the report refer to the statement before **BREAK** was pressed, but **CONTINUE** goes to the statement after (allowing for any jumps to be done), so it does not repeat any statements.	Any
M	RAMTOP no good The number specified for RAMTOP is either too big or too small.	**CLEAR**; possibly in **RUN**
N	Statement lost Jump to a statement that no longer exists.	**RETURN**, **NEXT**, **CONTINUE**
O	Invalid stream	Microdrive, etc, operations

Code	*Meaning*	*Situations*
P	FN without DEF User-defined function	**FN**
Q	Parameter error Wrong number of arguments, or one of them is the wrong type (string instead of number or vice versa).	**FN**
R	Tape loading error A file on tape was found but for some reason could not be read in, or would not verify.	**VERIFY LOAD** or **MERGE**

A description of the ZX Spectrum for reference

The first section of this appendix is a repeat of that part of the Introduction concerning the keyboard and screen.

The keyboard

ZX Spectrum characters comprise not only the single *symbols* (letters, digits, etc), but also the compound *tokens* (keywords, function names, etc) and all these are entered from the keyboard rather than being spelled out. To obtain all these functions and commands some keys have five or more distinct meanings, given partly by shifting the keys (i.e. pressing either the **CAPS SHIFT** key or the **SYMBOL SHIFT** key at the same time as the required one) and partly by having the machine in different *modes*.

The mode is indicated by the *cursor*, a flashing letter that shows where the next character from the keyboard will be inserted.

K (for keywords) mode automatically replaces L mode when the machine is expecting a command or program line (rather than **INPUT** data), and from its position on the line it knows it should expect a line number or a keyword. This is at the beginning of the line, or just after **THEN**, or just after : (except in a string). If unshifted, the next key will be interpreted as either a keyword (written on the keys), or a digit.

L (for letters) mode normally occurs at all other times. If unshifted, the next key will be interpreted as the main symbol on that key, in lower case for letters.

In both K and L modes, **SYMBOL SHIFT** and a key will be interpreted as the subsidiary red character on the key and **CAPS SHIFT** with a digit key will be interpreted as the control function written in white above the key. **CAPS SHIFT** with other keys does not affect the keywords in K mode, and in L mode it converts lower case to capitals.

C (for capitals) mode is a variant of L mode in which all letters appear as capitals. **CAPS LOCK** causes a change from L mode to C mode or back again.

E (for extended) mode is used for obtaining further characters, mostly tokens. It occurs after both shift keys are pressed together, and lasts for one key depression only. In this mode, a letter gives one character or token (shown in green above it) if unshifted, and another (shown in red below it) if pressed with either shift. A digit key gives a token if pressed with **SYMBOL SHIFT**; otherwise it gives a colour control sequence.

G (for graphics) mode occurs after **GRAPHICS** (**CAPS SHIFT** and **9**) is pressed, and lasts until it is pressed again. A digit key will give a mosaic graphic, quit **GRAPHICS** or **DELETE**, and each of the letter keys apart from V, W, X, Y and Z, will give a user-defined graphic.

If any key is held down for more than about 2 or 3 seconds, it will start repeating.

Keyboard input appears in the bottom half of the screen as it is typed, each character (single symbol or compound token) being inserted just before the cursor. The cursor can be moved left with **CAPS SHIFT** and **5**, or right with **CAPS SHIFT** and

8. The character before the cursor can be deleted with **DELETE** (**CAPS SHIFT** and **0**). (Note: the whole line can be deleted by typing **EDIT** (**CAPS SHIFT** and **1**) followed by **ENTER**.)

When **ENTER** is pressed, the line is executed, entered into the program, or used as **INPUT** data as appropriate, unless it contains a syntax error. In this case a flashing ? appears next to the error.

As program lines are entered, a listing is displayed in the top half of the screen. The manner in which the listing is produced is rather complicated, and explained more fully in Chapter 2. The last line entered is called the *current* line and is indicated by the symbol ▶, but this can be changed by using the keys ▼ (**CAPS SHIFT** and **6**) and ▲ (**CAPS SHIFT** and **7**). If **EDIT** (**CAPS SHIFT** and **1**) is pressed, the current line is brought down to the bottom part of the screen and can be edited.

When a command is executed or a program run, output is displayed in the top half of the screen and remains until a program line is entered, or **ENTER** is pressed with an empty line, or ▲ or ▼ is pressed. In the bottom part appears a report giving a code (digit or letter) referring you to Appendix B, a brief verbal summary of what Appendix B says, the number of the line containing the last statement executed (or **0** for a command) and the position of the statement within the line. The report remains on the screen until a key is pressed (and indicates K mode).

In certain circumstances, **CAPS SHIFT** with the **SPACE** key acts as a **BREAK**, stopping the computer with report **D** or **L**. This is recognised

(i) at the end of a statement while a program is running, or
(ii) while the computer is using the cassette recorder or printer.

The television screen

This has 24 lines, each 32 characters long, and is divided into two parts. The top part is at most 22 lines and displays either a listing or program output. When printing in the top part has reached the bottom, it all scrolls up one line; if this would involve losing a line that you have not had a chance to see yet, then the computer stops with the message **scroll?**. Pressing the keys **N**, **SPACE** or **STOP** will make the program stop with report **D BREAK - CONT repeats**; any other key will let the scrolling continue. The bottom part is used for inputting commands, program lines, and **INPUT** data, and also for displaying reports. The bottom part starts of as two lines (the upper one blank), but it expands to accommodate whatever is typed in. When it reaches the current print position in the top half, further expansions will make the top half scroll up.

Each character position has *attributes* specifying its *paper* (background) and *ink* (foreground) colours, a two-level brightness, and whether it flashes or not. The available colours are black, blue, red, magenta, green, yellow and white.

The edge of the screen can be set to any of the colours using the border statement.

A character position is divided into 8×8 pixels and high resolution graphics are obtained by setting the pixels individually to show either the ink or paper colour for

that character position.

The attributes at a character position are adjusted whenever a character is written there or a pixel is plotted. The exact manner of the adjustment is determined by the *printing parameters*, of which there are two sets (called *permanent* and *temporary*) of six: the **PAPER**, **INK**, **FLASH**, **BRIGHT**, **INVERSE** and **OVER** parameters. Permanent parameters for the top part are set up by **PAPER**, **INK**, etc, statements, and last until further notice. (Initially they are black ink on white paper. With normal brightness, no flashing, normal video and no overprinting). Permanent parameters for the bottom part use the border colour as the paper colour, with a black or white contrasting ink colour, normal brightness, no flashing, normal video and no overprinting.

Temporary parameters are set up by **PAPER**, **INK**, etc, items, which are embedded in **PRINT**, **LPRINT**, **INPUT**, **PLOT**, **DRAW** and **CIRCLE** statements, and also by **PAPER**, **INK**, etc control characters when they are printed to the television – they are followed by a further byte to specify the parameter value. Temporary parameters last only to the end of the **PRINT** (or whatever) statement, or, in **INPUT** statements, until some **INPUT** data is needed from the keyboard, when they are replaced by the permanent parameters.

PAPER and **INK** parameters are in the range 0 to 9. Parameters 0 to 7 are the colours used when a character is printed:

0 black
1 blue
2 red
3 magenta
4 green
5 cyan
6 yellow
7 white

Parameter 8 ('transparent') specifies that the colour on the screen is to be left unchanged when a character is printed.

Parameter 9 ('contrast') specifies that the colour in question (paper or ink) is to be made either white or black to show up against the other colour.

FLASH and **BRIGHT** parameters are 0, 1 or 8: 1 means that flashing or brightness is turned on, 0 that it is turned off, and 8 ('transparent') that it is left unchanged at any character position.

OVER and **INVERSE** parameters are 0 or 1.

OVER 0 new characters obliterate old ones
OVER 1 the bit patterns of the old and new characters are combined using an 'exclusive or' operation (*overprinting*)
INVERSE 0 new characters are printed as ink colour on paper colour (*normal video*)
INVERSE 1 new characters are printed as paper colour on ink colour (*inverse video*)

When a **TAB** control character is received by the television, two more bytes are expected to specify a tab stop n (less significant byte first). This is reduced modulo 32 to n_0 (say), and then sufficient spaces are printed to move the printing position into column n_0.

When a comma control character is received, then sufficient spaces (at least one) are printed to move the printing position into column 0 or column 16.

When an **ENTER** control character is received, the printing position is moved on to the next line.

The printer

Output to the ZX printer is via a buffer one line (32 characters) long, and a line is sent to the printer

(i) when printing spills over from one line to the next,
(ii) when an **ENTER** character is received,
(iii) at the end of the program, if there is anything left unprinted,
(iv) when a **TAB** control or comma control moves the printing position on to a new line.

TAB controls and comma controls output spaces in the same way as on the television.

The **AT** control changes the printing position using the column number, and ignores the line number.

The printer is affected by **INVERSE** and **OVER** controls (and also statements) in the same way as the screen is, but not by **PAPER**, **INK**, **FLASH** or **BRIGHT**.

The printer will stop with error B if **BREAK** is pressed.

If the printer is absent the output will simply be lost.

The BASIC

Numbers are stored to an accuracy of 9 or 10 digits. The largest number you can get is about 10^{38}, and the smallest (positive) number is about $4*10^{-39}$.

A number is stored in the ZX Spectrum in floating point binary with one exponent byte e ($1<=e<=255$), and four mantissa bytes m ($\frac{1}{2}<=m<1$). This represents the number $m*2^{e-128}$.

Since $\frac{1}{2}<=m<1$, the most significant bit of the mantissa m is always 1. Therefore in actual fact we can replace it with a bit to show the sign – 0 for positive numbers, 1 for negative.

Small integers have a special representation in which the first byte is 0, the second is a sign byte (0 or FFh) and the third and fourth are the integer in twos complement form, the less significant byte first.

Numeric variables have names of arbitrary length, starting with a letter and continuing with letters and digits. Spaces and colour controls are ignored and all letters are converted to lower-case letters.

Control variables of **FOR–NEXT** loops have names a single letter long.

Numeric arrays have names a single letter long, which may be the same as the name of a simple variable. They may have arbitrarily many dimensions of arbitrary size. Subscripts start at 1.

Strings are completely flexible in length. The name of a string consists of a single letter followed by $.

String arrays can have arbitrarily many dimensions of arbitrary size. The name is a single letter followed by $ and may not be the same as the name of a string. All the strings in a given array have the same fixed length, which is specified as an extra, final dimension in the **DIM** statement. Subscripts start at 1.

Slicing: Substrings of strings may be specified using *slicers*. A slicer can be

(i) empty
or
(ii) numerical expression
or
(iii) optional numerical expression **TO** *optional numerical expression*

and is used in expressing a substring either by

(a) string expression (slicer)
(b) *string array variable (subscript,..., subscript, slicer)*

which means the same as

string array variable (subscript....., subscript) (slicer)

In (a), suppose the string expression has the value s$.
If the slicer is empty, the result is s$ considered as a substring of itself.

If the slicer is a numerical expression with value m, then the result is the mth character of s$ (a substring of length 1).

If the slicer has the form (iii), then suppose the first numerical expression has the value m (the default value is 1), and the second, n (the default value is the length of s$).

If 1<=m<=n<=the length of s$ then the result is the substring of s$ starting with the mth character and ending with the nth.

If 0<=n<m then the result is the empty string.

Otherwise, error 3 results.

Slicing is performed before functions or operations are evaluated, unless brackets dictate otherwise.

Substrings can be assigned to (see **LET**).

If a string quote is to be written in a string literal, then it must be doubled.

Functions

The argument of a function does not need brackets if it is a constant or a (possibly subscripted or sliced) variable.

Function	*Type of argument* (x)	*Result*
ABS	number	Absolute magnitude
ACS	number	Arccosine in radians. Error A if x not in the range −1 to +1
AND	binary operation, right operand always a number. Numeric left operand: String left operand:	A **AND** B = { A if B<>0 ; 0 if B=0 } A$ **AND** B = { A$ if B<>0 ; "" if B=0 }
ASN	number	Arcsine in radians. Error A if x not in the range −1 to +1
ATN	number	Arctangent in radians
ATTR	two arguments, x and y, both numbers; enclosed in brackets	A number whose binary form codes the attributes of line x, column y on the television. Bit 7 (most significant) is 1 for flashing, 0 for not flashing. Bit 6 is 1 for bright, 0 for normal. Bits 5 to 3 are the paper colour. Bits 2 to 0 are the ink colour. Error B unless 0<=x<=23 and 0<=y<=31
BIN		This is not really a function, but an alternative notation for numbers: **BIN** followed by a sequence of 0s and 1s is the number with such a representation in binary

Function	*Type of argument*	*Result*
CHR$	number	The character whose code is x, rounded to the nearest integer
CODE	string	The code of the first character in x (or 0 if x is the empty string)
COS	number (in radians)	Cosine x
EXP	number	e^x
FN		**FN** followed by a letter calls up a user-defined function (see **DEF**). The arguments must be enclosed in brackets; even if there are no arguments the brackets must still be present.
IN	number	The result of inputting at processor level from port x (0<=x<=FFFFh) (loads the bc register pair with x and does the assembly language instruction in a(c))
INKEY$	none	Reads the keyboard. The result is the character representing (in L or C mode) the key pressed if there is exactly one, else the empty string.
INT	number	Integer part (always rounds down)
LEN	string	Length
LN	number	Natural logarithm (to base e). Error A if x<=0
NOT	number	0 if x<>0, 1 if x=0. **NOT** has priority 4
OR	binary operation, both operands numbers	a **OR** b= { 1 if b<>0 ; a if b=0 } **OR** has priority 2
PEEK	number	The value of the byte in memory whose address is x (rounded to the nearest integer). Error B if x is not in the range 0 to 65535
PI	none	π (3.14159265...)
POINT	Two arguments, x and y, both numbers; enclosed in brackets	1 if the pixel at (x,y) is ink colour. 0 if it is paper colour. Error B unless 0<=x<=255 and 0<=y<=175
RND	none	The next pseudorandom number in a sequence generated by taking the powers of 75 modulo 65537, subtracting 1 and dividing by 65536. 0≤y<1
SCREEN$	Two arguments, x and y, both numbers; enclosed in brackets	The character that appears, either normally or inverted, on the television at line x, column y. Gives the empty string, if the character is not

Function	*Type of argument*	*Result*
		recognised. Error B unless 0<=x<=23 and 0<=y<=31
SGN	number	Signum: the sign (−1 for negative, 0 for zero or +1 for positive) of x
SIN	number (in radians)	Sine x
SQR	number	Square root. Error A if x<0
STR$	number	The string of characters that would be displayed if x were printed
TAN	number (in radians)	Tangent
USR	number	Calls the machine code subroutine whose starting address is x. On return, the result is the contents of the bc register pair
USR	string	The address of the bit pattern for the user-defined graphic corresponding to x. Error A if x is not a single letter between a and u, or a user-defined graphic
VAL	string	Evaluates x (without its bounding quotes) as a numerical expression. Error C if x contains a syntax error, or gives a string value. Other errors possible, depending on the expression
VAL$	string	Evaluates x (without its bounding quotes) as a string expression. Error C if x contains a syntax error or gives a numeric value. Other errors possible, as for **VAL**
−	number	Negation

The following are binary operations:

+	Addition (on numbers), or concatenation (on strings)	
−	Subtraction	
*	Multiplication	
/	Division	
↑	Raising to a power. Error B if the left operand is negative	
=	Equals	Both operands must be of the same type. The result is a number 1, if the comparison holds and 0 if it does not
>	Greater than	
<	Less than	
<=	Less than or equal to	
>=	Greater than or equal to	
<>	Not equal to	

Functions and operations have the following priorities:

Operation	*Priority*
Subscripting and slicing	12
All functions except **NOT** and unary minus	11
↑	10
Unary minus (i.e. minus just used to negate something)	9
*, /	8
+, − (minus used to subtract one number from another)	6
=, >, <, <=, >=, <>	5
NOT	4
AND	3
OR	2

Statements

In this list,

α	represents a single letter
v	represents a variable
x, y, z	represent numerical expressions
m, n	represent numerical expressions that are rounded to the nearest integer
e	represents an expression
f	represents a string valued expression
s	represents a sequence of statements separated by colons :
c	represents a sequence of colour items, each terminated by commas , or semi-colons ;. A colour item has the form of a **PAPER**, **INK**, **FLASH**, **BRIGHT**, **INVERSE** or **OVER** statement.

Note that arbitrary expressions are allowed everywhere (except for the line number at the beginning of a statement).

All statements except **INPUT**, **DEF** and **DATA** can be used either as commands or in programs (although they be more sensible in one than the other). A command or program line can have several statements, separated by colons (:). There is no restriction on whereabouts in a line any particular statement can occur – although see **IF** and **REM**.

BEEP x, y — Sounds a note through the loudspeaker for x seconds at a pitch y semitones above middle C (or below if y is negative)

BORDER m — Sets the colour of the border of the screen and also the paper colour for the lower part of the screen.
Error K if m not in the range 0 to 7

BRIGHT Sets brightness of characters subsequently printed. n=0 for normal, 1 for bright, 8 for transparent.
Error K if n not 0, 1 or 8

CAT Does not work without Microdrive, etc

CIRCLE x, y, z Draws an arc of a circle, centre (x,y), radius z

CLEAR Deletes all variables, freeing the space they occupied.
Does **RESTORE** and **CLS**, resets the **PLOT** position to the bottom left-hand corner and clears the **GO SUB** stack

CLEAR n Like **CLEAR**, but if possible changes the system variable **RAMTOP** to n and puts the new **GO SUB** stack there

CLOSE # Does not work without Microdrive, etc

CLS (Clear Screen). Clears the display file

CONTINUE Continues the program, starting where it left off last time it stopped with report other than 0. If the report was 9 or L, then continues with the following statement (taking jumps into account); otherwise repeats the one where the error occurred.
If the last report was in a command line then **CONTINUE** will attempt to continue the command line and will either go into a loop if the error was in 0:1, give report 0 if it was in 0:2, or give error N if it was 0:3 or greater.
CONTINUE appears as **CONT** on the keyboard

COPY Sends a copy of the top 22 lines of display to the printer, if attached; otherwise does nothing. Note that **COPY** can not be used to print the automatic listings that appear on the screen.
Report D if **BREAK** pressed

DATA e_1, e_2, e_3, ... Part of the **DATA** list. Must be in a program

DEF FN $\alpha(\alpha_1,...,\alpha_k)=e$ User-defined function definition; must be in a program. Each of α and α_1 to α_k is either a single letter or a single letter followed by '$' for string argument or result.
Takes the form **DEF FN** $\alpha()=e$ if no arguments

DELETE f Does not work without Microdrive, etc

DIM $\alpha(n_1,...,n_k)$ Deletes any array with the name α, and sets up an array α of numbers with k dimensions $n_1,...,n_k$. Initialises all the values to 0

DIM $\alpha\$(n_1,...,n_k)$ Deletes any array or string with the name $\alpha\$$, and sets

up an array of characters with k dimensions $n_1,...,n_k$. Initialises all the values to " ". This can be considered as an array of strings of fixed length n_k, with $k-1$ dimensions $n_1,...,n_{k-1}$.

Error 4 occurs if there is no room to fit the array in. An array is undefined until it is dimensioned in a **DIM** statement

DRAW x,y — **DRAW** x,y,0

DRAW x,y,z — Draws a line from the current plot position moving x horizontally and y vertically relative to it while turning through an angle z.

Error B if it runs off the screen

ERASE — Does not work without Microdrive, etc.

FLASH — Defines whether characters will be flashing or steady. n=0 for steady, n=1 for flash, n=8 for no change.

FOR α=x **TO** y — **FOR** α=x **TO** y **STEP** 1

FOR α=x **TO** y **STEP** z — Deletes any simple variable α and sets up a control variable with value x, limit y, step z, and looping address referring to the statement after the **FOR** statement. Checks if the initial value is greater (if step>=0) or less (if step<0) than the limit, and if so then skips to statement **NEXT** α, giving error 1 if there is none. See **NEXT**.

Error 4 occurs if there is no room for the control variable

FORMAT f — Does not work without the Microdrive, etc

GOSUB n — Pushes the line number of the **GOSUB** statement onto a stack; then as **GO TO** n.

Error 4 can occur if there are not enough **RETURN**s

GO TO n — Jumps to line n (or, if there is none, the first line after that)

IF x **THEN** s — If x true (non-zero) then s is executed. Note that s comprises all the statements to the end of the line. The form '**IF** x **THEN** line number' is not allowed

INK n — Sets the ink (foreground) colour of characters subsequently printed. n is in the range 0 to 7 for a colour, n=8 for transparent or 9 for contrast. See *The television screen* – Appendix B.

Error K if n not in the range 0 to 9

INPUT ... — The '...' is a sequence of **INPUT** items, separated as in a **PRINT** statement by commas, semicolons or apostrophes. An **INPUT** item can be

(i) Any **PRINT** item not beginning with a letter
(ii) A variable name, or
(iii) **LINE**, then a string type variable name.

The **PRINT** items and separators in (i) are treated exactly as in **PRINT**, except that everything is printed in the lower part of the screen.

For (ii) the computer stops and waits for input of an expression from the keyboard; the value of this is assigned to the variable. The input is echoed in the usual way and syntax errors give the flashing ?. For string type expressions, the input buffer is initialised to contain two string quotes (which can be erased if necessary). If the first character in the input is **STOP**, the program stops with error H. (iii) is like (ii) except that the input is treated as a string literal without quotes, and the **STOP** mechanism doesn't work; to stop it you must type ◆ instead

INVERSE n — Controls inversion of characters subsequently printed. If n=0, characters are printed in *normal video*, as ink colour on paper colour.

If n=1, characters are printed in *inverse video*, i.e. paper colour on ink colour. See *The television screen* – Appendix B.

Error K if n is not 0 or 1

LET v=e — Assigns the value of e to the variable v. **LET** cannot be omitted. A simple variable is undefined until it is assigned to in a **LET**, **READ** or **INPUT** statement. If v is a subscripted string variable, or a sliced string variable (substring), then the assignment is *Procrustean* (fixed length): the string value of e is either truncated or filled out with spaces on the right, to make it the same length as the variable v

LIST — **LIST 0**

LIST n — Lists the program to the upper part of the screen, starting at the first line whose number is at least n, and makes n the current line

LLIST — **LLIST 0**

LLIST n — Like **LIST**, but using the printer

LOAD f — Loads program and variables

LOAD f **DATA** () — Loads a numeric array

LOAD f **DATA** $() — Loads character array $

LOAD f **CODE** m,n — Loads at most n bytes, starting at address m

LOAD f **CODE** m
Loads bytes starting at address m

LOAD f **CODE**
Loads bytes back to the address they were saved from

LOAD f **SCREEN$**
LOAD f **CODE** 16384,6912.

Searches for file of the right sort on cassette tape and loads it, deleting previous versions in memory. See Chapter 20

LPRINT
Like **PRINT** but using the printer

MERGE f
Like **LOAD** f, but does not delete old program lines and variables except to make way for new ones with the same line number or name

MOVE f_1,f_2
Does not work without the Microdrive, etc

NEW
Starts the BASIC system off anew, deleting program and variables, and using the memory up to and including the byte whose address is in the system variable RAMBOT and preserves the system variables UDG, P RAMT, RASP and PIP

NEXT α
(i) Finds the control variable α
(ii) Adds its step to its value
(iii) If the step>=0 and the value>the limit; or if the step<0 and the value<the limit, then jumps to the looping statement.

Error 2 if there is no variable α.
Error 1 if there is one, but it's not α control variable

OPEN #
Does not work without the Microdrive, etc

OUT m,n
Outputs byte n at port m at the processor level. (Loads the bc register pair with m, the a register with n, and does the assembly language instruction: out (c),a.)
0<=m<=65535, −255<=n<=255, else error B

OVER n
Controls overprinting for characters subsequently printed.

If n=0, characters obliterate previous characters at that position.

If n=1, then new characters are mixed in with old characters to give ink colour wherever either (but not both) had ink colour, and paper colour if they were both paper or both ink colour. See *The television screen* – Appendix B.

Error K if n not 0 or 1

PAPER n
Like **INK**, but controlling the paper (background) colour

PAUSE n
Stops computing and displays the display file for n frames (at 50 frames per second or 60 frames per second in North America) or until a key is pressed.

0<=n<=65535, else error B. If n=0 then the pause is not timed, but lasts until a key is pressed

PLOT c;m,n

Prints an ink spot (subject to **OVER** and **INVERSE**) at the pixel (|m|, |n|); moves the **PLOT** position.

Unless the colour items c specify otherwise, the ink colour at the character position containing the pixel is changed to the current permanent ink colour, and the other (paper colour, flashing and brightness) are left unchanged.

0<=|m|<=255, 0<=|n|<=175, else error B

POKE m,n

Writes the value n to the byte in store with address m.

0<=m<=65535, −255<=n<=255, else error B

PRINT ...

The '...' is a sequence of **PRINT** items, separated by commas **,**, semicolons **;** or apostrophes **'** and they are written to the display file for output to the television.

A semicolon **;** between two items has no effect: it is used purely to separate the items. A comma **,** outputs the comma control character, and an apostrophe **'** outputs the **ENTER** character.

At the end of the **PRINT** statement, if it does not end in a semicolon, or comma, or apostrophe, an **ENTER** character is output.

A **PRINT** item can be

(i) empty, i.e. nothing.

(ii) a numerical expression

First a minus sign is printed if the value is negative. Now let x be the modulus of value.

If $x<=10^{-5}$ or $x>=10^{13}$, then it is printed using scientific notation. The mantissa part has up to eight digits (with no trailing zeros), and the decimal point (absent if only one digit) is after the first. The exponent part is E, followed by + or −, followed by one or two digits.

Otherwise x is printed in ordinary decimal notation with up to eight significant digits, and no trailing zeros after the decimal point. A decimal point right at the beginning is always followed by a zero, so for instance .03 and 0.3 are printed as such.

0 is printed as a single digit 0.

(iii) a string expression

The tokens in the string are expanded, possibly with a space before or after.

Control characters have their control effect.

Unrecognized characters print as **?**.

(iv) **AT** m,n

Outputs an **AT** control character followed by a byte for m (the line number) and a byte for n (the column number).

(v) **TAB** n

Outputs a **TAB** control character followed by two bytes for n (less significant byte first), the **TAB** stop.

(vi) A colour item, which takes the form of a **PAPER**, **INK**, **FLASH**, **BRIGHT**, **INVERSE** or **OVER** statement

RANDOMIZE

RANDOMIZE 0

RANDOMIZE n

Sets the system variable (called SEED) used to generate the next value of **RND**. If n<>0, SEED is given the value n; if n=0 then it is given the value of another system variable (called FRAMES) that counts the frames so far displayed on the television, and so should be fairly random.

RANDOMIZE appears as **RAND** on the keyboard.

Error B occurs if n is not in the range 0 to 65535

READ v_1, v_2, ... v_k

Assigns to the variables using successive expressions in the **DATA** list.

Error C if an expression is the wrong type.

Error E if there are variables left to be read when the **DATA** list is exhausted

REM ...

No effect. '...' can be any sequence of characters except **ENTER**. This can include :, so no statements are possible after the **REM** statement on the same line

RESTORE

RESTORE 0

RESTORE n

Restores the **DATA** pointer to the first **DATA** statement in a line with number at least n: the next **READ** statement will start reading there

RETURN

Takes a reference to a statement off the **GO SUB** stack, and jumps to the line after it.

Error 7 occurs when there is no statement reference on the stack. There is some mistake in your program; **GO SUB**s are not properly balanced by **RETURN**s

RUN

RUN 0

RUN n

CLEAR, and then **GO TO** n

SAVE f

Saves the program and variables

SAVE f **LINE** m
Saves the program and variables so that if they are loaded there is an automatic jump to line m

SAVE f **DATA** ()
Saves the numeric array

SAVE f **DATA** $()
Saves the character array $

SAVE f **CODE** m,n
Saves n bytes starting at address m

SAVE f **SCREEN$**
SAVE f **CODE** 16384,6912.
Saves information on cassette, giving it the name f.
Error F if f is empty or has length eleven or more.
See Chapter 20

STOP
Stops the program with report 9. **CONTINUE** will resume with the following statement

VERIFY
The same as **LOAD** except that the data is not loaded into RAM, but compared against what is already there.
Error R if one of the comparisons shows different bytes

Example programs

This appendix contains some example programs to demonstrate the abilities of the ZX Spectrum.

The first of these programs requires a date to be input and gives the day of the week which corresponds to this date.

```
 10 REM convert date to day
 20 DIM d$(7,6): REM days of week
 30 FOR n=1 TO 7: READ d$(n): NEXT n
 40 DIM m(12): REM lengths of months
 50 FOR n=1 TO 12: READ m(n): NEXT n
100 REM input date
110 INPUT "day?";day
120 INPUT "month?";month
130 INPUT "year (20th century only)?";year
140 IF year<1901 THEN PRINT "20th century starts at 1901":
    GO TO 100
150 IF year>2000 THEN PRINT "20th century ends at 2000":
    GO TO 100
160 IF month<1 THEN GO TO 210
170 IF month>12 THEN GO TO 210
180 IF year/4-INT(year/4)=0 THEN LET m(2)=29: REM leap year
190 IF day>m(month) THEN PRINT "This month has only ";
    m(month);" days.": GO TO 500
200 IF day>0 THEN GO TO 300
210 PRINT "Stuff and nonsense. Give me a real date."
220 GO TO 500
300 REM convert date to number of days since start of century
310 LET y=year-1901
320 LET b=365*y+INT (y/4): REM number of days to start of year
330 FOR n=1 TO month-1: REM add on previous months
340 LET b=b+m(n): NEXT n
350 LET b=b+day
400 REM convert to day of week
410 LET b=b-7* INT (b/7)+1
420 PRINT day;"/";month;"/";year
430 FOR n=6 TO 3 STEP -1: REM remove trailing spaces
440 IF d$(b,n) <> " " THEN GO TO 460
450 NEXT n
460 LET e$=d$(b, TO n)
470 PRINT" is a "; e$; "day"
500 LET m(2)=28: REM restore February
510 INPUT "again?", a$
520 IF a$="n" THEN GO TO 540
```

```
530 IF a$ <> "N" THEN GO TO 100
1000 REM days of week
1010 DATA "Mon", "Tues", "Wednes"
1020 DATA "Thurs", "Fri", "Satur", "Sun"
1100 REM lengths of months
1110 DATA 31, 28, 31, 30, 31, 30
1120 DATA 31, 31, 30, 31, 30, 31
```

This program handles yards, feet and inches.

```
10 INPUT "yards?",yd,"feet?",ft, "inches?",in
40 GO SUB 2000: REM print the values
50 PRINT '"   =   ";
70 GO SUB 1000: REM the adjustment
80 GO SUB 2000: REM print the adjusted values
90 PRINT
100 GO TO 10
1000 REM subroutine to adjust yd, ft, in to the normal form for
     yards, feet and inches
1010 LET in=36*yd+12*ft+in: REM now everything is in inches
1030 LET s=SGN in: LET in=ABS in: REM we work with in
     positive, holding its sign in s
1060 LET ft=INT (in/12): LET in=(in-12*ft)*s: REM now in is ok
1080 LET yd=INT (ft/3)*s: LET ft=ft*s-3*yd: RETURN
2000 REM subroutine to print yd, ft and in
2010 PRINT yd;"yd";ft;"ft";in;"in";: RETURN
```

Here is a program to throw coins for the I Ching. (Unfortunately it produces the patterns upside down, but you might not worry about this.)

```
5 RANDOMIZE
10 FOR m=1 TO 6: REM for 6 throws
20 LET c=0: REM initialize coin total to 0
30 FOR n=1 TO 3: REM for 3 coins
40 LET c=c+2+INT (2*RND)
50 NEXT n
60 PRINT "    ";
70 FOR n=1 TO 2: REM 1st for the thrown hexagram, 2nd for
   the changes
80 PRINT "---";
90 IF c=7 THEN PRINT "-";
100 IF c=8 THEN PRINT " ";
110 IF c=6 THEN PRINT "X";: LET c=7
120 IF c=9 THEN PRINT "0";: LET c=8
130 PRINT "---  ";
140 NEXT n
150 PRINT
160 INPUT a$
170 NEXT m: NEW
```

To use this, type it in and run it, and then press **ENTER** five times to get the two hexagrams. Look these up in a copy of the Chinese Book of Changes. The text will describe a situation and the courses of action appropriate to it, and you must ponder deeply to discover the parallels between that and your own life. Press **ENTER** a sixth time, and the program will erase itself – this is to discourage you from using it frivolously.

Many people find the texts are always more apt than they would expect on grounds of chance; this may or may not be the case with your ZX Spectrum. In general, computers are pretty godless creatures.

Here is a program to play 'Pangolins'. You think up an animal, and the computer tries to guess what it is, by asking you questions that can be answered 'yes' or 'no'. If it's never heard of your animal before, it asks you to give it some question that it can use next time to find out whether someone's given it your new animal.

```
  5 REM pangolins
 10 LET nq=100: REM number of questions and animals
 15 DIM q$(nq,50): DIM a(nq,2): DIM r$(1)
 20 LET qf=8
 30 FOR n=1 TO qf/2-1
 40 READ q$(n): READ a(n,1): READ a(n,2)
 50 NEXT n
 60 FOR n=n TO qf-1
 70 READ q$(n): NEXT n

100 REM start playing
110 PRINT "Think of an animal.","Press any key to continue."
120 PAUSE 0
130 LET c=1: REM start with 1st question
140 IF a(c,1)=0 THEN GO TO 300
150 LET p$=q$(c): GO SUB 910
160 PRINT "?": GO SUB 1000
170 LET in=1: IF r$="y" THEN GO TO 210
180 IF r$="Y" THEN GO TO 210
190 LET in=2: IF r$="n" THEN GO TO 210
200 IF r$<>"N" THEN GO TO 150
210 LET c=a(c,in): GO TO 140

300 REM animal
310 PRINT "Are you thinking of"
320 LET p$=q$(c): GO SUB 900: PRINT "?"
330 GO SUB 1000
340 IF r$="y" THEN GO TO 400
350 IF r$="Y" THEN GO TO 400
360 IF r$="n" THEN GO TO 500
370 IF r$="N" THEN GO TO 500
```

```
380 PRINT "Answer me properly when I'm","talking to you.": GO
    TO 300

400 REM guessed it
410 PRINT "I thought as much.": GO TO 800

500 REM new animal
510 IF qf>nq-1 THEN PRINT "I'm sure your animal is very",
    "interesting, but I don't have","room for it just now.": GO TO 800
520 LET q$(qf)=q$(c): REM move old animal
530 PRINT "What is it, then?": INPUT q$(qf+1)
540 PRINT "Tell me a question which dist-","inguishes
    between "
550 LET p$=q$(qf): GO SUB 900: PRINT " and"
560 LET p$=q$(qf+1): GO SUB 900: PRINT " "
570 INPUT s$: LET b=LEN s$
580 IF s$(b)="?" THEN LET b=b-1
590 LET q$(c)=s$(TO b): REM insert question
600 PRINT "What is the answer for"
610 LET p$=q$(qf+1): GO SUB 900: PRINT "?"
620 GO SUB 1000
630 LET in=1: LET io=2: REM answers for new and old animals
640 IF r$="y" THEN GO TO 700
650 IF r$="Y" THEN GO TO 700
660 LET in=2: LET io=1
670 IF r$="n" THEN GO TO 700
680 IF r$="N" THEN GO TO 700
690 PRINT "That's no good. ": GO TO 600

700 REM update answers
710 LET a(c,in)=qf+1: LET a(c,io)=qf
720 LET qf=qf+2: REM next free animal space
730 PRINT "That fooled me."

800 REM again?
810 PRINT "Do you want another go?": GO SUB 1000
820 IF r$="y" THEN GO TO 100
830 IF r$="Y" THEN GO TO 100
840 STOP

900 REM print without trailing spaces
905 PRINT " ";
910 FOR n=50 TO 1 STEP -1
920 IF p$(n)<>" " THEN GO TO 940
```

```
930 NEXT n
940 PRINT p$(TO n);: RETURN

1000 REM get reply
1010 INPUT r$: IF r$="" THEN RETURN
1020 LET r$=r$(1): RETURN

2000 REM initial animals
2010 DATA "Does it live in the sea",4,2
2020 DATA "Is it scaly",3,5
2030 DATA "Does it eat ants",6,7
2040 DATA "a whale", "a blancmange", "a pangolin", "an ant"
```

Here is a program to draw a Union Flag.

```
  5 REM union flag
 10 LET r=2: LET w=7: LET b=1
 20 BORDER 0: PAPER b: INK w: CLS
 30 REM black in bottom of screen
 40 INVERSE 1
 50 FOR n=40 TO 0 STEP -8
 60 PLOT PAPER 0;7,n: DRAW PAPER 0;241,0
 70 NEXT n: INVERSE 0
100 REM draw in white parts
105 REM St. George
110 FOR n=0 TO 7
120 PLOT 104+n,175: DRAW 0,-35
130 PLOT 151-n,175: DRAW 0,-35
140 PLOT 151-n,48: DRAW 0,35
150 PLOT 104+n,48: DRAW 0,35
160 NEXT n
200 FOR n=0 TO 11
210 PLOT 0,139-n: DRAW 111,0
220 PLOT 255,139-n: DRAW -111,0
230 PLOT 255,84+n: DRAW -111,0
240 PLOT 0,84+n: DRAW 111,0
250 NEXT n
300 REM St. Andrew
310 FOR n=0 TO 35
320 PLOT 1+2*n,175-n: DRAW 32,0
330 PLOT 224-2*n,175-n: DRAW 16,0
340 PLOT 254-2*n,48+n: DRAW -32,0
350 PLOT 17+2*n,48+n: DRAW 16,0
360 NEXT n
370 FOR n=0 TO 19
```

```
380 PLOT 185+2*n,140+n: DRAW 32,0
390 PLOT 200+2*n,83-n: DRAW 16,0
400 PLOT 39-2*n,83-n: DRAW 32,0
410 PLOT 54-2*n,140+n: DRAW -16,0
420 NEXT n
425 REM fill in extra bits
430 FOR n=0 TO 15
440 PLOT 255,160+n: DRAW 2*n-30,0
450 PLOT 0,63-n: DRAW 31-2*n,0
460 NEXT n
470 FOR n=0 TO 7
480 PLOT 0,160+n: DRAW 14-2*n,0
485 PLOT 255,63-n: DRAW 2*n-15,0
490 NEXT n
500 REM red stripes
510 INVERSE 1
520 REM St. George
530 FOR n=96 TO 120 STEP 8
540 PLOT PAPER r;7,n: DRAW PAPER r;241,0
550 NEXT n
560 FOR n=112 TO 136 STEP 8
570 PLOT PAPER r;n,168: DRAW PAPER r;0,-113
580 NEXT n
600 REM St. Patrick
610 PLOT PAPER r;170,140: DRAW PAPER r;70,35
620 PLOT PAPER r;179,140: DRAW PAPER r;70,35
630 PLOT PAPER r;199,83: DRAW PAPER r;56,-28
640 PLOT PAPER r;184,83: DRAW PAPER r;70,-35
650 PLOT PAPER r;86,83: DRAW PAPER r;-70,-35
660 PLOT PAPER r;72,83: DRAW PAPER r;-70,-35
670 PLOT PAPER r;56,140: DRAW PAPER r;-56,28
680 PLOT PAPER r;71,140: DRAW PAPER r;-70,35
690 INVERSE 0: PAPER 0: INK 7
```

If you're not British, have a go at drawing your own flag. Tricolours are fairly easy, although some of the colours – for instance the orange in the Irish flag – might present difficulties. If you're an American, you might be able to fit the character * in.

Here is a program to play hangman. One player enters a word, and the other guesses.

```
  5 REM Hangman
 10 REM set up screen
 20 INK 0: PAPER 7: CLS
 30 LET x=240: GO SUB 1000: REM draw man
```

```
 40 PLOT 238,128: DRAW 4,0: REM mouth
100 REM set up word
110 INPUT w$: REM word to guess
120 LET b=LEN w$: LET v$=" "
130 FOR n=2 TO b: LET v$=v$+" "
140 NEXT n: REM v$=word guessed so far
150 LET c=0: LET d=0: REM guess & mistake counts
160 FOR n=0 TO b-1
170 PRINT AT 20,n;"-";
180 NEXT n: REM write -'s instead of letters
200 INPUT "Guess a letter: ";g$
210 IF g$="" THEN GO TO 200
220 LET g$=g$(1): REM 1st letter only
230 PRINT AT 0,c;g$
240 LET c=c+1: LET u$=v$
250 FOR n=1 TO b: REM update guessed word
260 IF w$(n)=g$ THEN LET v$(n)=g$
270 NEXT n
280 PRINT AT 19,0;v$
290 IF v$=w$ THEN GO TO 500: REM word guessed
300 IF v$<>u$ THEN GO TO 200: REM guess was right
400 REM draw next part of gallows
410 IF d=8 THEN GO TO 600: REM hanged
420 LET d=d+1
430 READ x0,y0,x,y
440 PLOT x0,y0: DRAW x,y
450 GO TO 200
500 REM free man
510 OVER 1: REM rub out man
520 LET x=240: GO SUB 1000
530 PLOT 238,128: DRAW 4,0: REM mouth
540 OVER 0: REM redraw man
550 LET x=146: GO SUB 1000
560 PLOT 143,129: DRAW 6,0, PI/2: REM smile
570 GO TO 800
600 REM hang man
610 OVER 1: REM rub out floor
620 PLOT 255,65: DRAW -48,0
630 DRAW 0,-48: REM open trapdoor
640 PLOT 238,128: DRAW 4,0: REM rub out mouth
650 REM move limbs
655 REM arms
660 PLOT 255,117: DRAW -15,-15: DRAW -15,15
670 OVER 0
680 PLOT 236,81: DRAW 4,21: DRAW 4,-21
```

```
690 OVER 1: REM legs
700 PLOT 255,66: DRAW -15,15: DRAW -15,-15
710 OVER 0
720 PLOT 236,60: DRAW 4,21: DRAW 4,-21
730 PLOT 237,127: DRAW 6,0, -PI/2: REM frown
740 PRINT AT 19,0;w$
800 INPUT "again? ";a$
810 IF a$="" THEN GO TO 850
820 LET a$=a$(1)
830 IF a$="n" THEN STOP
840 IF a$(1)="N" THEN STOP
850 RESTORE : GO TO 5
1000 REM draw man at column x
1010 REM head
1020 CIRCLE x,132,8
1030 PLOT x+4,134: PLOT x-4,134: PLOT x,131
1040 REM body
1050 PLOT x,123: DRAW 0,-20
1055 PLOT x,101: DRAW 0,-19
1060 REM legs
1070 PLOT x-15,66: DRAW 15,15: DRAW 15,-15
1080 REM arms
1090 PLOT x-15,117: DRAW 15,-15: DRAW 15,15
1100 RETURN
2000 DATA 120,65,135,0,184,65,0,91
2010 DATA 168,65,16,16,184,81,16,-16
2020 DATA 184,156,68,0,184,140,16,16
2030 DATA 204,156,-20,-20,240,156,0,-16
```

Binary and hexadecimal

This appendix describes how computers count, using the binary system.

Most European languages count using a more or less regular pattern of tens – in English, for example, although it starts off a bit erratically, it soon settles down into regular groups:

twenty, twenty one, twenty two,.....twenty nine
thirty, thirty one, thirty two,.....thirty nine
forty, forty one, forty two,.....forty nine

and so on, and this is made even more systematic with the Arabic numerals that we use. However, the only reason for using ten is that we happen to have ten fingers and thumbs.

Instead of using the *decimal* system, with ten as its base, computers use a form of binary called *hexadecimal* (or *hex*, for short), based on sixteen. As there are only ten digits available in our number system we need six extra digits to do the counting. So we use A, B, C, D, E and F. And what comes after F? Just as we, with ten fingers, write 10 for ten, so computers write 10 for sixteen. Their number system starts off:

Hex	*English*
0	nought
1	one
2	two
:	:
:	:
9	nine

just as ours does, but then it carries on

A	ten
B	eleven
C	twelve
D	thirteen
E	fourteen
F	fifteen
10	sixteen
11	seventeen
:	:
:	:
19	twenty five
1A	twenty six
1B	twenty seven
:	:
:	:
1F	thirty one
20	thirty two

21	thirty three
:	:
:	:
9E	one hundred and fifty eight
9F	one hundred and fifty nine
A0	one hundred and sixty
A1	one hundred and sixty one
:	:
B4	one hundred and eighty
:	:
FE	two hundred and fifty four
FF	two hundred and fifty five
100	two hundred and fifty six

If you are using hex notation and you want to make the fact quite plain, then write 'h' at the end of the number, and say 'hex'. For instance, for one hundred and fifty eight, write '9Eh' and say 'nine E hex'.

You will be wondering what all this has to do with computers. In fact, computers behave as though they had only two digits, represented by a low voltage, or off (0), and a high voltage, or on (1). This is called the *binary* system, and the two binary digits are called *bits*: so a bit is either 0 or 1.

In the various systems, counting starts off

English	*Decimal*	*Hexadecimal*	*Binary*
nought	0	0	0 or 0000
one	1	1	1 or 0001
two	2	2	10 or 0010
three	3	3	11 or 0011
four	4	4	100 or 0100
five	5	5	101 or 0101
six	6	6	110 or 0110
seven	7	7	111 or 0111
eight	8	8	1000
nine	9	9	1001
ten	10	A	1010
eleven	11	B	1011
twelve	12	C	1100
thirteen	13	D	1101
fourteen	14	E	1110
fifteen	15	F	1111
sixteen	16	10	10000

The important point is that sixteen is equal to two raised to the fourth power, and this makes converting between hex and binary very easy.

To convert hex to binary, change each hex digit into four bits, using the table above.

To convert binary to hex, divide the binary number into groups of four bits, starting on the right, and then change each group into the corresponding hex digit.

For this reason, although strictly speaking computers use a pure binary system, humans often write the numbers stored inside a computer using hex notation.

The bits inside the computer are mostly grouped into sets of eight, or *bytes*. A single byte can represent any number from nought to two hundred and fifty five (11111111 binary or FF hex), or alternatively any character in the ZX Spectrum character set. Its value can be written with two hex digits.

Two bytes can be grouped together to make what is technically called a word. A word can be written using sixteen bits or four hex digits, and represents a number from 0 to (in decimal) $2^{16}-1=65535$.

A byte is always eight bits, but words vary in length from computer to computer.

The **BIN** notation in Chapter 14 provides a means of writing numbers in binary on the ZX Spectrum: '**BIN** 0' represents nought, '**BIN** 1' represents one, '**BIN** 10' represents two, and so on.

You can only use 0's and 1's for this, so the number must be a non-negative whole number; for instance you can't write '**BIN** −11' for minus three – you must write '−**BIN** 11' instead. The number must also be no greater than decimal 65535 – i.e. it can't have more than sixteen bits.

ATTR really was binary. If you convert the result from **ATTR** into binary, you can write it in eight bits.

The first is 1 for flashing, 0 for steady.
The second is 1 for bright, 0 for normal.
The next three are the code for the paper colour, written in binary.
The last three are the code for the ink colour, written in binary.

The colour codes also use binary: each code written in binary can be written in three bits, the first for green, the second for red and the third for blue.

Black has no light at all, so all the bits are 0 (off). Therefore the code for black is 000 in binary, or nought.

The pure colours, green, red and blue have just one bit 1 (on) out of the three. Their codes are 100, 010 and 001 in binary, or four, two and one.

The other colours are mixtures of these, so their codes in binary have two or more bits 1.

Index

This index includes the keys on the keyboard and how to obtain them (the mode – **K**, **L**, **E**, **C** or **G** – and which shift key where appropriate).

Usually an entry is referenced only once per chapter, so having found one reference, look through the rest of the chapter including the exercises.

C

D

E

F

G

H

M

N

Q

R

S

Printed by
The Leagrave Press Ltd
Luton and London

Sinclair Research Limited
6 King's Parade,
Cambridge CB2 1SN
England