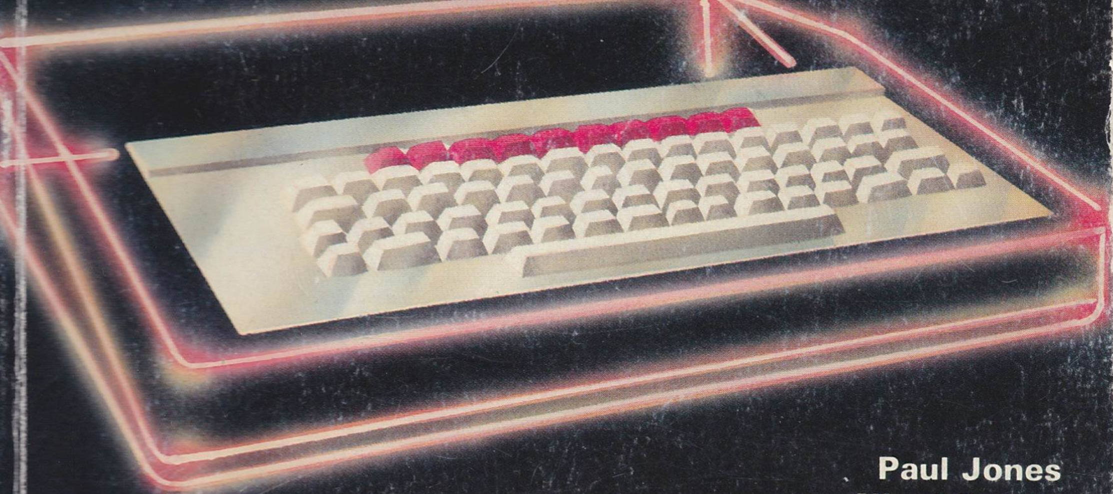
THE MICRO USER

GETTING STARTED IN sound & graphics on the BBC Micro



Paul Jones
Nigel Peters
Michael Noels

DATABASE PUBLICATIONS

Getting started in Sound and Graphics

Paul Jones
Nigel Peters
Michael Noels

DATABASE PUBLICATIONS

First published 1985 by:

Database Publications Ltd.
68 Chester Road
Hazel Grove
Stockport SK7 5NY.

ISBN number 0.948104 00 7

Dedicated to the memory of
JOHN GREEN – the first
man to put our ideas into print.

CONTENTS

Chapter 1: First Steps

THE BBC Micro has eight different modes, Mode 0 to 7. The mode you are in effectively decides how many characters you can display on the screen and in how many colours. Now when a micro is designed memory has to be allocated carefully. The more characters you have per line (cpl) the more memory it takes. Similarly, the more colours you use, the more memory it takes to remember them.

Because of the limited amount of memory available, a compromise has often to be made between the number of characters per line and the number of colours available to the display – and, as often happens with a compromise, you get the worst of both worlds.

In this, as in so many areas, the BBC Micro offers options that the others do not. By choosing between the various modes you can trade off the number of colours available against the number of characters displayed and so, hopefully, obtain the optimum for a particular need. Figure I displays the characteristics of the various modes.

Before going further let's "dispose" of Mode 7, the teletext mode, which is the one the computer is in when first switched on. This mode is organised in a totally different way to the others, so, for the major part of this book, it is "out

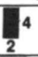
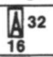
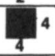
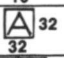
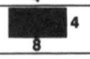
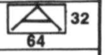
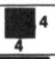
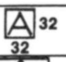
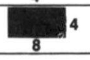
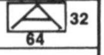
MODE	Graphics resolution		No. of colours	Text		Memory used	Smallest plotable point or pixel (gu)	Character size (gu)
	hor.	vert.		char.	lines			
Ø	64Ø	256	2	8Ø	32	2Øk		
1	32Ø	256	4	4Ø	32	2Øk		
2	16Ø	256	16	2Ø	32	2Øk		
3	text only		2	8Ø	25	16k		
4	32Ø	256	2	4Ø	32	1Øk		
5	16Ø	256	4	2Ø	32	1Øk		
6	text only		2	4Ø	25	8k		
7	text only		16	4Ø	25	1k		

Figure I: Modes available on the BBC Micro

of context".

We will however be devoting chapters eleven and twelve to Mode 7. It is very valuable, as an examination of Figure I discloses. It can display 40 cpl in 16 colours and yet only uses 1k of memory.

The only other mode to display 16 colours is Mode 2, and that only has 20 cpl and uses a massive 20k. The drawback is that teletext is not a graphics mode, i.e. you aren't able to use BBC Basic's advanced graphic commands. This does not mean that you cannot use teletext to draw some effective pictures – you can, as Ceefax and Oracle, which also use Mode 7 format, demonstrate.

For the moment, though, we'll look at modes 0 to 6. If we examine Figure I we can see clearly the cost, in terms of memory, of increasing either the number of colours or of characters. Mode 3 and Mode 6 are both text-only modes, which, as in Mode 7, means that the graphic commands are not available. Both support only two colours.

Notice that while Mode 6 supports only 40 cpl, Mode 3 supports 80 cpl. This doubling of characters, however, necessitates an increase in memory required by the screen from 8k to 16k. Similarly, if we look at the two four colour modes (i.e. Mode 1 and Mode 5) we find that Mode 5, which has only 20 cpl, uses 10k, while Mode 1, which supports double the characters (40 cpl) uses double the memory – 20k.

The two colour graphics modes, 0 and 4, show a similar pattern. This time, though, because you have fewer colours, Mode 4 can display 40 cpl using 10k of memory, while Mode 0 displays 80 cpl using 20k. From the Memory Used column of Figure I it can be seen that Modes 0, 1 and 2 need 20k for their display, while Mode 3 requires 16k, hence these modes cannot be used on the standard model A, which has only 16k of RAM fitted. The remaining modes (4, 5, 6, 7), which use less memory, are available on both models A and B.

From Figure I it can be seen that the modes fall into three categories of two, four and sixteen colours. You must remember, however, that one of the colours available is the background colour – for example, in a two colour mode you do not have two colours to write on, say, a black background. If you want a black background, that's one of your two colours, leaving you only one colour to write on it with.

Also to say that there are 16 colours available on the BBC Micro is slightly misleading. There are, in fact, only eight separate colours – black, red, green, yellow, blue, magenta, cyan and white. As an investigation of Figure II will show, the other colours are made up of pairs of these colours, flashing alternately.

Figure II also shows that, in the four colour modes (Mode 1 and Mode 5) the colours available are black, red, yellow and white. This is only the "default" situation on entering a mode. You can, if you choose, display any of the 16 colours available, but you are restricted to displaying a maximum of four of these colours on the screen at any one time.

Similarly, the two-colour modes have black and white as their colours on entry, but you can select any two colours from a "palette" containing 16 colours. You could, say, write in white on a blue background.

Enough theory – let's get some hands on experience. So switch on your computer. At power on, the machine is in Mode 7. To change mode you simply type MODE followed by the relevant number, e.g. to enter Mode 5 type: **MODE 5** [Return], where [Return] means "press the Return key". The first thing you'll notice is that the screen clears leaving the prompt and the flashing cursor at the top left-hand corner of the screen. You cannot change mode without clearing the screen.

Try typing in a few letters at random to see the chunky set of letters Mode 5 gives you – remember it has 20 cpl. Press Return, ignoring any error messages, and enter: **MODE 4** [Return]. Again, the screen should clear. Try typing a few letters this time. Immediately it should become apparent that the characters are of more normal proportions – Mode 4 is a 40 cpl mode. The penalty for having 40 cpl is that Mode 4 is only a two colour mode, while Mode 5 supports four colours. Both require 10k of memory.

The last of the modes available on an unexpanded model A is Mode 6. It may seem odd to have this special text-only mode with only 25 lines of 40 characters when there already exists Mode 4 which supports 32 lines of 40 characters. Part of the answer is memory saving: Mode 6 only uses 8k whereas Mode 4 requires 10k.

MODES 0, 3, 4, 6

Logical number		Colour
Fore-ground	Back-ground	(on entering mode)
0	128	Black
1	129	White

MODES 1, 5

Logical number		Colour
Fore-ground	Back-ground	(on entering mode)
0	128	Black
1	129	Red
2	130	Yellow
3	131	White

N.B. The foreground logical colour numbers on entering mode 2 are also the actual numbers.

MODE 2 (and actual colours)

Logical number		Colour
Fore-ground	Back-ground	(on entering mode)
0	128	Black
1	129	Red
2	130	Green
3	131	Yellow
4	132	Blue
5	133	Magenta
6	134	Cyan
7	135	White
8	136	Flashing black-white
9	137	Flashing red-cyan
10	138	Flashing green-magenta
11	139	Flashing yellow-blue
12	140	Flashing blue-yellow
13	141	Flashing magenta-green
14	142	Flashing cyan-red
15	143	Flashing white-black

Figure II

To see the rest try this in both modes – hold a key down so that the automatic repeat functions until you have a couple of lines filled with characters. You should be able to see that in Mode 6 the lines are separated slightly, making for greater legibility. If you are fortunate enough to have the extra memory you can experiment with the size of characters in the other modes (0, 1, 2 and 3).

Mode 2 gives 20 cpl as did Mode 5, but offers you 16 colours instead of four. Mode 1 offers you four colours, as did Mode 5, but with double the number of characters per line. Mode 0 offers you only two colours, but with 32 lines of 80 characters – while Mode 3, a text-only mode, provides 25 lines of 80 characters with better separation and using 4k less memory than Mode 0. (On many TV sets it will be rather hard to read the 80 character modes – monitors function better.)

Let's return to Mode 5 with **MODE 5** [Return]. This is a four colour mode. On entering such a mode the colours available are: 0 black; 1 red; 2 yellow; 3 white.

As you can see, each colour has a number associated with it – the logical colour number. *(It must be stressed that the colour associated with a logical colour number is not fixed – you can alter things so that logical colour number 2 refers, say, to blue. We shall see how to do this later in the article.)*

On entering any mode, if you type something, it will appear in white on a black background. The colour the characters appear in is called the foreground colour. In this instance, since we have just entered Mode 5, the foreground is in logical colour number 3, while the background is in logical colour number 0.

Enter, on a new line: **COLOUR 1** [Return]. Now type some letters – the characters will now appear in logical colour 1 which is red. The effect of COLOUR followed by a number is to change the foreground logical colour number to the number specified.

For example, **COLOUR 2** [Return] will make subsequent typing appear in logical colour 2 which is yellow, while **COLOUR 3** [Return] will restore the foreground colour to white.

You can, if you wish, try **COLOUR 0** [Return]. This changes the foreground colour to black (logical colour 0). However, as the background is also black, there is no contrast between the characters and their background – hence you cannot see what you are writing! To return to normality you must either type in a command to change the foreground colour very carefully (since you cannot see what you are doing) or, more drastically, press Break, which will return the screen to Mode 7.

Of course we can use black as our foreground colour provided we write on a contrasting background. To change the background colour of the characters we must again use COLOUR, but this time followed by the logical colour number *PLUS* 128. This addition of 128 tells the computer that the logical colour number combined with it refers to the background colour, e.g.

COLOUR 129 will cause subsequent characters to be printed out on a red background since 129=128+1 and logical colour number 1 is, at the moment, red.

Experiment with different fore and background combinations. Notice that when you change the background colour, only the backgrounds of subsequent characters are affected. Earlier characters retain their previous background colours. If, however, you type CLS after a change of background colour, the screen will be cleared to that background colour. So **COLOUR 130:CLS** [Return] will give you an empty yellow screen while **COLOUR 2:COLOUR 129:CLS** [Return] will give you a red screen on which text appears in yellow.

```
10 REM PROGRAM I
20 MODE 5
30 PRINT'"MODE 5"
40 COLOUR 1
50 PRINT'"THIS IS IN COLOUR 1"
60 COLOUR 2
70 PRINT'"THIS IS IN COLOUR 2"
80 COLOUR 3
90 PRINT'"THIS IS IN COLOUR 3"
```

That is enough new information for the moment, let's try a few programs to consolidate our knowledge. Program I should help you familiarise yourself with the text colours. Instead of simply running the program, try the alterations and additions suggested.

```
10 REM PROGRAM II
20 MODE 5
30 C%=0
40 REPEAT
50 C%=C%+1
60 ASTERISK$=ASTERISK$+"*"
70 COLOUR C% MOD3 +1
80 PRINT ASTERISK$
90 UNTIL C%=18
```

Programs II and III are quite pretty. If you are a beginner, see if you can follow what is going on. If you are past the beginning stage, but not yet an expert, try adapting the programs as suggested – you may find the problems posed rather testing.

Program I prints out three of the colours available for text on entering Mode 5. Actually there is another colour. Try adding:

100 COLOUR 0
110 PRINT "THIS IS IN COLOUR 0"

Can you explain, and recover from, the results of this? Also, try changing the background colour with any of the following line 25s:

25 COLOUR 129:CLS
25 COLOUR 130:CLS
25 COLOUR 131:CLS

What happens if you omit CLS?

Program II prints out a triangle of asterisks, each line of which is in a different colour. This is achieved by line 70. C% MOD 3 gives the remainder when you divide C% by 3; these remainders can only be 0, 1 and 2 (you cannot have remainders of 3 or over when you are dividing by 3). This means that C% MOD 3+1 will return the values 1, 2 and 3 since we are adding one to the remainder. The COLOUR statement of line 70 then uses these numbers to change the foreground colour.

As an exercise, see if you can alter the program so that it prints the colours of the lines out randomly. Use RND(3) – see the User Guide for an explanation of RND( ). Far more advanced, can you print out the triangle so that each asterisk within a line is randomly coloured? Finally, can you produce a red triangle, inside a yellow triangle, inside a white triangle of asterisks? These two problems require far more than a simple alteration of Program II.

```
10 REM PROGRAM III
20 MODE 5
30 FOR I%= 0 TO 19
40 FOR J%= 0 TO 30
50 COLOUR RND(3)+128
60 PRINTTAB(I%,J%)CHR$(32);
70 NEXT J%
80 NEXT I%
```

Program III prints out a nice tapestry, filling in each of the screen columns in turn, top to bottom, left to right. As a test, can you alter it so that it fills in each row in turn, left to right, top to bottom?

You might also like to try the following versions of line 40 – they're quite interesting. See if you can work out what is happening.

```
40 COLOUR (I%+J%) MOD 3 +129
40 COLOUR (I%*J%) MOD 3 +129
40 COLOUR (I%-J%) MOD 3 +130
40 COLOUR (2*I%+J%) MOD 3 +129
40 COLOUR (I%=J%) MOD 3 +129
```

So far in a four colour mode we have had available the colours black, red, yellow and white. As we have seen, the computer doesn't refer to them by name but by the logical colour numbers 0, 1, 2 and 3. These numbers "label" the colours and we can, if we wish, change the colours that the logical colour numbers refer to. All we have to do is to use the VDU 19 command to tell the

NUMBERS	ACTUAL COLOUR
0	black
1	red
2	green
3	yellow
4	blue
5	magenta
6	cyan
7	white
8	flashing black-white
9	flashing red-cyan
10	flashing green-magenta
11	flashing yellow-blue
12	flashing blue-yellow
13	flashing magenta-green
14	flashing cyan-red
15	flashing white-black

Figure III

computer that we wish to assign a different colour to a particular logical colour number.

This would be simplicity itself if the syntax were such that to assign blue to logical colour number 1 we would use **VDU 19,1,BLUE.** Unfortunately the computer refers to the colours available in its palette not by name but by number – each colour in the palette has assigned to it a fixed number, called the actual colour number *(see Figure III).* The colour number for blue is 4.

To tell the computer that henceforward logical colour number 1 is to be interpreted as actual colour 4 (blue) we use **VDU 19,1,4,0,0,0**. Those trailing zeros are necessary for future expansion of the graphics system, we are told. Note the format: VDU 19, logical colour reassigned, actual colour assigned, 0,0,0

So, to assign actual colour 2 to logical colour 2 in Mode 5 we would use **VDU 19,2,2,0,0,0**. *(Before we reassign logical colour 2 in Mode 5 it is yellow i.e. actual colour 3. The command above tells the computer to interpret logical colour 2 as actual colour 2, i.e. green.)*

Remember, logical colour numbers are variable in the sense that we can change the colours they "label". The number of logical colours available, however, is fixed by the mode and it is these logical colours that are referred to in such statements as COLOUR 1 or GCOL0,3 (more of which anon).

Actual colour numbers are fixed. That is, actual colour number 2 always refers to green. We use them in the VDU 19 command when assigning colours to a logical colour number.

If you find the trailing zeros a nuisance you can use the form **VDU 19, logical, actual;0;** The example above then becomes VDU 19,2,2;0; Make sure you do not omit the final semi-colon – it can cause disaster!

One point to note is that when you reassign a logical colour to a new actual colour everything on the screen that is written in that logical colour (even if it was written before the reassignment) will instantly appear in the new colour. As we shall see, this will allow us to produce quite dramatic effects.

Program IV shows the effects of writing a message in logical colour 1 and then using a loop to assign each of the actual colours in turn to logical colour 1.

```
 10 REM PROGRAM IV
 20 MODE 5
 30 COLOUR 1
 40 PRINTTAB(2,12)"THIS MESSAGE"
 50 PRINTTAB(2,14)"DISPLAYS ALL THE "
 60 PRINTTAB(2,16)"ACTUAL COLOURS"
 70 PRINTTAB(2,18)"IT IS IN LOGICAL "
 80 PRINTTAB(2,20)"COLOUR ONE."
 90 REPEAT
100 FOR I%= 0 TO 15
110 VDU 19,1,I%,0,0,0
120 FOR J%= 1 TO 9999:NEXT J%
130 NEXT I%
140 UNTIL FALSE
```

Program V is more complex. Lines 30-50 assign actual colour 0 (black) to logical colours 1, 2 and 3. This ensures that anything that is written in any logical colour cannot be distinguished from the black background. This is a technique often used to blank out screen pictures that take a long and rather untidy time to build up. Once built up invisibly they are instantly made visible by reassigning the logical colours to the actual colours of the picture.

Lines 60-200 print out a rectangle of blanks (CHR$(32)) in logical colours 1, 2 and 3 in order around the perimeter of the rectangle.

Lines 220-280 then make the colours visible by use of the VDU 19 command. Each time round the loop the actual colours are reassigned to a different logical colour number so that, for example, red appears in different places. This is done cyclically so that the colours appear to rotate around the rectangle.

Finally if you want to test your command of the VDU 19 command, try to arrange it so that the four colours used in Mode 5 are green, cyan, magenta and blue.

```
 10 REM PROGRAM V
 20 MODE 5
 30 FOR I%= 1 TO 3
 40 VDU 19,I%,0,0,0,0
 50 NEXT I%
 60 COLOUR 129
 70 PRINTTAB(6,14)CHR$(32)
 80 PRINTTAB(12,14)CHR$(32)
 90 PRINTTAB(6,20)CHR$(32)
100 PRINTTAB(12,20)CHR$(32)
110 COLOUR 130
120 PRINTTAB(10,14)CHR$(32)
130 PRINTTAB(6,16)CHR$(32)
140 PRINTTAB(12,18)CHR$(32)
150 PRINTTAB(8,20)CHR$(32)
160 COLOUR 131
170 PRINTTAB(6,18)CHR$(32)
180 PRINTTAB(10,20)CHR$(32)
190 PRINTTAB(12,16)CHR$(32)
200 PRINTTAB(8,14)CHR$(32)
210 C%=0
220 REPEAT
230 FOR I%= 1 TO 3
240 VDU 19,I%+128,(C%+I%)MOD3+1,0,0 ,0
250 NEXT I%
260 FOR J%= 1 TO 2000:NEXT
270 C%= (C%+1)MOD3+1
280 UNTIL FALSE
```

Chapter 2: Text and Graphics Screens

IN the last chapter we saw how we could use the COLOUR statement to obtain text in different colours on the screen. We also used the TAB( ) statement to good effect without describing exactly how it functions.

Now although we did not make it explicit all these operations took place on the TEXT screen. To put it simply, if not totally accurately, you can use the screen of your television either for writing, when we say we are using the text screen, or for drawing pictures and suchlike, when we say we are using the graphics screen.

The graphics screen has its own special set of commands entirely different from those we learnt for the text screen. You can mix the text and graphics screens – they can overlap or occupy entirely separate areas or "windows" of the screen. In fact when you enter modes capable of supporting graphics, the text and graphics screens initially coincide – you can print text or draw graphics over the entire area of the VDU. To avoid the schizophrenia that this may induce, for the moment we shall consider the screens in isolation, and not attempt to mix text and graphics.

Before describing the graphics screen, which takes up the bulk of this article, let's formalise our ideas about the text screen. In all eight modes of the BBC Micro we can print characters to the screen. The maximum number of characters on the screen at one time varies from mode to mode.

- **Modes 6** and **7,** text only modes, support 25 lines of 40 characters.
- **Modes 1** and **4** support 32 lines of 40 characters.
- **Modes 2** and **5** support 32 lines of 20 characters.
- **Mode 0** supports 32 lines of 80 characters.
- **Mode 3,** a text only mode, supports 25 lines of 80 characters.

We can consider each character to be occupying one cell of the screen, different modes having different numbers of cells. Now the TAB (X,Y) function uses a type of coordinate system to allow us to print in specific cells. Figure I illustrates the coordinate system available in Mode 4.

Notice that the origin, – point (0,0) – is at the top left of the screen. Y increases as it goes downwards. Also although there are 40 cells for characters across each line, they are numbered 0 to 39. Similarly, down the Y-axis, the range for the 32 lines is from 0 to 31.

Of course in the other modes the values for the X and Y ranges will differ, since the number of character cells differ. The above two points still hold, though.

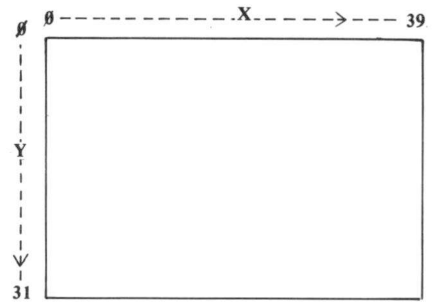


Figure I: Text screen in Mode 4

Program I illustrates a simple use of the TAB( ) function by drawing a line of asterisks diagonally across the screen. Try altering the mode in line 20 to modes 4, 6 and 0 to see the effect on the output. Remember, the meaning of the coordinates varies from mode to mode because of the differing number of

```
10 REM PROGRAM I
20 MODE 5
30 FOR position = 0 TO 19
40 PRINT TAB(position,position)"*"
50 NEXT position
```

cells available. You might also try altering the program so that the diagonal runs from right to left.

Program II draws an 'X' of asterisks using two loops, one for each diagonal. Can you adapt it so that only one loop is used? Also the 'X' is not

```
   10 REM PROGRAM II
   20 MODE 5
   30 FOR position = 0 TO 19
   40 PRINTTAB(position,position)"*"
   50 NEXT position
   60 FOR position = 0 TO 19
   70 PRINTTAB(19-position,position)"
*"
   80 NEXT position
```

central – try to rectify this.

Program III uses nested loops to draw a diamond of asterisks on the screen. See if you can follow the logic, then try adding the following versions of line 65. Can you visualise the effects accurately before you run them?

65 COLOUR (start+length) MOD 3+1

65 COLOUR (top + row) MOD 3 + 1

Incidentally, the VDU 30 on the last line simply returns the cursor to the

```
    10 REM PROGRAM III
    20 MODE 5
    30 centre=9:top=7:bottom=25
    40 FOR row= 1 TO 9
    50 start=centre - row+1
    60 FOR length = 0 TO 2*(row-1)
    70 PRINTTAB(start+length,top+row)"
*"
    80 PRINTTAB(start+length,bottom-ro
w)"*"
    90 NEXT length
   100 NEXTrow
   110 VDU 30
```

top left of the screen to stop it spoiling our pattern.

Program IV uses virtually the same technique to produce a rather striking pattern. Here, instead of asterisks, spaces are used, so the colours which vary are the background colours. We also use VDU 19 to reassign the background colours. Line 90, which ensures that the choice of colours is symmetrical, is far more complex than the lines we added to Program III.

```
    10 REM PROGRAM IV
    20 MODE 5
    30 VDU 19,3,4,0,0,0:VDU 19,0,2,0,0
,0
    40 COLOUR 128:CLS
    50 centre=9:top=7:bottom=25
    60 FOR row = 1 TO 9
    70 start=centre- row+1
    80 FOR length = 0 TO 2*(row-1)
    90 COLOUR(ABS(start+length-9)+ABS(
top+row-16))MOD3+129
   100 PRINTTAB(start+length, top+row)
;" "
   110 PRINTTAB(start+length,bottom-ro
w);" "
   120 NEXT length
   130 NEXT row
   140 VDU 30
```

Now let's have a look at the graphics screen. There is no complicated way of "entering" the graphics screen – when you enter a mode the whole of the screen is available for graphics, as it is for text, of course. We shall concentrate on the former. Actually, the ground we're going to cover is fairly simple – we are going to learn to draw lines in various colours. The command for drawing lines is DRAW and the command to change colours is GCOL, which stands for "graphics colours". The statement COLOUR is used for text colours only.

Firstly, let's consider the coordinate system for the graphics screen, as shown in Figure II.

Three things to note: The origin – point (0,0) – is at the bottom left-hand corner of the screen. Y increases as it goes upwards; the "cells" of the coordinate system range from 0 to 1279 along the X-axis (horizontally) and 0 to 1023 along the Y-axis (vertically). Each is measured in "graphic units"

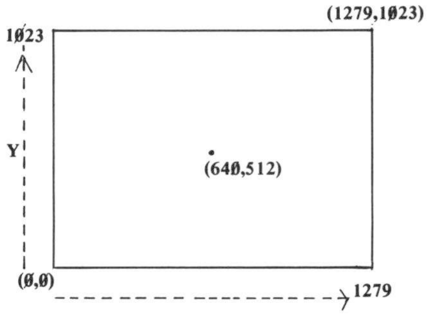


Figure II Graphics Screen

(gu); when you change modes (assuming that you pick a mode capable of supporting graphics), the physical positions of the points don't change. That is, (0,0) will still be at the bottom left-hand corner of the screen, (1279,1023) at the top right, and (640,512) at the centre of the screen.

Let's try drawing some lines on the screen. To do this we imagine something called the graphics cursor, which is invisible yet occupies a specific point on the screen. We use the command DRAW to draw a line between the last two points "mentioned", or "visited" by the graphics cursor. For example, if the last point the graphics cursor had "ended up" at was (0,0), then DRAW 1279,1023 would draw a line between (0,0) and (1279,1023), which is diagonally from bottom left to top right across the screen.

To position the graphics cursor at a point – say the middle, (640,512) – we use the command MOVE 640,512. This places the cursor at the specified point – without making any mark on the screen. Notice that neither MOVE nor DRAW use brackets with the points they are specifying.

```
10 REM PROGRAM V
20 MODE 5
30 MOVE 0,0
40 DRAW 1000,1000
```

Let's try a simple program using these ideas – Program V. As you can see, this draws a simple diagonal line from (0,0) to (1000,1000). Now try drawing it with the additional line **50 DRAW 1000,0** This draws a line vertically down to (1000,0), a point on the X-axis. Remember, DRAW connects the last point the cursor visited – in this case (1000,1000) from line 40 – to the point specified in the DRAW command, which in this case is (1000,0).

Perhaps you can see how **60 DRAW 0,0** will complete a triangle by drawing a horizontal line to the origin. The following completes the square and adds the second diagonal:

```
70 DRAW 0,1000
80 DRAW 1000,1000
90 MOVE 0,1000:DRAW 1000,0
```

The final outline is shown in Figure III. Note the MOVE necessary in line 90. I think that it is impossible to obtain the screen shown in Figure III by using just DRAW without using MOVE or going over the same line twice. Is there anyone out there who can prove me wrong – or at least prove me right?

Before abandoning Program V, try it in Mode 4. There are two things to notice here: When you change mode, the position of the triangle does not alter on the screen – graphics points are fixed and the lines drawn on the screen are much finer than in Mode 5. The latter is because Mode 4 has "higher

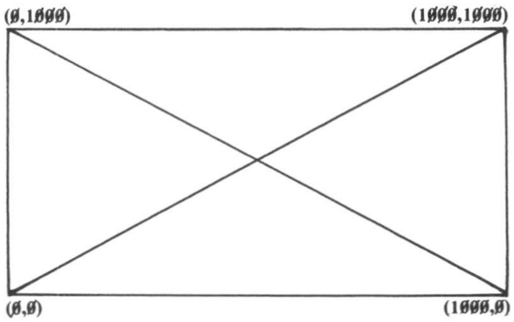


Figure III: Figure produced by Program V

resolution" than Mode 5. That is, it works to a finer "grid" or, if you like, it draws with a finer pen. Mode 4 can fit in more characters per line than Mode 5 because of this "fineness" or higher resolution. In a particular graphic mode the fewer characters per line it can support, the coarser it draws its graphic lines, which means the lower its resolution. So while changing modes may not cause our picture to be drawn in different positions, it certainly affects the detail available.

Now to change the colour of the lines we use the GCOL statement. In Mode 5, assuming we haven't changed the logical colour assignments with VDU 19: **GCOL0,1** will cause future lines to be drawn in red; **GCOL0,2** gives yellow lines; **GCOL0,3** gives white lines; **GCOL0,4** gives black lines.

We say that GCOL0 changes the graphics foreground colour to the logical colour specified after the comma. At the moment we are going to treat the command as if it were always **GCOL0** followed by a logical colour number. In fact, as we shall learn, we can use numbers other than zero before the comma.

Being able to define a graphics foreground colour suggests that we should be able to define a background colour. This works in very much the way it did for text colour – to define a background colour we use GCOL0, with the logical colour number we want as background PLUS 128. For example, GCOL0,129 will give us a red background in Mode 5 (assuming the standard colour assignments).

Just as we used CLS to clear the screen to the background colour while using the text screen, so we use CLG to obtain the same effect on the graphics screen. Thus **GCOL0,130 : CLG** will clear the graphics screen to yellow. Note that although CLG and CLS may appear to achieve the same effect, this is solely due to the graphics and text screens overlapping. If they were separate, they would only effect their respective "windows".

And just as CLS can be obtained with **Ctrl+L** so CLG can be obtained with **Ctrl+P.** Program VI shows the final version of the last program with

```
   10 REM PROGRAM VI
   20 MODE 5
   30 VDU 19,0,5,0,0,0:VDU 19,3,4,0,0
,0
   40 GCOL0,128:CLG
   50 GCOL0,1
   60 MOVE 0,0
   70 DRAW 1000,1000
   80 GCOL0,2
   90 DRAW 1000,0
  100 DRAW 0,0
  110 DRAW 0,1000
  120 DRAW 1000,1000
  130 GCOL0,3
  140 MOVE 0,1000:DRAW 1000,0
```

graphics colours added. Try giving it a nicer colour scheme!

Program VII should demonstrate amply the difference between CLG and CLS – lines 40 and 50 set up different background colours for them to clear to. Press 'S' to stop the program. Program VIII uses all the above techniques to produce a "sunburst" of colour from the origin. It draws 100 lines of random colour to random points on the screen.

```
 10 REM PROGRAM VII
 20 MODE 5
 30 REPEAT
 40 COLOUR 129
 50 GCOL0,130
 60 CLS
 70 A$=GET$
 80 CLG
 90 A$=GET$
100 UNTIL A$="S"
```

Try adding all, or random combinations of the following lines to Program VIII:

```
105 MOVE 1279,0:DRAW x,y
106 MOVE 1279,1023:DRAW x,y
107 MOVE 0,1023:DRAW x,y
```

Can you predict the outcome before running it?

```
   10 REM PROGRAM VIII
   20 MODE 5
   30 VDU 19,0,5,0,0,0:VDU 19,3,4,0,0
,0
   40 count=0
   50 REPEAT
   60 count=count+1
   70 MOVE 0,0
   80 x=RND(1279):y=RND(1023)
   90 GCOL0,RND(3)
  100 DRAW x,y
  110 UNTIL count >99
```

Another variation is to alter the assignment of logical colours in the sunburst after it has been drawn. For this add:

```
120 REPEAT
130 A=RND(4)-1:B=RND(7)
140 VDU 19,A,B,0,0,0
150 FOR I= 1 TO 2000:NEXT I
160 UNTIL FALSE
```

Program IX plots a series of random triangles over the screen, using a simple procedure PROCtriangle that produces an equilateral triangle using MOVE and DRAW. Figure IV should make the workings of this procedure clearer. The actual values of *x,y* and *length* are passed to the procedure in line

```
   10 REM PROGRAM IX
   20 MODE 5
   30 VDU 19,3,4,0,0,0
   40 TALLY=0
   50 REPEAT
   60 XPOS=RND(1000):YPOS=RND(800)
   70 SIZE=RND(500)
   80 PROCtriangle(XPOS,YPOS,SIZE,TAL
LY MOD3+1)
   90 TALLY=TALLY+1
  100 UNTIL TALLY=50
  110 END
  120 DEFPROCtriangle(x,y,length,colo
ur)
  130 LOCAL height
  140 GCOL0,colour
  150 height=length*1.732/2
  160 MOVE x,y
  170 DRAW x+length,y
  180 DRAW x+length/2,y+height
  190 DRAW x,y
  200 ENDPROC
```

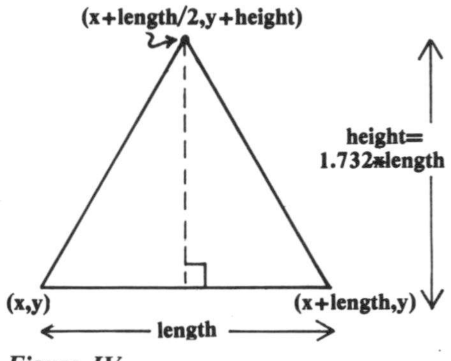


Figure IV

80 as *XPOS, YPOS* and *SIZE* respectively, which have been determined randomly.

Program X then uses exactly the same procedure to nest three equilateral triangles within a fourth. To round off this chapter, you might like to define a procedure for drawing a rectangle – by defining two corners, or alternately using one corner, length and breadth. Then, as in Program IX, use this procedure to draw random rectangles on the screen.

Finally, as in Program X, try nesting the rectangles within one another.

```
   10 REM PROGRAM X
   20 MODE 5
   30 VDU 19,3,4,0,0,0
   40 TALLY=0
   50 SIZE=1000
   60 HEIGHT=1.732*SIZE/2
   70 REPEAT
   80 BASE=SIZE - TALLY*250
   90 UP=HEIGHT*TALLY/4
  100 XPOS=100 + 250*TALLY/2
  110 YPOS=UP/2 + 100
  120 PROCtriangle(XPOS,YPOS,BASE,TAL
LY MOD3+1)
  130 TALLY=TALLY+1
  140 UNTIL TALLY >3
  150 END
  160 DEFPROCtriangle(x,y,length,colo
ur)
  170 LOCAL height
  180 GCOL0,colour
  190 height=length*1.732/2
  200 MOVE x,y
  210 DRAW x+length,y
  220 DRAW x+length/2,y+height
  230 DRAW x,y
  240 ENDPROC
```

Chapter 3: Areas of Colour

SO far we have learned how to draw coloured lines on the graphics screen and use them to outline various shapes. Now we shall see how to fill those shapes with colour so as to really bring the screen to life.

Firstly, let's recap. We learned these new commands:
GCOL0, which sets the graphics colours.
MOVE, which moves the (imaginary) graphics cursor to the point specified.
DRAW, which draws a line, in the current foreground colour, from the last

```
10 REM PROGRAM I
20 MODE 5
30 GCOL 0,1
40 GCOL 0,130:CLG
50 MOVE 10,10
60 DRAW 1270,10
70 DRAW 640,1020
80 DRAW 10,10
```

point visited by the graphics cursor to the point specified.

Program I illustrates the use of these commands to draw a red triangle similar to that of Figure I. We can cause the triangle to be filled in with colour by using a new statement, PLOT 85. Before we go into it in detail, I suggest that you run Program II to get a feel of what happens – it draws the same triangle as Program I, this time completely filled with red, the graphics foreground colour.

When you think about it, you can specify a triangle on the screen by giving

```
10 REM PROGRAM II
20 MODE 5
30 GCOL 0,1
40 GCOL 0,130:CLG
50 MOVE 10,10
60 DRAW 1270,10
70 PLOT 85,640,1020
```

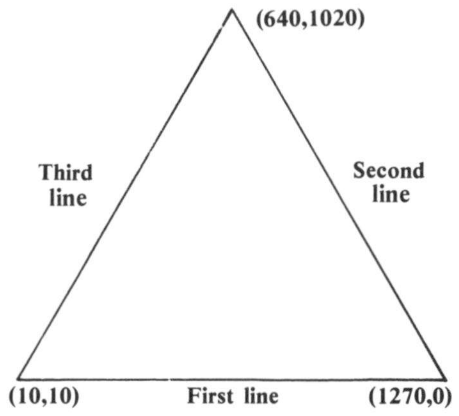


Figure I: Drawing a simple triangle

the coordinates of its three corners. Now Plot 85 is the BBC Micro's triangle-filling command. When the machine receives this command it needs to know those three sets of coordinates. You always follow PLOT 85 with the coordinates of one of the points. For example, in Program II, line 70 uses PLOT 85,640,1020, since the coordinates of the top of the triangle are (640,1020). But how does the BBC Micro know where to get the other two points to complete the triangle?

Well, it takes it for granted that the other two points are the last two points the graphics cursor has visited before it meets the PLOT 85 statement. So when you are programming you have to keep track of the last two positions the graphics cursor has visited – remembering that the MOVE and DRAW affect this. If the last two points are unsuitable for the triangle you want to

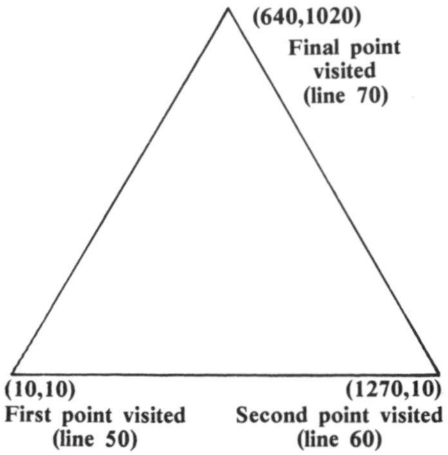

(640,1020)
Final point visited (line 70)
(10,10)
First point visited (line 50)
(1270,10)
Second point visited (line 60)

Figure II: How Program II works

draw you have to fix this by using MOVE to visit the appropriate points.

In Program II lines 50 and 60 use MOVE to visit the first two points of the triangle. Line 70 then uses PLOT 85 to specify the last point and fill in the triangle defined with the current foreground colour. Figure II should make this clear. This ability to fill triangles is the key to the whole business of graphics. Virtually all the other shapes you see in BBC Micro programs are constructed from triangles – even the circles!

It is worth spending some time now playing with variations of Program II. It's easy to read and understand what we've been doing so far – but it is another thing to put the ideas to use. So please, before you continue, have a go at writing programs based on Program II to draw your own triangles on the screen. Try changing their size and colour.

```
 10 REM PROGRAM III
 20 MODE 5
 30 VDU 19,3,4,0,0,0
 40 VDU 19,0,7,0,0,0
 50 GCOL 0,128:CLG
 60 REPEAT
 70 firstx=RND(1200)
 80 firsty=RND(1000)
 90 secondx=RND(1200)
100 secondy=RND(1000)
110 thirdx=RND(1200)
120 thirdy=RND(1000)
130 colour=RND(3)
140 GCOL 0,colour
150 MOVE firstx,firsty
160 MOVE secondx,secondy
170 PLOT 85,thirdx,thirdy
180 FOR I=1 TO 1000:NEXT I
190 UNTIL FALSE
```

Then write a program to put two on the screen at once. Can you make them different colours? What happens if they overlap? And what happens if you change MOVE in lines 50 and 60 to DRAW? (Putting in a line 75 to change the graphics colour might help here.)

Program III uses the ideas of Program II to generate a random sequence of triangles.

Line 20 sets the mode.

Lines 30 and 40 alter the colour assignments of logical colours 3 and 0 respectively.

Lines 60 and 190 make up the REPEAT UNTIL LOOP which generates the triangles.
Lines 70 to 130 pick out at random the three points *(firstx,firsty)*, etc. and choose the colour.
Lines 150 to 170 do the actual work of plotting the triangles.
150 and 160 MOVE the cursor to the first and second points respectively.
Line 170 fills the triangle between these and the third point with a PLOT 85.

```
    10 REM PROGRAM IV
    20 MODE 5
    30 VDU 19,3,4,0,0,0
    40 tally=0
    50 REPEAT
    60 xpos=RND(1000):ypos=RND(800)
    70 size=RND(500)
    80 PROCtriangle(xpos,ypos,size,tal
ly MOD 3+1)
    90 tally=tally+1
   100 FOR wait=0 TO 500:NEXT wait
   110 UNTIL tally=50
   120 END
   130 DEFPROCtriangle(x,y,length,colo
ur)
   140 LOCAL height
   150 GCOL 0,colour
   160 height=length*1.732/2
   170 MOVE x,y
   180 MOVE x+length,y
   190 PLOT 85,x+length/2,y+height
   200 ENDPROC
```

If the program were any more complex it would have been better to put the triangle-drawing part of it in a procedure. This is the strategy we adopt in Program IV, which prints out 50 random equilateral triangles on the screen by repeatedly calling PROCtriangle.

If you experience a feeling of déjà vu when you look at PROCtriangle, don't worry – it is virtually identical to the procedure of that name in the last chapter's Program IX, save that we use PLOT 85 since we are filling in triangles rather than drawing outlines. There's a lot to be gained from comparing both procedures.

Figure III should also help make PROCtriangle clearer. Line 60 randomly chooses the position of the left hand corner of the triangle *(xpos,ypos)*. Line 70

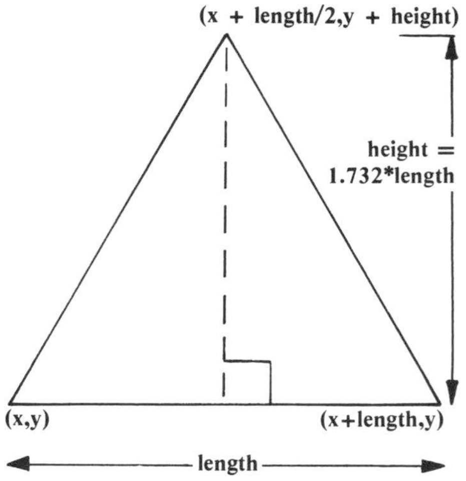


Figure III: PROCtriangle variables

fixes the size of the triangle. *tally* counts the number of times the REPEAT UNTIL loop (lines 50 to 110) is repeated. A little thought should show you that *tally* MOD 3 + 1 returns the values 1,2,3,1,2,3,1,2,3 cyclically. We use this to cycle through the colours for the triangle by passing *tally* MOD 3 + 1 to the variable colour in the procedure call (lines 80 to 130).

In the last chapter, we not only used PROCtriangle to draw the random triangle outlines, we also used it in Program X to draw the nested triangles. I have followed in the same vein by using PROCtriangle from Program IV to produce a series of colour-filled triangles, as you will see if you run Program V.

```
   10 REM PROGRAM V
   20 MODE 5
   30 VDU 19,3,4,0,0,0
   40 tally=0
   50 size=1000
   60 height=1.732*size/2
   70 REPEAT
   80 base=size-tally*250
   90 up=height*tally/4
  100 xpos=100+250*tally/2
  110 ypos=up/2+100
  120 PROCtriangle(xpos,ypos,base,tal
ly MOD 3+1)
  130 tally=tally+1
  140 UNTIL tally>3
  150 END
  160 DEFPROCtriangle(x,y,length,colo
ur)
  170 LOCAL height
  180 GCOL 0,colour
  190 height=length*1.732/2
  200 MOVE x,y
  210 MOVE x+length,y
  220 PLOT 85,x+length/2,y+height
  230 ENDPROC
```

In Program V we not only use *tally* to once more cycle through the colours, but also to alter the position and size of the triangle, so that each successfully nests within the preceding one (lines 80 to 110). Program VI uses two different coloured triangles positioned to give a multicoloured rectangle. It does this by repeating our triangle drawing formula:
1. MOVE to first point
2. MOVE to second point
3. PLOT 85 to third point.

```
 10 REM PROGRAM VI
 20 MODE 5
 30 VDU 19,3,4,0,0,0
 40 REM First triangle
 50 GCOL 0,1
 60 MOVE 0,0
 70 MOVE 1279,0
 80 PLOT 85,1279,1023
 90 REM Second triangle
100 GCOL 0,3
110 MOVE 0,0
120 MOVE 0,1023
130 PLOT 85,1279,1023
```

Figure IV should make this clear. This is not the most efficient method, though. If we take care in choosing the order of the points we visit we can arrange that the last two points visited in order to draw the first triangle become the first two points of the second. That is, having drawn the first triangle, we can then draw the second with just another PLOT 85 to supply the final point.

Program VII uses this method to produce the same output as Program VI. Figure V should illustrate the idea. To show how important the order of

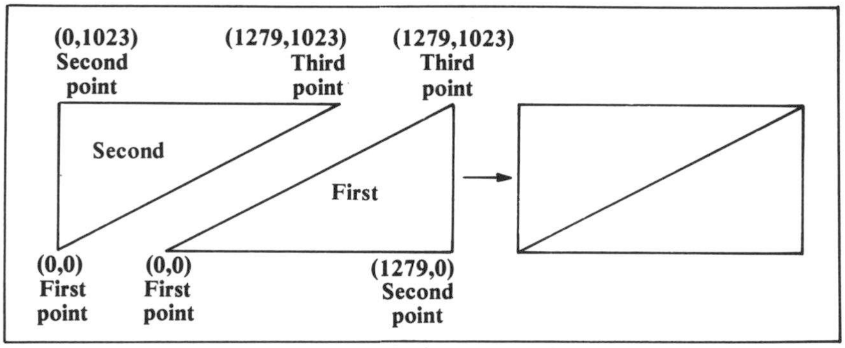


Figure IV: Constructing a rectangle from triangles

```
 10 REM PROGRAM VII
 20 MODE 5
 30 VDU 19,3,4,0,0,0
 40 REM First triangle
 50 GCOL 0,1
 60 MOVE 1279,0
 70 MOVE 1279,1023
 80 PLOT 85,0,0
 90 REM Second triangle
100 GCOL 0,3
110 PLOT 85,0,1023
```

visiting the points is when you're using this method, try swapping lines 60 and 70, then run the program.

If you think about it we can use this technique in a procedure to draw rectangles – though we'd probably have both triangles the same colour. In fact, we use this in PROCrectangle in Program VIII. The procedure assumes that the sides of the rectangle are parallel to the axes, that is that the rectangle does not slope. Figure VI should help make the procedure's variables clear.

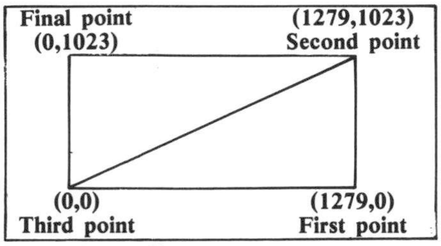


Figure V: Efficient rectangles!

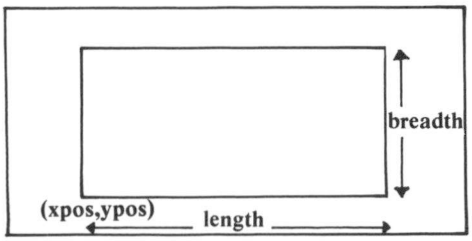


Figure VI: Variables used in Program VIII

The program simply computes random values for those variables (once more using tally MOD 3 + 1 to pick colours) and calls PROCrectangle 50 times to produce random rectangles in much the same manner as we produced random triangles in Program IV.

```
   10 REM PROGRAM VIII
   20 MODE 5
   30 VDU 19,3,4,0,0,0
   40 tally=0
   50 REPEAT
   60 xpos=RND(1000):ypos=RND(800)
   70 length=RND(500)
   80 breadth=RND(500)
   90 PROCrectangle(xpos,ypos,length,
breadth,tally MOD 3+1)
  100 tally=tally+1
  110 FOR wait=0 TO 500:NEXT wait
  120 UNTIL tally=50
  130 END
  140 DEFPROCrectangle(xpos,ypos,widt
h,height,colour)
  150 GCOL 0,colour
  160 MOVE xpos+width,ypos
  170 MOVE xpos,ypos
  180 PLOT 85,xpos+width,ypos+height
  190 PLOT 85,xpos,ypos+height
  200 ENDPROC
```

Continuing with this theme of adapting previous programs, Program IX uses PROCrectangle to produce a series of nested rectangles in a manner strictly analogous to the way that Program V produced nested triangles. Lines 70 to 100 ensure that successive rectangles nest by altering the side lengths and the corner positions.

Although I normally use the above procedure for rectangles, there is

```
   10 REM PROGRAM IX
   20 MODE 5
   30 VDU 19,3,4,0,0,0
   40 tally=0
   50 length=1000:breadth=800
   60 REPEAT
   70 width=length*(4-tally)/4
   80 height=breadth*(4-tally)/4
   90 xpos=100+500*tally/4
  100 ypos=100+400*tally/4
  110 PROCrectangle(xpos,ypos,width,h
eight,tally MOD 3+1)
  120 tally=tally+1
  130 UNTIL tally>3
  140 END
  150 DEFPROCrectangle(xpos,ypos,widt
h,height,colour)
  160 GCOL 0,colour
  170 MOVE xpos+width,ypos
  180 MOVE xpos,ypos
  190 PLOT 85,xpos+width,ypos+height
  200 PLOT 85,xpos,ypos+height
  210 ENDPROC
```

another way of defining a rectangle (again assuming it doesn't slope). This is by simply giving the procedure the co-ordinates of two diagonally opposite corners of the rectangle.

Figure VII shows the method. Program X uses it in PROCrectangle to produce a "staircase" of six rectangles. Each successive rectangle is drawn *ystep* graphical units taller than the preceding one and *xstep* graphical units to the right. Although the output may seem rather trivial, it is by using much the same techniques that we are able to draw bar charts and graphs.

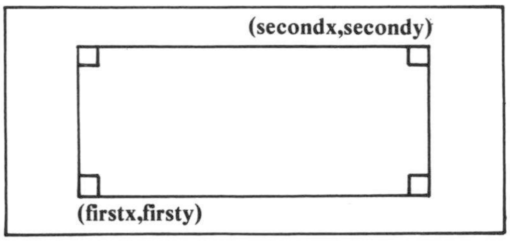


Figure VII: Alternative rectangle definition

```
   10 REM PROGRAM X
   20 MODE 5
   30 VDU 19,3,4,0,0,0
   40 VDU 19,0,5,0,0,0
   50 bottomx=0:bottomy=0
   60 topx=0:topy=0
   70 xstep=213:ystep=170
   80 counter=0
   90 REPEAT
  100 colour=counter MOD 3+1
  110 topy=topy+ystep
  120 topx=bottomx+xstep
  130 PROCrectangle(bottomx,bottomy,t
opx,topy,colour)
  140 bottomx=topx
  150 counter=counter+1
  160 UNTIL counter=6
  170 END
  180 DEFPROCrectangle(firstx,firsty,
secondx,secondy,colour)
  190 GCOL 0,colour
  200 MOVE secondx,firsty
  210 MOVE firstx,firsty
  220 PLOT 85,secondx,secondy
  230 PLOT 85,firstx,secondy
  240 ENDPROC
```

In the meantime, why not practice your graphic techniques by writing programs to draw simple, multicoloured pictures constructed from triangles and rectangles? Houses, rockets and boats seem to be favourite subjects.

Chapter 4: Looking into Windows

RIGHT – lots of "hands-on" in this chapter, so let's get to it! Enter the following:

```
MODE 5
VDU 28,0,31,9,0
COLOUR 129
CLS
```

What exactly is going on? Well, if you cast your mind back you'll remember that the display we look at is actually composed of two screens as far as the BBC is concerned, one on top of the other. They were called the graphics and text screens, and when you first switch on or change mode they overlap.

When we used PRINT, TAB( ), COLOUR and CLS we were using the text screen. When we used MOVE, DRAW, PLOT, GCOL and CLG we were using the graphics screen. Normally both screens are on top of each other, and we might be tempted to think of them as really the same thing.

They are quite separate, though. If you doubt me, set the graphics background to one colour and the text background to another. Then alternately try CLS and CLG. That should convince you!

But to return to our present problem, if you've done what I asked you should be left with a screen red on the left half and black on the right, with some remnants of your typing. VDU 28 has restricted the BBC Micro's text screen to the left half of the display. To use the jargon, we have created a text window. We'll go into the details in a moment. Let's just prove that we have made such a window and see what it means.

Type **CLG**. Everything disappears to leave a blank screen, save for the prompt and the flashing cursor. You see, the graphics screen is still full sized, and when you clear it you also wipe out what's on the text screen – that is, the red rectangle disappears. This should become clearer in a moment.

Notice that the prompt is written on a red background. After all, ">" is a text character, and so is printed on the text background, which is red.

Now hold down a key and keep it down so that it repeats. Surprised? The repeated letter only gets halfway across the screen before starting a new line. This is because we have restricted the text screen to half the screen's width.

In Mode 5 the screen is 20 characters wide, numbered 0 to 19, as you can see from Figure I. Now you can restrict this text screen to any particular rectangular "chunk" or window you choose. Of course, that rectangle mustn't

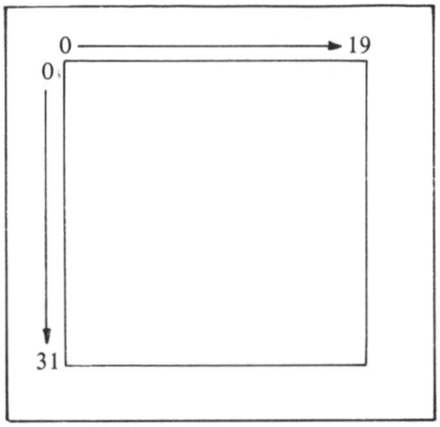


Figure I: Text screen in Mode 5

slope, it must be upright.

We have seen how we could fix such a rectangle with just two points, the opposite corners. To define a text rectangle, or window, we use VDU 28 followed by the character co-ordinates of the bottom left corner of the screen, then those on the top right corner.

Look at Figure II. This shows the text window we set up at the beginning of the article. To define, or fix the shaded area as a text window we typed **VDU 28,0,31,9,0.** Notice that we use commas to separate the figures, but there is no final comma.

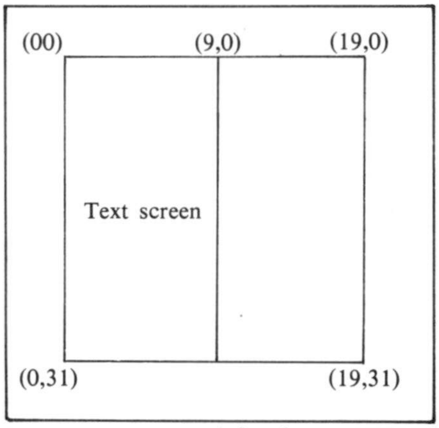


Figure II: Our re-defined text screen

Although we have defined a text window, the graphics screen works normally. Try:

```
MOVE 0,0
DRAW 1279,1023
```

The graphic commands act as they usually do and overwrite the text window as if it weren't there. After all, the graphics screen still fills the whole of the display, and we have only limited the text screen.

Before you clear the screen, press the Return key and keep it down to see what happens. When the prompt reaches the bottom the text window will scroll as normal. However, the only part of the line we drew that scrolls up is the part that crosses the text window. The rest of the line is immune.

This means that if we use our windows carefully we can stop text and graphics intefering with each other.

Now don't think that the text window has to go on the left side of the screen, or from top to bottom. It can be any rectangular portion of the screen. Try **VDU 28,7,20, 12,11** which sets up the text window in Figure III. (Notice that you are still in the old text window as you type this – it doubles back on you.)

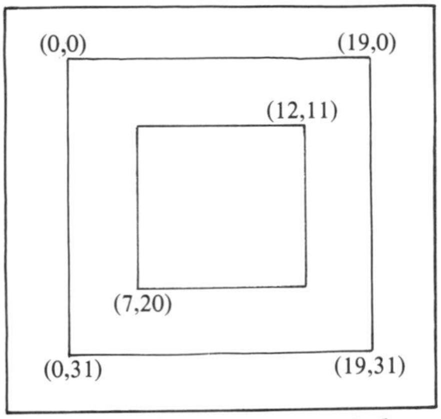


Figure III: Altering our text window

Now type **CLG** which should clear the whole display, since the graphics area hasn't been restricted. Then type **CLS**. The text window shown in Figure III should appear.

Try typing a few words in here and see what happens. This proves that:

● You can have a text window anywhere on the screen.

● Redefining a text window (that is using VDU 28 twice) automatically destroys the old one, although it doesn't clear it.

Now type:

```
VDU 26
CLS
```

Hopefully the whole screen will turn red. The effect of VDU 26 is to restore the text and graphics windows to the state they were in when you switched on – totally overlapping.

```
10 REM PROGRAM I
20 MODE 5
30 VDU 28,7,20,12,11
40 FOR I= 0 TO 31
50 PRINT TAB(I)"*"
60 A$=GET$
70 NEXT I
80 A$=GET$
90 CLS:VDU 26
```

Program I illustrates the use of the simple TAB( ) function in a text window. Line 30 sets up a text window identical to the one we last used. 40 to 70 form a loop attempting to print an asterisk in all positions from TAB(0) to TAB(31). The A$=GET$ in line 60 is simply to step you through each printing, as you have to press a key before the program continues. Line 90 returns the screen to normal.

It should be immediately apparent that the zero position as far as TAB( ) is concerned is at the left of the new text window, TAB(1) is next to that and so on. Since the text window is only six characters wide (columns 7 to 12 on the original display, 0-5 on our text screen), TAB(6) will be a complete line across the text screen plus one column. That is, the asterisk will appear to miss a line. Similarly, TAB(7) is a complete line plus two characters and so on.

This wraparound effect is exactly what happens with large values of TAB( ) on the original text screen. Program II demonstrates this.

```
10 REM PROGRAM II
20 MODE 5
30 FOR I= 1 TO 255
40 PRINT TAB(I)"*"
50 A$=GET$
60 NEXT I
```

The main point is that in a text window TAB(0) is the left hand column of that window and TAB( ) only considers the width of the text screen in its workings.

Much the same thing happens with the "multiple" TAB(,) such as TAB (3,4). TAB(0,0) is at the top left of the text window. However, unlike the simple TAB( ), if the number in the brackets exceeds the size of the window, this multiple TAB(,) ceases to work. (It also collapses in this way when you go out of bounds on the normal screen.) Program III illustrates the point.

```
10 REM PROGRAM III
20 MODE 5
30 VDU 28,7,20,12,11
40 FOR I = 0 TO 31
50 PRINT TAB(I,I)"*"
60 A$=GET$
70 NEXT I
80 A$=GET$
90 CLS:VDU26
```

Experiment with setting up your own text screens. See if COLOUR works normally. What happens when text and graphics overlap? Do they overwrite each other? What exactly happens when scrolling occurs? What effect does getting your co-ordinates mixed up and using the top right corner instead of the bottom left have, as in **VDU 28,12,12,7,20?**

Just as we can define a text window on the screen, so we can define a graphics window. That is, we can restrict the area where our graphics commands apply to a rectangle within the whole screen.

We use VDU 24 to do this, followed by the co-ordinates of the bottom left corner of the window and the top right hand corner. Of course, we now use the co-ordinates of the graphics screen not the text screen. To define the graphics window shown in Figure IV we use **VDU 24,200;300;1000;800;**

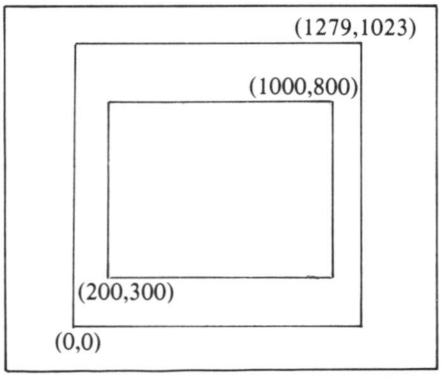


Figure IV: Defining a graphics window

Notice that while we use a comma to separate the VDU 24 from the list of co-ordinates, after that each co-ordinate is followed by a semi-colon (;). Also there is a final semi-colon – if you omit this your programs will crash.

Now enter **MODE 5**, then:

```
VDU 24,200;300;1000;800;
COLOUR 129
GCOL 0,130
```

Now try:

CLS
CLG

alternately until you get the hang of things. Nice, isn't it! Notice how clearing the text screen also clears the graphics area to the text background colour. This is because the text window still occupies the whole screen, so when you clear it the graphics screen, which also overlaps it, suffers the same fate. Similarly, text gets printed across the graphics area.

The only way to avoid this problem is to define two entirely separate text and screen areas.

Now with the graphics screen as defined above try:

MOVE 0,0
DRAW 1000,1000

You'll see that only the part of the line that crosses the graphics window appears – you really have a "window" onto part of the original graphics screen. So (0,0) is still where it was on the graphics screen, as is (1000,1000). Having a graphics window does not automatically move (0,0) to the bottom left of that window.

The only graphic effects you will see from your graphic commands will be those that occur within the region of your graphics window. Try drawing a few triangles with parts outside the graphics windows to see what I mean.

Just as with text windows, defining a new graphics window immediately cancels the old one (without clearing it from the screen). Also VDU 26 still has the effect of restoring the text and graphics areas to their original extent.

We can use the idea of defining graphics windows to draw rectangles on the

```
   10 REM PROGRAM IV
   20 MODE 5
   30 VDU 19,3,4,0,0,0
   40 FOR loop%= 1 TO 10
   50 leftx=RND(1000):rightx=RND(1000
)
   60 IF leftx>rightx THEN store=left
x:leftx=rightx:rightx=store
   70 lefty=RND(1000):righty=RND(1000
)
   80 IF lefty>righty THEN store=left
y:lefty=righty:righty=store
   90 VDU 24,leftx;lefty;rightx;right
y;
  100 GCOL0,RND(3)+128:CLG
  110 NEXT loop%
```

screen rapidly. We just define a graphics window where we want the rectangle, then clear to the appropriate background colour. Program IV illustrates the techniques, by printing out ten random rectangles on the screen.

Finally, how about writing a simple program for children? Define a graphics window in the top two-thirds of the screen and a text window on the bottom third. Try not to let them overlap.

Then draw a house piece by piece in the graphics screen. Each time you press a key another bit appears. As each part appears, make sure that the name of that part is printed on the text screen. If you can manage that, you are well on the way to mastering graphics!

Chapter 5: Getting Coordinated

WE now know how to define graphics windows with the use of VDU 24. However, when you define a window the graphics coordinate system does not change in line with this new window.

What this means is that (0,0), the point at the bottom of the graphics screen at switch on, doesn't automatically move to the bottom left hand corner of the window. This can lead to graphics effects being chopped off, as Program I demonstrates.

```
 10 REM PROGRAM I
 20 MODE 5
 30 VDU 24,200;300;1000;800;
 40 VDU 19,0,5,0,0,0
 50 VDU 19,3,4,0,0,0
 60 count=0
 70 REPEAT
 80 count=count+1
 90 MOVE 0,0
100 x=RND(1279):y=RND(1023)
110 GCOL 0,RND(3)
120 DRAW x,y
130 UNTIL count>99
```

Line 30 defines a graphics window, illustrated in Figure I. Only the graphics within this window are displayed. To see the full sunburst effect of

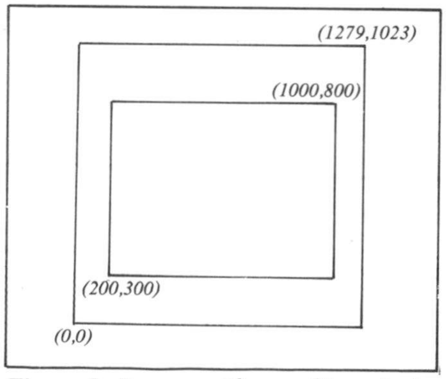


Figure I: Program I's graphics window

the program try leaving out line 30. Incidentally, lines 20 and 30 have to be in that order. Changing modes destroys any graphics or text windows, so these must be defined after you change.

We could make sure the beginning of our sunburst fits into the graphics window by MOVEing to its bottom left hand corner each time before we DRAW. That is, we could change line 90 to **90 MOVE 200,300**

There is a neater way. Simply tell the micro that, from now on, it is to consider the point (0,0) – called the origin – to be at the bottom left of the graphics display. We do this with the VDU 29 command. This allows us to place the origin at any point on the display.

For instance **VDU 29,640;512;** places the origin at the centre of the screen. This means that from now on (0,0) will refer to the middle of the screen. Try:

```
MODE 5
VDU 29,640;512;
MOVE 0,0
DRAW 200,200
DRAW 200,0
DRAW 0,0
```

to prove that, as far as the micro is concerned, (0,0) has been moved to the centre of the screen. Notice how we use the command to move the origin:

- VDU 29 followed by a comma.
- The X co-ordinate followed by a semicolon.
- The Y co-ordinate followed by another semicolon.

Make sure you get your commas and semicolons in the right places!

In Program II line 40 ensures that the origin is moved to (200,300), which is at the bottom left hand corner of the graphics window defined in line 30.

```
 10 REM PROGRAM II
 20 MODE 5
 30 VDU 24,200;300;1000;800;
 40 VDU 29,200;300;
 50 VDU 19,0,5,0,0,0
 60 VDU 19,3,4,0,0,0
 70 count=0
 80 REPEAT
 90 count=count+1
100 MOVE 0,0
110 x=RND(1279):y=RND(1023)
120 GCOL 0,RND(3)
130 DRAW x,y
140 UNTIL count>99
```

This ensures that the sunburst starts at the bottom of the window. The MOVE 0,0 of line 100 makes certain that the DRAW of line 120 starts from there. Moving the origin in this way is more formally known as "redefining the graphics origin". You can use this idea of redefining the origin without using a graphics window – though you often do, since they work quite nicely together. Look at Program III.

```
 10 REM PROGRAM III
 20 MODE 5
 30 VDU 29,640;512;
 40 VDU 19,0,5,0,0,0
 50 VDU 19,3,4,0,0,0
 60 count=0
 70 REPEAT
 80 count=count+1
 90 MOVE 0,0
100 x=RND(640):y=RND(512)
110 GCOL 0,RND(3)
120 DRAW x,y
130 UNTIL count>99
```

Line 30 moves the origin to the centre of the screen. Although we haven't defined a window here, you might think that we are restricted to the top right hand corner of the screen because of the position of the origin. We can use the rest of the screen, though, if we use negative co-ordinates. These involve using numbers smaller than 0, that is, numbers with a minus ('–') sign in front of them.

So far we have just used positive coordinates, which are numbers bigger than or equal to zero. Figure II illustrates the idea. If both coordinates are

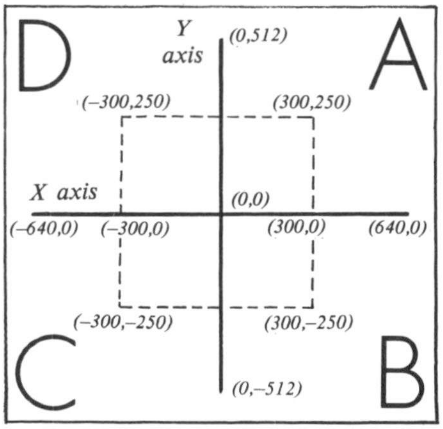


Figure II: Positive and negative coordinates

positive the point will be in region A. If the X coordinate is positive and the Y coordinate negative, it is in region B. If both coordinates are negative the point is in region C, while if the X is negative and the Y positive it will be in region D.

Notice that as you go left along the X axis from the origin the figure following the minus sign increases. That is, –300 is nearer to the origin than –600. Similarly, as you move down the Y axis from the origin, –250 is nearer to the origin than –500.

Try running Program III with the following versions of line 100:

```
100 x=RND(640):y=–RND(512)
100 x=–RND(640):y=–RND(512)
100 x=–RND(640):y=RND(512)
```

Each version will draw our sunburst at different corners of the screen. Program IV uses the idea of negative coordinates to produce a full sunburst effect.

```
 10 REM PROGRAM IV
 20 MODE 5
 30 VDU 29,640;512;
 40 VDU 19,0,5,0,0,0
 50 VDU 19,3,4,0,0,0
 60 count=0
 70 REPEAT
 80 count=count+1
 90 GCOL 0,RND(3)
100 x=RND(640):y=RND(512)
110 MOVE 0,0:DRAW x,y
120 x=RND(640):y=-RND(512)
130 MOVE 0,0:DRAW x,y
140 x=-RND(640):y=-RND(512)
150 MOVE 0,0:DRAW x,y
160 x=-RND(640):y=RND(512)
170 MOVE 0,0:DRAW x,y
180 UNTIL count>99
```

Program V uses a procedure, PROCburst, to give the sunburst effect. PROCburst is defined to allow us to choose the position of the origin *(xpos,ypos)* and the maximum size of line.

Variable *x* and *y* are then chosen and used in the four combinations of negative and positive (lines 130 to 170). This gives the sunburst – this time of one colour – a pleasing symmetry. The procedure is called three times. The value of *xpos* is determined by the variable parameter, as is the maximum line length.

The value of *ypos* is actually fixed at 512. If you like, you can replace the

512 in line 70 with 512/2↑ count, which moves each successive sunburst down the screen.

```
   10 REM PROGRAM V
   20 MODE 5
   30 VDU 19,0,5,0,0,0:VDU 19,3,4,0,0
,0
   40 FOR count=0 TO 2
   50 parameter=1100/(2^count)
   60 GCOL 0,count+1
   70 PROCburst(parameter,512,paramet
er)
   80 NEXT count
   90 END
  100 DEFPROCburst(xpos,ypos,size)
  110 VDU 29,xpos;ypos;
  120 FOR loop=0 TO 50
  130 x=RND(size):y=RND(size)
  140 MOVE 0,0:DRAW x,y
  150 MOVE 0,0:DRAW x,-y
  160 MOVE 0,0:DRAW -x,-y
  170 MOVE 0,0:DRAW -x,y
  180 NEXT loop
  190 ENDPROC
```

Chapter 6: Going Round in Circles

NOW we are going to look at drawing circles on the BBC Micro. It would be nice if we had a CIRCLE command which would allow us to specify the coordinates of the centre of the circle and the radius we want it to be. Unfortunately the BBC Micro does not have such a command.

We can, however, design a general purpose procedure to do it – once we're sure exactly what we want to do and how to do it! The mathematics behind drawing circles takes a lot of working out, but you don't have to understand the sums to be able to use them effectively.

The secret lies in the use of triangles. Thousands of years ago man found that he could survey land by the technique of triangulation. Any piece of land could be approximately divided into a set of triangles. You could then measure those triangles and find the total area. What we're going to do is to divide the circle into triangles, which we can then fill with our PLOT85 command – we'll see exactly how later.

First of all let's look at a particular set of triangles that should help us to do the job properly – right angled triangles. In them one of the sides sticks out vertically from the other – a carpenter would say that the two sides are square to one another. Mathematically speaking, the two sides are at 90 degrees to each other – a right angle.

Actually we've been using right angled triangles throughout this book to give our X and Y coordinates. The reason you haven't noticed is that we haven't drawn in the third side. Figure I illustrates the point. The X and Y we

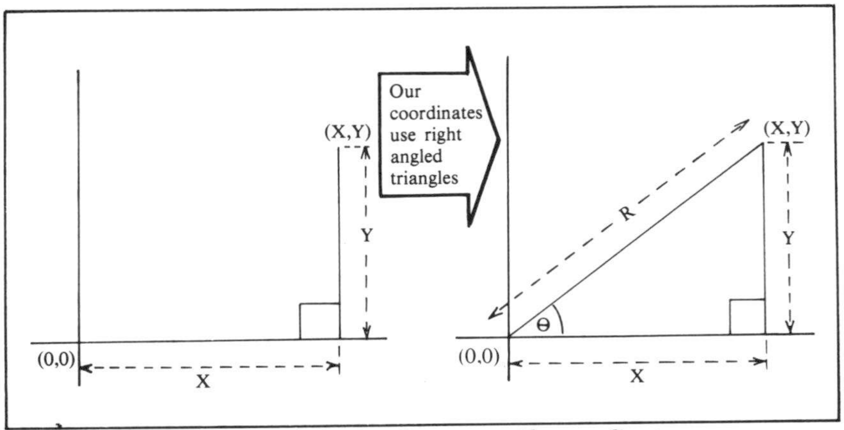


Figure I: Using right-angled triangles to specify coordinates

use to give us our coordinates are two sides of a right angled triangle. Notice the following:

● The sides we take as X and Y are the two that are at right angles to each other.

● We mark the fact that they are at right angles with the sign ⌜.

● The third side of the triangle is opposite the right angle. We call that side the hypotenuse – though I've labelled it R to make things easier! It is R graphic units long.

● I've marked one of the angles Θ.

The thing about right angled triangles is that if you know what R and Θ are you can easily work out X and Y's values. We use two of BBC Basic's functions, SIN( ) and COS( ).

Then, for any particular R and Θ: **X=R*COS(Θ)** and **Y=R*SIN(Θ)** Let's not worry exactly how it works, but just put it to work.

The idea is that we can now represent any point on the screen by using either X,Y coordinates – called cartesian coordinates – or by using R and Θ – called polar coordinates. It happens that we can draw circles more easily thinking in terms of polar coordinates. The trick is that every point on the circumference, or edge, of the circle is the same distance from the centre – so for a circle R will remain the same length for all its points. That's why we're using R : R for radius.

All we have to do to specify a particular point on the circumference is to give the value of Θ that "points" the radius at it. Figure II shows how we can

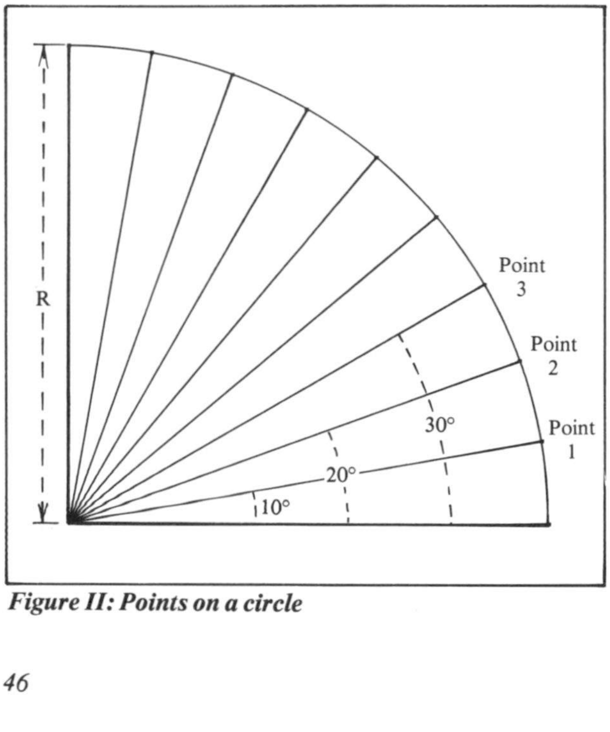


Figure II: Points on a circle

use this idea to draw a quarter circle by illustrating how we can divide the right angle up into nine steps of 10 degrees. In our old system of coordinates *Point1* is **(R*COS(10),R*SIN(10))** and *Point2* is **(R*COS(20),R*SIN(20))** and so on – ideal for loops that go up in intervals of 10 degrees. If we join all these points together with straight lines, as in Figure IIa, you can see that the shape we obtain, though not a circle, is fairly close to

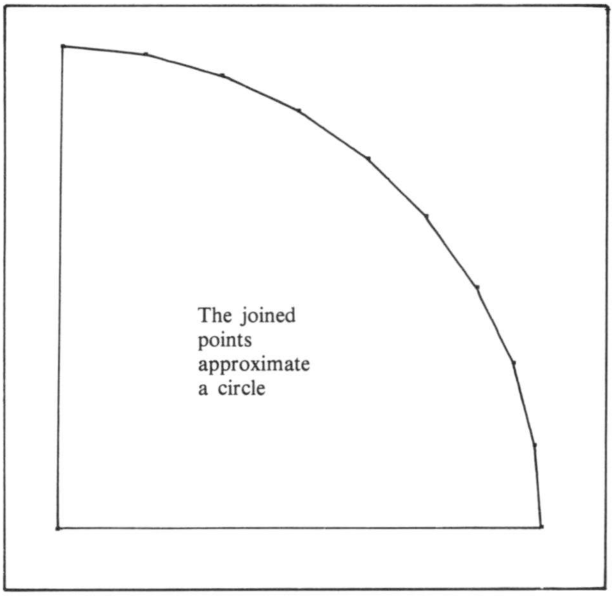


Figure IIa: Using a polygon to approximate a circle

one. If we had gone up in steps of five degrees the "fit" would have been even better.

This is how we draw circles. We calculate the points on the circumference using SIN( ) and COS( ), then join them with straight lines. Provided we choose our values properly, the approximation to a circle is close enough for most purposes.

Program I draws a quarter circle. Here, instead of R we've used *radius%* and set it equal to 1024 (line 30). Line 40 initialises the program by moving the graphics cursor out to the circumference of the circle. Instead of Θ we are using *angle%* and lines 50-80 form a loop in which *angle%* increases in steps of 10 degrees.

Unfortunately the BBC Micro measures angles not in degrees, but in radians. To use SIN and COS, the angle must be in radians. We don't have to worry about this since the micro provides us with a function to convert our usual degrees to radians, RAD( ). We use this in line 60 to change *angle%* in degrees to angle in radians. Note that we cannot use an integer variable for angles expressed in radians – they don't go in nice whole numbers!

```
    10 REM PROGRAM I
    20 MODE 4
    30 radius%= 1024
    40 MOVE radius%,0
    50 FOR  angle%= 0 TO 90 STEP 10
    60 angle=RAD(angle%)
    70 DRAW radius%*COS(angle),radius%
*SIN(angle)
    80 NEXT
```

The conversion done, line 70 then draws a line to the point specified by that angle.

*radius%**COS*(angle)* corresponds to R*COS(Θ)
*radius%**SIN*(angle)* corresponds to R*SIN(Θ)

```
    10 REM PROGRAM II
    20 MODE 4
    30 radius%= 1024
    40 MOVE radius%,0
    50 FOR  angle%= 0 TO 90 STEP 10
    60 angle=RAD(angle%)
    70 MOVE 0,0
    80 PLOT 85, radius%*COS(angle),rad
ius%*SIN(angle)
    90 NEXT
```

Program II again draws a quarter circle, this time filling in the triangles with PLOT85 (line 80). Notice the move back to the centre (line 70), so that we have three points for our triangle. Leave it out to see what happens. Perhaps you could alter the program to a four colour mode, and have each triangle plotted in a different colour. Also in both programs I and II, the loop variable can start at 10 rather than 0. Can you see why?

Quarter circles are all very well, but how do we manage full circles? If you cast your mind back once again to schooldays, you might remember that there are 360 degrees in a circle. There are 90 degrees in a quarter circle – a full circle is four times bigger, 360 degrees. All you have to do is to make the loop variable, *angle%,* go to 360 instead of 90. SIN( ) and COS( )

automatically take care of things for you.

Program III illustrates the technique. The actual calculation of points is identical with Program II. We have simply moved the origin to the centre of the screen (line 50), decreased the radius (line 30) and made the high loop

```
   10 REM PROGRAM III
   20 MODE 4
   30 radius%=512
   40 GCOL0,1
   50 VDU29,640;512;
   60 MOVE radius%,0
   70 FOR angle%=0 TO 360 STEP 10
   80 angle=RAD(angle%)
   90 MOVE0,0
  100 PLOT85,radius%*COS(angle),radiu
s%*SIN(angle)
  110 NEXT
```

parameter 360. Try altering the coarseness of the circle by changing the value of STEP in line 70. Keep to numbers that go into 360 evenly, such as 15,20,30 and 60.

As you increase the step, which reduces the number of points joined, you will begin to see that what appears as a circle is actually made up of a series of straight lines. Such a figure is known as a polygon. What value of step gives you the six-sided figure called a hexagon?

Program IV uses PROCcircle, a general circle drawing procedure with the following parameters: *xpos%* and *ypos%* give the coordinates of the centre of the circle, *radius%* gives the radius and *colour%* gives the logical colour number of the circle.

The way the procedure works is identical to Program III, except that at the end of the procedure the graphics origin is moved back to its original position (line 180) – a "tidying up" operation. The main body of the program (lines 10 to 70) calls the procedure four times for four decreasing radii (line 30).

Could you alter the program to, say Mode 2, and have circles of all the colours? How about giving the circles different origins? Finally, can you alter PROCcircle so that it just draws the circumference of the circle?

If you start to draw a circle and as you do so, gradually decrease the radius, what happens? It's not too hard to see that the edge of the "circle" starts to spiral inwards. This is the technique we use in Program V . The loop (lines 70-140) repeatedly calculates points on a "circle" (line 100), the radius of which is shrinking as *radius%* decreases (line 130).

Program VI produces the same spiral but then uses VDU19 to alter the assignments of logical colours so that the colours of the spiral's segments

```
   10 REM PROGRAM IV
   20 MODE 5
   30 FOR radius%=512 TO 512/4 STEP -
512/4
   40 colour%=colour%+1
   50 PROCcircle(640,512,radius%,colo
ur%)
   60 NEXT radius%
   70 END
   80 DEF PROCcircle(xpos%,ypos%,radi
us%,colour%)
   90 LOCAL angle%,angle
  100 GCOL0,colour%
  110 VDU29,xpos%;ypos%;
  120 MOVE radius%,0
  130 FOR angle%=0 TO 360 STEP 10
  140 angle=RAD(angle%)
  150 MOVE0,0
  160 PLOT85,radius%*COS(angle),radiu
s%*SIN(angle)
  170 NEXT
  180 VDU29,-xpos%;-ypos%;
  190 ENDPROC
```

appear to rotate. After a suitable delay each segment transfers its colour to its neighbour on one side while adopting the colour of its neighbour on the other side. Hence the colours gradually appear to move from segment to segment

```
   10 REM PROGRAM V
   20 MODE 2
   30 VDU 29,640;512;
   40 radius%=512:counter%=1
   50 angle%=18
   60 MOVE radius%,0
   70 REPEAT
   80 MOVE 0,0
   90 GCOL0,counter%
  100 PLOT 85,COS(RAD(angle%))*radius
%,SIN(RAD(angle%))*radius%
  110 counter%=counter% MOD 7+1
  120 angle%=angle%+18
  130 radius%=radius%-4
  140 UNTIL radius%<0
```

giving the illusion of rotation.

You might like to try a similar technique to make waves of colour flow across the circles of Program IV.

```
   10 REM PROGRAM VI
   20 MODE 2
   30 VDU 29,640;512;
   40 radius%=512:counter%=1
   50 angle%=18
   60 MOVE radius%,0
   70 REPEAT
   80 MOVE 0,0
   90 GCOL0,counter%
  100 PLOT 85,COS(RAD(angle%))*radius
%,SIN(RAD(angle%))*radius%
  110 counter%=counter% MOD 7+1
  120 angle%=angle%+18
  130 radius%=radius%-4
  140 UNTIL radius%<0
  150 REPEAT
  160 FOR loop%=1 TO 7
  170 VDU 19,loop%,(counter%+loop%)MO
D7+1,0,0,0
  180 NEXT loop%
  190 FOR wait%=1 TO 200:NEXT wait%
  200 counter%=counter%+1
  210 UNTIL counter%>200
  220 REPEAT
  230 FOR loop%=1 TO 7
  240 VDU 19,(counter%+loop%)MOD7+1,l
oop%,0,0,0
  250 NEXT loop%
  260 FOR wait%=1 TO 200:NEXT wait%
  270 counter%=counter%+1
  280 UNTIL FALSE
```

Chapter 7: Building Your Own Characters

Time for some practical work now. We are going to use the graphic methods we've already discussed, plus some new ones, to produce a Christmas tree with an appropriate greeting. It's never too early to start planning for Christmas!

The finished product is Program VII. Before that, though, we'd better look at a few of the techniques it uses, the most important of which concern user defined characters.

These characters, as their name implies, allow you to use the PRINT statement (or VDU) to print out your own special characters on the screen. In the BBC Micro's graphics modes, all the characters are drawn on an 8 × 8 grid, each cell of the grid being known as a pixel.

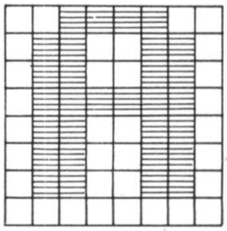


Figure I: Pixel pattern for the letter A

Figure I illustrates how the grid for the letter A is filled out. To print this letter by using its Ascii code simply enter **PRINT CHR$(65)** Now the BBC Micro has several blank grids ready for you to fill. These grids have the Ascii codes 224 to 255.

Once you've filled, say, 224 with the required pattern of cells, to print out this user defined character, you type **PRINT CHR$(224)** – and it appears where the cursor is, with the appropriate foreground and background colours.

Figure I illustrates how you can define your own character. You simply fill in the cells of the 8 × 8 grid until you get the pattern required. Then all that's needed is a few simple sums.

Each column of the grid has a value. The value of the rightmost column is 1, the one to its left 2, the one to the left of that 4, and so on, doubling each time until the leftmost column has a value of 128. You may have noticed that these are the column values of a binary number.

Each row of the grid has a number calculated for it and you add together

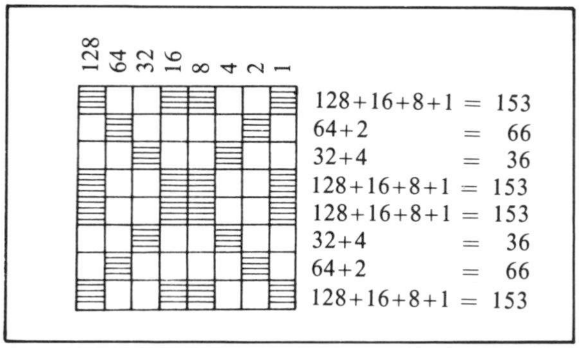


Figure II: User defined characters

the column values of each cell shaded in that row. Figure II shows the idea. Each row has its number. To teach the computer this pattern we use **VDU 23** followed by the Ascii code we're going to use for it followed by the eight numbers we've calculated for the rows, starting with the top and working down. All these are separated by commas.

So, to define character 224 to be the pattern shown in Figure II, we use **VDU 23,224,153,66,36,153,153,36,66,153** Program I makes use of this character. Try altering line 20 to **20 MODE 5** and notice the difference – the character is spread sideways. This is because the shape of the pixels in Mode 5 differs from that in Mode 4.

```
   10 REM PROGRAM I
   20 MODE 4
   30 VDU 23,224,153,66,36,153,153,36
,66,153
   40 FOR I%=0 TO 19
   50 PRINT TAB(I%,I%)CHR$(224)
   60 NEXT I%
```

A pixel is the smallest point you can plot on a screen. Its size limits the resolution, or fineness, available. As Figure III shows, in Modes 1 and 4, the

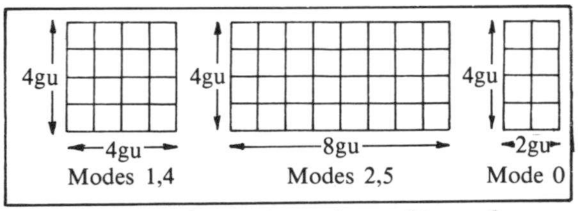


Figure III: Pixel sizes in each graphic mode

pixel is four graphic units (gu) square, whereas in Modes 2, 5 and 0 it is variously oblong.

So when you print out an 8 × 8 pixel grid in Mode 4 it comes out square, whereas in Mode 5 the wider pixels stretch the figure. Figure IV shows the size of a character in each graphic mode in terms of graphical units.

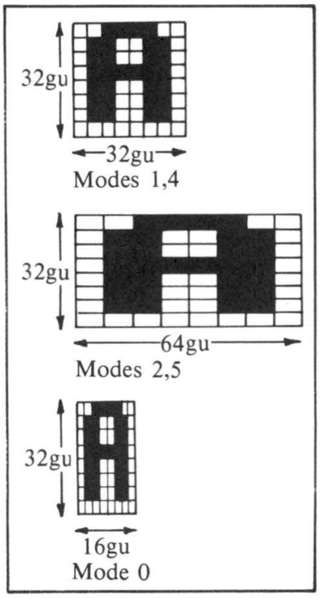


Figure IV: Character sizes in each graphic mode

You have to allow for this alteration in width when you are visualising your user defined character. I find it easier to see the character I want in terms of square pixels, so I've stuck to Mode 1 in Program VII even though it limits the range of available colours.

Let's try to print a pattern with our user defined character. We'll simply print the character three times, in a vertical line. We can use the cursor control codes to ensure that the characters actually end up vertically aligned.

VDU 8	moves the cursor left
VDU 9	moves the cursor right
VDU 10	moves the cursor down
VDU 11	moves the cursor up

Program II illustrates the technique.

Remember that we can string VDU statements together. Line 40 works like this: 224 prints the character, leaving the cursor directly to its right. 8 moves the cursor back left and 10 moves the cursor vertically down, leaving it directly under the first character. 224 prints the character again. 8 moves the

```
   10 REM PROGRAM II
   20 MODE 4
   30 VDU 23,224,153,66,36,153,153,36
,66,153
   40 VDU 224,8,10,224,8,10,224
```

cursor left and 10 moves the cursor down so that 224 prints the character vertically under the other two.

We can also use **PRINT CHR$(8)** to move the cursor left, and so on for the other cursor movements. Although slightly longer, this has the advantage that we can add all the characters, user defined and cursor movement, into one string and then simply print out that string. This results in the combined characters appearing far more quickly than with the VDU method. (See Program III.) We use this technique in Program VII. Lines 210 to 290 each add together the user defined characters required to make the various shapes. Lines 210, for example, defines a candle. Notice how I use the name *candle$* for the combined character string – it's far more meaningful than, say *shape1$*.

```
   10 REM PROGRAM III
   20 MODE 4
   30 VDU 23,224,153,66,36,153,153,36
,66,153
   40 character$=CHR$(224)+CHR$(8)+CH
R$(10)+CHR$(224)+CHR$(8)+CHR$(10)+CHR
$(224)
   50 PRINT character$
```

In Program VII the actual user defined characters are defined in lines 170 to 200, which read in the values for each VDU 23 from the DATA statements in lines 930 to 1060, 'VDUing' each parameter at a time (line 190). Notice how −1 is used in line 1060's DATA statement to mark the end of the parameters.

When we print a user defined character, it prints, as we have seen, in the currently defined foreground and background colour. Suppose, however, that we want a two-colour foreground, such that the centre pixels of our previous character and those in the middle of the outer edges are yellow, while the other pixels are red. (We'll leave the background black.)

You might think that we could do this by defining our character (224) as before and printing it in red, then overprinting it in yellow with another user defined character (225) containing only the pixels we want to be yellow. The red pixels should then show through. Figure V illustrates the idea. However, it

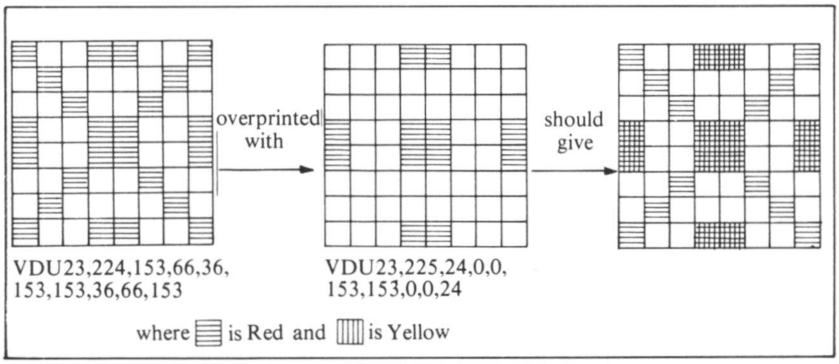


Figure V: Overprinting user defined characters

isn't as straightforward as this, as Program IV shows.

What happens is that the second character's background is printed, as well as its foreground, completely overwriting the first character! We need some way of writing just the foreground of the characters – making the background transparent.

```
   10 REM PROGRAM IV
   20 MODE 1
   30 VDU 23,224,153,66,36,153,153,36
,66,153
   40 VDU 23,225,24,0,0,153,153,0,0,24
   50 COLOUR 1
   60 PRINT TAB(19,15)CHR$(224)
   70 COLOUR 2
   80 PRINT TAB(19,15)CHR$(225)
```

We can do this by using the command VDU 5. After such a command, the micro only prints the foreground of characters. This works not only for user defined characters, but for the whole character set as well. There is one snag, though. VDU 5 also causes the characters to be printed not at the text cursor, but where the graphics cursor is.

Technically, we say that VDU 5 joins the text and graphics cursors. However, it's the graphics commands that are used to position the joined cursors – and to set colours – so I consider that VDU 5 causes characters to be printed at the graphics cursor.

This means that, after a VDU 5, to print a character at a certain location on the screen you have to place the graphics cursor there with the MOVE statement before PRINTing.

The numbers involved might take a bit more thought, but you gain two advantages. Firstly, you can position your character to the resolution of the screen, that is, within one pixel. With the TAB statement, you are limited to the eight pixel resolution of the character grid.

Secondly, of course, you only print the foreground of the character – there's no background to overwrite things. Another useful aspect of operations after VDU 5 is that the cursor movement characters still work. For example, CHR$(8) still moves the graphics/text cursor one whole character (eight pixels) left from its previous position. Try Program V.

```
   10 REM PROGRAM V
   20 MODE 1
   30 VDU 5
   40 VDU 23,224,153,66,36,153,153,36
,66,153
   50 VDU 23,225,24,0,0,153,153,0,0,2
4
   60 GCOL0,1
   70 MOVE 640,512:PRINT CHR$(224)
   80 GCOL0,2
   90 MOVE 640,512:PRINT CHR$(225)
  100 VDU 4
```

Notice the use of VDU 4 at the end – this turns off the VDU 5. You should always tidy up this way – leaving VDU 5 in operation can cause problems. Try listing a long program with VDU 5 on to see what I mean. Also the relevant colours are the graphics colours, not the text colours, so we use GCOL0 instead of COLOUR.

A word of warning. Don't use TAB with VDU 5 on. Although you can get away with it in OS 0.1, your program won't work when you upgrade to later systems.

Now run Program VI. This has the same effect as Program V. In this case we define a string to print out our multicoloured character (line 50). The latter might need some explanation. Instead of using GCOL0, 1 we can **PRINT CHR$(18) + CHR$(0) + CHR$(1)** where CHR$(18) is in place of GCOL, CHR$(0) is in place of 0 and CHR$(1) is in place of 1.

So the first four CHR$ statements of line 60 give the foreground colour 1 (red), followed by the user defined character (224). The next three CHR$ statements change the graphics colour to logical colour 2 (yellow), then CHR$(8) performs a backspace, ensuring that CHR$(225) overprints the first character.

We use this technique in Program VII to obtain our fairy. Line 270 adds together the string *fairy$*, consisting of its body. This is printed over *wfairy$* which is the fairy's body plus wings (PROCfairy, 710-780).

fairy$* and *wfairy$ each consists of nine user defined characters, in a 3 × 3 grid. Actually, two of the characters of *fairy$* (235 and 237) are completely blank. Since one variable overprints the other, I decided to keep the grid standard for easier handling.

```
   10 REM PROGRAM VI
   20 MODE 1
   30 VDU 5
   40 VDU 23,224,153,66,36,153,153,36
,66,153
   50 VDU 23,225,24,0,0,153,153,0,0,2
4
   60 character$=CHR$(18)+CHR$(0)+CHR
$(1)+CHR$(224)+CHR$(18)+CHR$(0)+CHR$(
2)+CHR$(8)+CHR$(225)
   70 MOVE 640,512
   80 PRINT character$
   90 VDU 4
  100 VDU 4
```

Similarly, we print a filled-in ball, *fball$*, followed by its outer rim, *uball$*, on top of it (PROCballs, 590-690).

The candles are printed out as a combined string *(candle$)* in PROCcandles (490-570). The actual tree shape is drawn by printing three green triangles, the apexes of which overlap (PROCtriangles, 320-390). The base of the tree is drawn in PROCbase (410-470).

PROCmessage simply prints out a seasonal greeting, using VDU 5 to give a sort of 3D effect.

```
   10 REM PROGRAM VII
   20 MODE 1
   30 VDU23;11,0;0;0;0
   40 PROCinit
   50 PROCtriangles
   60 PROCbase
   70 PROCcandles
   80 PROCballs
   90 PROCfairy
  100 PROCmessage
  110 END
  120 REM ============================
=
  130 DEF PROCinit
  140 VDU 19,3,2,0,0,0
  150 VDU 19,0,4,0,0,0
  160 centre%=640
  170 REPEAT
  180 READ vdu%
  190 IF vdu%<>-1 THEN VDU vdu%
  200 UNTIL vdu%=-1
  210 candle$=CHR$(18)+CHR$(0)+CHR$(1
)+CHR$(224)+CHR$(10)  +CHR$(8)+CHR$(22
5)+CHR$(8)+CHR$(10)+CHR$(225)+CHR$(18
)+CHR$(0)+CHR$(2)+CHR$(11)+CHR$(11)  +
CHR$(8)+CHR$(226)
  220 REM Unfilled ball
  230 uball$=CHR$(227)+CHR$(228)+CHR$
(10)+CHR$(8)+CHR$(8)+CHR$(229)+CHR$(2
30)
  240 REM Filled ball
  250 fball$=CHR$(231)+CHR$(232)+CHR$
(10)+CHR$(8)+CHR$(8)+CHR$(233)+CHR$(2
34)
  260 REM Fairy
  270 fairy$=CHR$(18)+CHR$(0)+CHR$(1)
+CHR$(235)+CHR$(236)+CHR$(237)+CHR$(1
0)+CHR$(8)+CHR$(8)+CHR$(8)+CHR$(238)+
CHR$(239)+CHR$(240)+CHR$(10)+CHR$(8)+
CHR$(8)+CHR$(8)+CHR$(241)+CHR$(242)+C
HR$(243)
  280 REM Fairy with wings
  290 wfairy$=CHR$(18)+CHR$(0)+CHR$(2
)+CHR$(244)+CHR$(245)+CHR$(246)+CHR$(
10)+CHR$(8)+CHR$(8)+CHR$(8)+CHR$(247)
+CHR$(248)+CHR$(249)+CHR$(10)+CHR$(8)
+CHR$(8)+CHR$(8)+CHR$(250)+CHR$(251)+
CHR$(252)
  300 ENDPROC
  310 REM ============================
=
  320 DEF PROCtriangles
  330 GCOL0,3
  340 FOR counter%=1 TO 3
  350 READ top%,offset%,bottom%
  360 MOVE centre%,top%:MOVEcentre%-o
ffset%,bottom%
  370 PLOT 85,centre%+offset%,bottom%
  380 NEXT counter%
  390 ENDPROC
  400 REM ============================
=
  410 DEF PROCbase
  420 GCOL0,1
  430 READ offset%,height%
  440 MOVE centre%+offset%,96:MOVE ce
ntre%-offset%,96
  450 PLOT 85,centre%+offset%,height%
  460 PLOT 85,centre%-offset%,height%
  470 ENDPROC
  480 REM ============================
=
  490 DEF PROCcandles
  500 VDU 5
  510 FOR counter%=1 TO 6
  520 READ xpos%,ypos%
  530 MOVE xpos%,ypos%
  540 PRINT candle$
  550 NEXT counter%
  560 VDU4
  570 ENDPROC
  580 REM ============================
=
  590 DEF PROCballs
  600 VDU 5
  610 FOR counter%=1 TO 6
  620 READ xpos%,ypos%,outer%,inner%
  630 GCOL0,inner%:MOVE xpos%,ypos%
  640 PRINT fball$
  650 GCOL0,outer%:MOVE xpos%,ypos%
  660 PRINT uball$
  670 NEXT
  680 VDU 4
  690 ENDPROC
  700 REM ============================
==
  710 DEF PROCfairy
  720 MOVE 596,1000
  730 VDU 5
  740 PRINT wfairy$
  750 MOVE 596,1000
  760 PRINT fairy$
  770 VDU 4
  780 ENDPROC
  790 REM ============================
  800 DEF PROCmessage
  810 VDU 5
  820 GCOL0,2
  830 FOR counter%=0 TO 16STEP 4
  840 MOVE 64-counter%,68-counter%
  850 PRINT "MERRY CHRISTMAS AND A HA
PPY NEW YEAR"
  860 NEXT counter%
  870 MOVE 64-counter%,68-counter%
  880 GCOL0,1
  890 PRINT "MERRY CHRISTMAS AND A HA
PPY NEW YEAR"
  900 VDU4
  910 ENDPROC
  920 REM ===========================
=
  930 REM Candle
  940 DATA 23,224,16,56,56,124,124,56
,16,56,23,225,56,56,56,56,56,56,56,56
  950 DATA 23,226,16,56,56,124,124,56
,16,0,23,255,16,56,56,124,124,56,0,0
  960 REM Unfilled Ball
  970 DATA 23,227,1,1,14,56,32,64,128
,128,23,228,128,128,112,28,4,2,1,1,23
,229,128,128,64,64,32,24,7,0,23,230,1
,1,3,2,4,24,224,0
  980 REM Filled Ball
  990 DATA 23,231,1,1,15,63,63,127,25
5,255,23,232,128,128,240,252,252,254,
255,255,23,233,255,255,127,127,63,31,
7,0,23,234,255,255,255,254,252,248,22
4,0
 1000 REM Fairy
 1010 DATA 23,235,0,0,0,0,0,0,0,0,23,
236,0,0,0,56,124,124,124,56,23,237,0,
0,0,0,0,0,0,0,23,238,0,0,1,3,7,0,0,0,
23,239,124,254,255,125,125,124,254,25
4
 1020 DATA 23,240,0,0,0,128,192,0,0,0
,23,241,1,1,3,3,7,0,0,0,23,242,255,25
5,255,255,255,0,0,0,23,243,0,0,128,12
8,192,0,0,0
 1030 REM Fairy plus Wings
 1040 DATA 23,244,0,32,48,56,60,60,62
,63,23,245,0,0,0,56,124,124,124,57,23
,246,0,8,24,56,120,120,248,248,23,247
,63,63,63,63,63,63,62,60
 1050 DATA 23,248,255,255,255,255,255
```

```
,255,254,126,23,249,248,248,248,248,2
48,248,248,120,23,250,61,57,51,35,7,0
,0,0,23,251,255,255,255,255,255,0,0,0
,23,252,120,56,152,136,192,0,0,0
 1060 DATA -1
 1070 REM Triangles Data
 1080 DATA 950,175,775,850,300,550,67
5,400,300
 1090 REM Base Data
 1100 DATA 50,296
 1110 REM Candle Positions
 1120 DATA 460,871,805,871,335,646,93
0,646,235,396,1030,396
 1130 REM Ball Positions
 1140 DATA 400,400,2,1,800,450,1,2,70
0,650,0,1,600,500,1,0,450,625,0,2,575
,750,2,0
```

Program VIII illustrates this technique. What it does is to print out 'A WORD' in yellow several times, each time moving it down four graphic units to the left and down (line 60). This is because, each time through the loop, the number we reduce the X and Y coordinates by, I%, is increased by 4. The reason we move four graphic units at a time is because this is the minimum resolution in Mode 1.

When you exit a loop the loop variable is still incremented before the check to see if the loop should be terminated is made. So, when line 90 is reached, I% is four greater than when the loop last printed 'A WORD'. Also the colour has changed to red (line 100) so line 110 prints out 'A WORD' in red, one pixel down and to the left of the last 'A WORD'.

Try to decipher Program VII – working through other people's programs is one of the best ways to improve your own programming.

```
 10 REM PROGRAM VIII
 20 MODE 1
 30 GCOL 0,2
 40 VDU 5
 50 FOR I%=0 TO 12 STEP 4
 60 MOVE 544-I%,512-I%
 70 PRINT"A WORD"
 80 NEXT I%
 90 MOVE 544-I%,512-I%
100 GCOL 0,1
110 PRINT"A WORD"
120 VDU4
```

Chapter 8: Combining Colours Logically

SO far we have used GCOL0 to choose our graphics colours. In fact, we've treated GCOL as if it always had a 0 after it. This isn't always so – we can follow GCOL directly with any numbers in the range 0 to 4. When we use GCOL0, the colour specified by the following logical colour number is "forcibly painted" onto the micro's screen, completely overpainting whatever was below.

However the other options, such as GCOL1, while still allowing you to specify a logical colour number, blend that colour with the colour already on the screen at the position you're plotting. The micro doesn't work on the old "blue + yellow = green" rule we learnt at school. Instead, the colour the computer "mixes" depends on how the two logical colour numbers involved (the one already on the screen, and that specified in the GCOL) combine logically.

The various GCOL options either OR, AND, EOR or NOT the numbers involved. GCOL1 specifies OR, GCOL2 specifies AND, GCOL3 specifies EOR and GCOL4 quite simply NOTs, or inverts, the colour at present on the screen, the logical colours you specify after the comma being a dummy. Try Program I.

```
 10 REM PROGRAM I
 20 MODE 5
 30 VDU 19,3,4,0,0,0
 40 GCOL0,1
 50 PROCbox(200)
 60 A$=GET$
 70 GCOL1,2
 80 PROCbox(200)
 90 END
100 DEF PROCbox(offset%)
110 MOVE 800+offset%,200
120 MOVE offset%,200
130 PLOT 85,800+offset%,800
140 PLOT 85,offset%,800
150 ENDPROC
```

Line 30 simply changes logical colour 3 to blue. Line 40 selects logical colour 1, in this case red, as foreground, while line 50 prints a red rectangle.

Line 60 holds the program until you press a key, then line 70 selects a new logical colour to reprint the box in.

Now it may look as if you're redrawing the box in logical colour 2, but notice you're using GCOL1,2 not GCOL0,2. The GCOL1 means that the BBC Micro will logically OR the present foreground colour with 2, and use the result as the foreground colour. As the previous logical colour number was 1, 1 OR 2 gives 3, or, in the rather clearer binary, %01 OR %10 gives %11. So the second time it is called PROCbox draws the rectangle with logical colour 3, in this case blue.

You're probably wondering what all the fuss is about. It seems a longwinded way of printing a blue rectangle. Why didn't I just use a GCOL0,3 followed by PROCbox? Well the point is that by doing things cleverly you can overprint one colour with another in such a way that the micro "remembers" the colour that was there first! Try Program II.

```
 10 REM PROGRAM II
 20 MODE 5
 30 VDU 19,3,4,0,0,0
 40 GCOL0,1
 50 PROCbox(0)
 60 GCOL1,2
 70 PROCbox(400)
 80 END
 90 DEF PROCbox(offset%)
100 MOVE 800+offset%,200
110 MOVE offset%,200
120 PLOT 85,800+offset%,800
130 PLOT 85,offset%,800
140 ENDPROC
```

Here we omit the pause before the second rectangle is drawn, and the two rectangles, though overlapping, are offset from each other. Amazingly, three rectangles appear on the screen. Figure I should help to make this clearer.

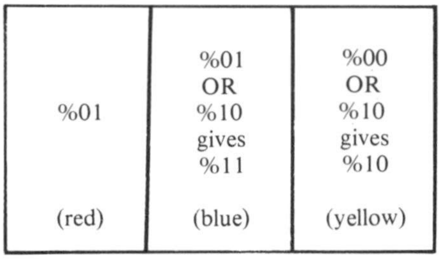


Figure I: The rectangles composing Program I

What happens is that the first rectangle is drawn normally in red (GCOL0,1). The second rectangle is drawn with GCOL1,2. This ORs logical colour 2 with the logical colour numbers underneath where it happens to be plotting.

In this case, the second rectangle is spread over two colours – the red of the first rectangle (%01) and the black of the background (%00). This gives two different rectangles since %01 OR %10 gives %11 (blue) and %00 OR %10 gives %10 (yellow).

```
 10 REM PROGRAM III
 20 MODE 5
 30 VDU 19,3,1,0,0,0
 40 GCOL1,1
 50 PROCbox(0)
 60 GCOL1,2
 70 PROCbox(400)
 80 END
 90 DEF PROCbox(offset%)
100 MOVE 800+offset%,200
110 MOVE offset%,200
120 PLOT 85,800+offset%,800
130 PLOT 85,offset%,800
140 ENDPROC
```

Program III is a variant of Program II. What we've done here is to use line 30 to redefine logical colour 3 to be actual colour 1 (red). This means that there are now two logical colour numbers for red (%01 and %11). Also we're ORing the first rectangle onto the screen – line 40 uses a GCOL1,1. As it is going on the background (%00) this doesn't make any effective difference. For the rest of this article we're going to be ORing on rather than using GCOL0.

If you've run Program III you'll have noticed that we get two rectangles, red and yellow. The red actually comprises the old red and blue rectangles combined. This is because the old blue rectangle is in colour %11 since we've ORed %01 with %10. As we've redefined that as red in line 30, the two rectangles merge into one red rectangle (actually consisting of two logical colour numbers).

One way to think about this arrangement is to imagine that the colour red is somehow "in front" of the colour yellow, so our original red rectangle blocks out or obscures the yellow rectangle where they overlap. You can consider the red to be the "foreground colour" and the yellow a "midground colour". (We've already got a background.) When you think like this, it's natural that the foreground should obscure the midground in this way.

When we think of logical colours as belonging to fore-, mid- and background we say we are dealing with "multi-plane images". Think of each colour as being drawn on separate screens, or planes, one on top of the other. The one on top is the foreground colour, which will then obscure the mid- and background colours underneath.

In Program III the red is drawn in the first plane and the yellow in the second, so the red overlaps the yellow – any yellow painted on the midplane "under" a red area will not show. So by using this system of ORing colours onto the screen, we can deal with overlapping images.

There's more to this than meets the eye. When you OR a foreground onto a midground colour the micro "remembers" that there was a midground colour there. So if the foreground "moves off" the midground colour shows through. You see when you OR logical colour 2 with logical colour 1, the result is logical colour 3 which, with a cunning VDU 19, we arranged to be the foreground colour.

The situation is easier to follow in binary: %01 and %11 are both the foreground colour. Notice that bit zero is set – that is, equal to one – when we are talking about a foreground colour. The other two colour numbers in a four colour mode, %00 and %10, which do not have this bit set, are the back- and midground colours respectively.

Now we are ORing the colours onto the graphics screen. So, the fact that bit one is set in %11 must mean that originally there was some yellow "underneath". After all, if you're just ORing the three logical colours %00, %01 and %10 how else can bit one be set without having %10 in on the deal? (We never use %11 when ORing onto the screen since we already have %01 to do the job – they're the same colour.)

So we can consider bit zero to be the "foreground bit" and bit one to be the "midground bit". If either bit is set on its own, the respective colour is shown. If both are set the foreground takes precedence, but the computer remembers (by keeping bit one set) that there was yellow there.

To recap, in the last program bit zero set shows that there's some red in the shape, and you're always bound to see it because it's foreground. Bit one set means that there is yellow in the shape – even if you can't see it because it's obscured by a foreground colour. Figure II summarises this.

Now I hinted previously that the foreground colour could "move off", leaving the midground showing through (if there was anything there). Our system can cope with this since it remembers when the foreground is obscuring the background.

You see by using another GCOL statement you can "turn off" or "clear to zero" the foreground colour bit in the logical colour number. After all, if you change %11 to %10, you'll be left with something in the midground colour. We can "clear" bits like this with the AND statement. We just AND the number we're working on with another binary number consisting of 1s in the bits we want to preserve, and 0s in the bits we wish to clear. So, to clear the foreground bit we want to AND the logical colour number with %10.

To show how this works consider the case of a foreground colour

Logical colour number		Interpretation
Binary	Decimal	
00	0	Background
01	1	Simple foreground
10	2	Simple midground
11	3	Foreground obscuring midground

Figure II: Multiplane images in a four colour mode

obscuring a background colour. As we've seen, the logical colour number for this is %11. Now %11 AND %10 gives %10 – that is, we've cleared the foreground bit leaving only the midground.

If there were no midground present – that is, if we had just a simple foreground (%01) – then %01 AND %10 gives %00, leaving just the background. So we can "turn off" the foreground by ANDing it with %10. GCOL2 allows us to do this sort of ANDing of logical colour numbers. It ANDs the logical colour number following it with that on the screen.

```
  10 REM PROGRAM IV
  20 MODE 5
  30 VDU 19,3,1,0,0,0
  40 GCOL1,1
  50 PROCbox(0)
  60 A$=GET$
  70 GCOL1,2
  80 PROCbox(400)
  90 A$=GET$
 100 GCOL2,2
 110 PROCbox(0)
 120 END
 130 DEF PROCbox(offset%)
 140 MOVE 800+offset%,200
 150 MOVE offset%,200
 160 PLOT 85,800+offset%,800
 170 PLOT 85,offset%,800
 180 ENDPROC
```

Program IV is identical to Program III in that it draws a red foreground rectangle, partly obscuring a yellow midground rectangle. However in lines 100 and 110 we AND the foreground rectangle with %10, clearing the foreground bit. This effectively "unprints" the foreground rectangle, revealing the part of the midground rectangle previously obscured. *(If you remember, the obscured part was in logical colour %11, so ANDing with %10 leaves you with %10, the midground colour.)*

Similarly, we can clear the midground colour. This time we AND the logical colour number with %01, the zero "killing" the required bit. To demonstrate this, add these lines to Program IV:

```
100 GCOL 2,1
110 PROCbox(400)
```

These changes ensure that we are ANDing the second rectangle with %01. When you run it the second rectangle disappears. Actually, the whole of the yellow midground rectangle goes – even though you can't see it behind the red foreground. Also, the red foreground that previously overlapped the yellow is now totally in logical colour %01. Before it was %11 – now we've ANDed it with %01 to give %01.

So, as our scheme has developed, we *OR to put shapes on the screen, AND to take shapes off the screen.* Although we have restricted ourselves to a four colour mode, this has given us three effective colours (one of which has two

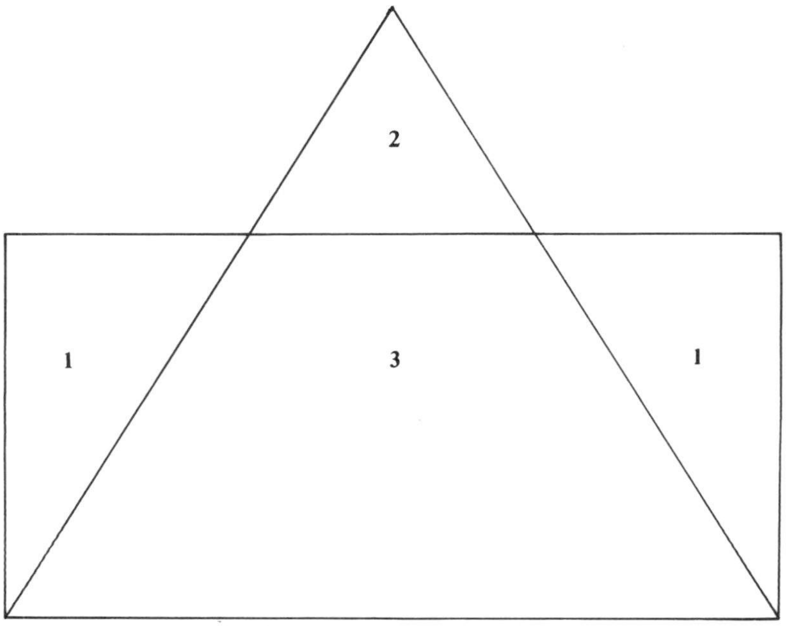


Figure III: Logical colours in Program IV

logical colour numbers), and the power to overlap and recover shapes.

Finally, we can use the ability to OR shapes to display two overlapping shapes separately and instantly on the same screen without overlapping them. Program V shows how. As Figure III demonstrates, we simply OR the rectangle onto the triangle. To display the triangle, we use VDU 19 statements to turn colours 2 and 3 "on" in the triangle colour, and to make colour 1 the

```
   10 REM PROGRAM V
   20 MODE 5
   30 VDU 19,3,1,0,0,0
   40 GCOL1,2:PROCtriangle(200):PROCw
ait
   50 GCOL1,1:PROCbox(200):PROCwait
   60 REPEAT
   70 PROCdisplay_triangle:PROCwait
   80 PROCdisplay_box:PROCwait
   90 UNTILFALSE
  100 DEF PROCbox(offset%)
  110 MOVE 800+offset%,200
  120 MOVE offset%,200
  130 PLOT 85,800+offset%,600
  140 PLOT 85,offset%,600
  150 ENDPROC
  160 DEF PROCtriangle(offset%)
  170 MOVE 800+offset%,200
  180 MOVE offset%,200
  190 PLOT 85,400+offset%,1000
  200 ENDPROC
  210 DEF PROCdisplay_box
  220 VDU19,1,1,0,0,0
  230 VDU19,2,0,0,0,0
  240 VDU19,3,1,0,0,0
  250 ENDPROC
  260 DEF PROCdisplay_triangle
  270 VDU19,1,1,0,0,0
  280 VDU19,2,0,0,0,0
  290 VDU19,3,1,0,0,0
  300 ENDPROC
  310 DEF PROCwait
  320 FOR loop%= 0 TO 2000
  330 NEXT loop%
  340 ENDPROC
```

background colour (PROCdisplay_triangle). Similarly, to display the rectangle, we turn 3 and 1 "on" and set 2 to background (PROCdisplay_rectangle).

Chapter 9: Using Exclusive OR and Invert

WE'LL go straight onto GCOL3 and GCOL4, two commands that many find baffling. To understand them properly we need to put them in the context of GCOL0, GCOL1 and GCOL2. Enter Program I. This has one procedure, PROCtriangle. We use it in a REPEAT . . . UNTIL loop (lines 30-70) to draw a triangle on the screen, as its name suggests. Line 60 serves to prevent the loop being repeated until you press a key, and line 40 chooses the GCOL option you draw the rectangle with.

If you've run the program you'll have found that repeating the loop doesn't seem to have any effect. This shouldn't surprise you – after all, with GCOL0,1 you are simply drawing the triangle in white on a black

```
   10 REM PROGRAM I
   20 MODE 4
   30 REPEAT
   40 GCOL0,1
   50 PROCtriangle(0, 0, 1279, 1023)
   60 delay$=GET$
   70 UNTIL FALSE
   80 DEF PROCtriangle( x%, y%, base%
, height%)
   90 MOVE x%, y% :MOVE x%+base%, y%
  100 PLOT 85, x%+base%/2, y%+height%
  110 ENDPROC
```

background, and in a fixed position. Repeating this will cause no visible change. Of course we could use other GCOL options. For instance, GCOL2,1 will AND logical foreground colour one onto the screen. To see this, replace line 40 with:

```
40 GCOL2, 1
```

and run the program. Nothing appears on the screen no matter how many times you press the key. When you think about it, this is correct. You *are* drawing the triangle, but as you are ANDing logical colour one onto background colour zero, you in fact plot colour zero since 1 AND 0 = 0. The next time through the loop you are again ANDing 1 onto 0 and so on . . .

Another option is to use GCOL1,0 to OR our graphics onto the screen. Change line 40 to:

```
40 GCOL1, 1
```

and run it. The triangle's back. This is because we are ORing colour one onto background zero, which gives us colour one since 1 OR 0 = 1. The next time through the loop we then OR colour one onto the colour one that is already there. Since 1 OR 1 = 1, the triangle is unchanged, and so on.

There are other GCOL options, however, that we have not covered. For example, with GCOL3 we can EOR graphics onto the screen. Replace line 40 with:

```
40 GCOL3, 1
```

and run the program. The triangle appears. However pressing the key causes the triangle to disappear. Press once more and it reappears. Press again, it's gone and so on.

Initially you EOR colour one onto background zero. This plots the triangle in colour one, since 1 EOR 0 is 1. The next time through the loop you EOR a triangle in colour one onto the triangle that is already there, also in colour one. These two now cancel each other out, since 1 EOR 1 = 0. Thus the triangle disappears, being in the background colour. Once more through the loop and you are EORing a triangle in colour one onto the triangle already present in colour zero. The triangle reappears, since 1 EOR 0 = 1. I think you can see how it continues . . .

There's another option we haven't tried – GCOL4. Use this version of line 40 in Program I:

```
40 GCOL4, 1
```

Here the 4 following the GCOL causes the micro to *invert* whatever colour is on the screen at the points drawn, or to use the proper terminology, plotted. To work out the inverse of a logical colour number simply subtract that number from the highest logical colour number available in that mode. Remembering that for a four colour mode the maximum is 3, not 4 – that is, one less than the number of colours available. In other words, to find the inverse of logical colour number x, in a two colour mode the inverse is 1–x, in a four colour mode the inverse is 3–x, and in a 16 colour mode the inverse is 15–x.

Referring to our latest version of Program I, we're in a two colour mode. When we clear the screen our background colour is zero, so since we're inverting with GCOL4 the colour in which our triangle appears in is 1 – 0 = 0. However when we press the space bar and plot it again the screen under the new triangle being plotted is in logical colour one, the colour of the previous triangle.

Now with GCOL4 it's the colour *under* the plotting that's important. Since it's in colour one we invert it, giving us a triangle drawn in 1 – 1 = 0. As our background's already zero the triangle disappears. The next time we draw it, however, we're plotting our new triangle onto an area that is all colour zero. As the inverse is 1 – 0 = 1, the triangle reappears.

Don't let the last two variants of Program I convince you that GCOL3 and GCOL4 are the same thing – they're not, although they often seem uncannily alike. Try Program II. This is another variant on Program I. All that's different is that it's in Mode 5, a four colour mode.

```
   10 REM PROGRAM II
   20 MODE 5
   30 REPEAT
   40 GCOL3,1
   50 PROCtriangle(0, 0, 1279, 1023)
   60 delay$=GET$
   70 UNTIL FALSE
   80 DEF PROCtriangle( x%, y%, base%
, height%)
   90 MOVE x%, y% :MOVE x%+base%, y%
  100 PLOT 85, x%+base%/2, y%+height%
  110 ENDPROC
```

This time the triangle appears in red since 0 EOR 1 = 1, and in this mode logical colour one is red (assuming there's been no "VDU19ing"). The next time through it disappears, and so on as before. If you now change line 40 to:

```
40 GCOL4, 1
```

the triangle is white when it appears. This is because it is plotting onto logical colour zero, the inverse of which, as we're in a four colour mode, is 3 – 0 = 3, which in Mode 5 is initially white. When we replot next time round the triangle goes since we're plotting onto an area of logical colour three, the inverse of which is zero (3 – 3 = 0). Round the loop once more, of course, and it's back.

Try adding the line:

```
25 GCOL0, 130 : CLG
```

to give a yellow background. Can you see what's going on? Two clues:

2 EOR 1 = 3

3 – 2 = 1

The point is that GCOL3 and GCOL4 aren't identical. They are similar though in this respect – do each of them twice and you're back to the beginning. Just as in a double negative the two negatives cancel each other out, so if you EOR a number with another number twice, or invert a number twice, you get back to the original number. That's why the triangles kept coming and going.

Let's look at EOR first. If we have 3 and we EOR it with 2 we get: **3 EOR 2 = 1.** If we EOR the result with 2 yet again we obtain: **1 EOR 2 = 3** and our number – or colour – is back. It always works this way.

It might be easier to see in binary:

```
    %11
EOR %10
    ---
    %01
EOR %10
    ---
    %11
```

Let's now have a look at inverting in a four colour mode. Choose the number 2. Now the inverse of 2 is 3 – 2 = 1. And if we then invert our answer (1) we obtain 3 – 1 = 2, the number we started with.

We can use these techniques to give us a sneaky sort of animation.

Suppose we want to move a man across a background. We simply print him in the first position using GCOL3, or GCOL4. To then wipe him out, we simply print him again in the same place, since EORing or inverting twice restores the status quo. We then MOVE onto another position. We are, of course, under the influence of VDU 5, printing at the graphics cursor.

```
   10 REM PROGRAM III
   20 MODE 5
   30 VDU 23,240,28,28,8,127,8,20,34,
65
   40 VDU 5
   50 GCOL 3,1
   60 GCOL 0,129: CLG
   70 REPEAT
   80 FOR move%= 64 TO 1216 STEP 32
   90 MOVE move%,512
  100 VDU 240
  110 FOR wait%=0 TO 500: NEXT wait%
  120 MOVE move%,512
  130 VDU 240
  140 NEXT move%
  150 UNTIL 0
```

Program III does this in Mode 5. Line 30 defines character 240 to be a little man, line 40 does the required VDU 5, line 50 causes us to PLOT in colour one under EOR and line 60 ensures a red background. The animation is done by lines 80-140, a FOR . . . NEXT loop. The REPEAT . . . UNTIL loop of lines 70, 150 simply keeps the animation cycling across the screen.

move% determines the X coordinate to print the man at. It increases each time through the loop. Firstly – lines 90,100 – the loop prints the man at *(move%* ,512). It then pauses so you can see him (line 110). Finally lines 120, 130 print him again at the same position. Remember, though, that he has been EOR'd on. So the second printing on the same spot undoes the first and he disappears, ready to move on to a different position next time round the loop. To show how vital this second printing is try leaving out line 130.

Now change line 50 to:

```
50 GCOL4, 1
```

so that we are inverting onto the screen. This time when you run it you see much the same thing happening, although the man will have changed colour. Do the sums and you'll see why. Also try using GCOL4 with numbers other than 1 – say 0, 2 and 3. As you'll see, there's no effect. This is because the second figure is used in GCOL to specify the current foreground colour. However with GCOL4 the only colour taken into account is that being plotted onto, so the second figure is irrelevant. Don't leave it out though. The machine expects it, even if it is a dummy!

One subject beloved of graphics programmers is curve-stitching. Personally, after the first dozen or so programs, I find them as boring as the real thing . . . Still, they do tend to illustrate the uses of GCOL well, so I make no excuse for Program IV.

```
 10 REM PROGRAM IV
 20 MODE 1
 30 VDU 23;8202;0;0;0;
 40 FOR offset%=0 TO 1024 STEP 16
 50 GCOL3,RND(3)
 60 MOVE offset%,0
 70 DRAW 0, offset%
 80 DRAW offset%,1023
 90 DRAW 1023, offset%
100 DRAW offset%,0
110 NEXT offset%
120 wait$=GET$
```

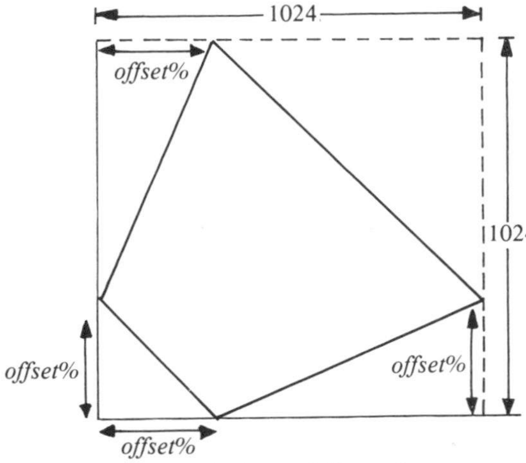


Figure I: Curve-stitch quadrilateral

All I'm doing is repeatedly drawing a quadrilateral of the type shown in Figure I, offset increasing from 0 in steps of 16 (lines 40, 110). Line 50 is the crux:

```
50 GCOL3, RND(3)
```

Here we are EORing random colours onto the screen, giving us rather attractive effects. As an experiment, see what would happen if you used GCOL4 instead as in:

```
50 GCOL4, RND(3)
```

Is that what you expected? Finally, try GCOL0, 1 and 2 in line 50. Can you predict the results beforehand?

Chapter 10: Rubber Banding Revealed

REMEMBER how GCOL3 and GCOL4 work: GCOL3 Exclusive ORs (EOR) the foreground colour specified with that already on the screen, while GCOL4 simply inverts the colour that is already there. In different circumstances they will have different effects, but there is one property they have in common – when under the influence of either, if you perform the same DRAW or PLOT twice, the second time cancels out the first.

In practice we tend to use this property to erase lines in drawing programs – computer aided design (CAD), as the jargon has it. We simply draw the line we want with, say, GCOL3,1 and if it's not what we desire we rub it out by drawing exactly the same line still with GCOL3,1.

```
 10 REM PROGRAM I
 20 MODE 4
 30 GCOL0,1
 40 MOVE 1000. 0 :MOVE 200, 0
 50 PLOT 85, 1000, 1000
 60 PLOT 85, 200, 1000
 70 REPEAT
 80 FOR height% = 1 TO 4
 90 GCOL3,1
100 MOVE 0, 200 * height%
110 DRAW 1279, 200 * height%
120 NEXT height%
130 wait$=GET$
140 UNTIL 0
```

Program I illustrates the idea. It simply draws a large rectangle (lines 40 to 60) and then rules four lines across it, using GCOL3,1 – that is it EORs logical colour one onto the rectangle (lines 80 to 120). Now these lines are in a REPEAT . . . UNTIL loop (lines 70 to 140) but after initially drawing the lines the loop is prevented by the GET$ in line 130 from being repeated until a key is pressed. When the lines are first drawn they are in logical colour number zero (1 EOR 1 = 0) where they cross the rectangle, so it appears to have been cut into pieces.

Having pressed a key though the lines are once more drawn across the rectangle. This time the segments that cross the rectangle are being EORed onto colour zero – the result of the last "drawing". So now the parts of the lines across the rectangle appear in logical colour one, since 1 EOR 0 = 1. The effect of this is to give us a whole rectangle. Next time round we are once more EORing logical colour one onto itself, and we regain our divided rectangle. I think you'll be able to predict without too much difficulty the effect of changing line 90 to:

```
90 GCOL4, 1
```

However bear in mind that GCOL3 and GCOL4 are not identical. To illustrate this, try Program II:

```
 10 REM PROGRAM II
 20 MODE 5
 30 GCOL0,1
 40 MOVE 1000, 0 :MOVE 200, 0
 50 PLOT 85, 1000, 1000
 60 PLOT 85, 200, 1000
 70 REPEAT
 80 FOR height% = 1 TO 4
 90 GCOL3,(height%-1)
100 MOVE 0, 200 * height%
110 DRAW 1279, 200 * height%
120 NEXT height%
130 wait$=GET$
140 UNTIL 0
```

See if you can predict the outcome before you run it. It's virtually identical to Program I, save that we're in Mode 5 (line 20). We take advantage of the change in mode by varying the colours of our line. The formula *height%–1* in line 90 ensures that the colours of the lines EORed onto the screen are logical colour numbers 0, 1, 2 then 3.

Can you visualise what happens? Suppose we change line 90 to:

```
90 GCOL4, (height% - 1)
```

Can you guess what would happen now? Try it and see. The main point is that GCOL3 and GCOL4 are different in their effects. If anything, GCOL3 gives you the more control, since you can "mix" the colour you specify with the colour already there. Whereas with GCOL4 you are simply inverting the colour already on the screen.

A good application of the ability of these two GCOL statements to "self-erase" is when you have a pointer rotating around a dial. Look at Program III:

```
   10 REM PROGRAM III
   20 MODE 4
   30 radius%=400
   40 VDU 29,640;512;
   50 MOVE radius%,0
   60 FOR angle = 0 TO 360 STEP 10
   70 DRAW radius%*COS(RAD(angle)),ra
dius%*SIN(RAD(angle))
   80 NEXT
   90 angle=0
  100 GCOL3,1
  110 REPEAT
  120 IF angle < 350 THEN angle=angle
+10 ELSE angle=0
  130 MOVE 0,0
  140 DRAW radius%*COS(RAD(angle)),ra
dius%*SIN(RAD(angle))
  150 FOR wait%=0 TO 400 :NEXT
  160 MOVE 0,0
  170 DRAW radius%*COS(RAD(angle)),ra
dius%*SIN(RAD(angle))
  180 UNTIL 0
```

If when you're entering it you initially end with line 80 and run the program, you'll see that the first few lines simply draw a circle. Go through it carefully until you can see how it works. Briefly, it draws a 36 sided polygon – which is close enough to a circle for our purposes. The method is fully explained in Chapter Six. Line 40 moves the graphics origin to the centre of the screen.

If you now type in the rest of the program and run it you see that a pointer appears, rotating anti-clockwise like the hands of an indisciplined clock. What we're doing is drawing a radius from the centre of the circle to the circumference (lines 130 to 140). We then wait for a while with a dummy loop (line 150), so we can see the line.

Next we redraw it from the centre to the circumference – but as we're under GCOL3 (line 100) this simply unplots our previous line. Now this whole draw, undraw routine is in a REPEAT . . . UNTIL loop (lines 110 to 180). However each time through the loop line 120 intervenes to increase *angle* – that is, how far round the "face" the arm is – by 10 degrees. Once we've plotted for 350 degrees though, we restore *angle* to 0 degrees, since 0 degrees is the same as 360 degrees, where our next increment of 10 degrees would take us (still line 120). After all, getting back to where we started is the whole point of going round in circles! Again, try it with GCOL4 in place of GCOL3.

One of the nice things about this type of plotting is that we can use the techniques to pass over "background" objects and leave them totally unchanged. To see this, add the following line to Program III:

```
85 MOVE 50, -100: MOVE 50, 100: PLOT
85, 200, -100
```

These techniques are ideal for the scanners of your intergalactic battle cruiser, aren't they? As a challenge, why not tinker with the above to turn the program into a clock?

One of the most useful applications of these techniques is in "rubber banding". This is just a way of drawing lines on your screen that allows you to "slide and stretch" them back and forth until you get them to fit exactly where you want.

Program IV draws a line from the centre of the screen to a cross-shaped cursor defined in line 30. Now you can shift this cursor about the screen by using the cursor keys. The *FX4,1 of line 80 allows them to return Ascii values.

The position of the cursor is determined by *newx* and *newy*. As you can see, lines 290 to 320 of PROCinput vary their values depending on the key pressed (line 250). PROCinput is in a REPEAT ... UNTIL loop so let's see what happens. Remembering we're using GCOL3 (line 100). Line 240 draws the cursor. The –8 and +16 offsets are to ensure that the centre of the cross is at *newx, newy*. That is, *newx–8, newy+16* are the top left coordinates of the user-defined character printed by that line's VDU 244 (notice we're under VDU 5 – line 90). Line 250 then waits for a key to be pressed. Once it has been pressed line 260 reprints the cross in the same position – but because of the GCOL3 it disappears, ready for us to move it in accordance with the cursor key that's been "got". PROCline of 270 simply draws a line from *oldx, oldy* – which in this program never vary from the centre of the screen – to *newx, newy*.

This may seem a bit odd, since we've not taken into account the cursor's "move", but in practise this will be the deleting line, going over a line we've already drawn. At the beginning, *oldx, oldy* and *newx, newy* coincide, so in effect no line is drawn. Lines 290 to 320 calculate the cursor's new position and ensure it doesn't go off screen. The PROCline of 330 then draws the line from the centre of the screen to the circle. The next time through the procedure, after the key press, line 270 redraws – and hence wipes out – the last line. After the adjustments of 290-320, line 330 draws the new line.

The effect has to be seen to be appreciated. As you direct the cursor round

```
   10 REM PROGRAM IV
   20 MODE 0
   30 VDU 23,224,24,24,24,255,255,24,
24,24
   40 oldx=640: oldy=512
   50 newx=640: newy=512
   60 MOVE oldx,oldy
   70 fix=FALSE
   80 *FX4,1
   90 VDU 5
  100 GCOL3,1
  110 REPEAT
  120 PROCinput
  130 UNTIL fix
  140 *FX4,0
  150 VDU4
  160 END
  170 REM ===========================
  180 DEF PROCline
  190 MOVE oldx,oldy
  200 DRAW newx,newy
  210 ENDPROC
  220 REM ===========================
  230 DEF PROCinput
  240 MOVE newx-8,newy+16:VDU 224
  250 key=GET
  260 MOVE newx-8,newy+16:VDU 224
  270 PROCline
  280 IF key=135 THEN fix=TRUE
  290 IF key=136 THEN IF newx>15 THEN
 newx=newx-16
  300 IF key=137 THEN IF newx<1263 TH
EN newx=newx+16
  310 IF key=138 THEN IF newy>15 THEN
 newy=newy-16
  320 IF key=139 THEN IF newy<1007 TH
EN newy=newy+16
  330 PROCline
  340 ENDPROC
```

the screen a seemingly elastic line constantly joins it to the centre – hence the term "rubber banding". If we should wish to freeze our line in one position we just press the Copy key. Line 280 then sets the logical variable *fix* to TRUE, which causes us to drop out of the REPEAT . . . UNTIL (line 130).

You may be wondering what happens if you press other keys – after all PROCinput doesn't test for the legality of keys. If you look carefully you'll see that all that happens is that 270 erases the line and then 330 draws it back in the same place. Lines 140 and 150 simply restore the cursor keys and cursor movement to the status quo ante program.

There's far more potential than this though. Suppose that when we'd fixed our line we then started a new line from the last position of the cursor. Then when we'd fixed that we could start a new line from the end of that and so on . . . we'd have a computer etch-a-sketch! It's not all that difficult. Program V has what's needed:

```
   10 REM PROGRAM V
   20 MODE 0
   30 VDU 23,224,24,24,24,255,255,24,
24,24
   40 oldx=640: oldy=512
   50 newx=640: newy=512
   60 MOVE oldx,oldy
   70 fix=FALSE: end =FALSE
   80 *FX4,1
   90 VDU 5
  100 GCOL3,1
  105 REPEAT
  110 REPEAT
  120 PROCinput
  130 UNTIL fix OR end
  135 GCOL0,1: PROCline: GCOL3,1
  136 oldx=newx: oldy=newy: fix=FALSE
  137 UNTIL end
  140 *FX4,0
  150 VDU4
  160 END
  170 REM ==========================
  180 DEF PROCline
  190 MOVE oldx,oldy
  200 DRAW newx,newy
  210 ENDPROC
  220 REM ==========================
  230 DEF PROCinput
  240 MOVE newx-8,newy+16:VDU 224
  250 key=GET
  260 MOVE newx-8,newy+16:VDU 224
  270 PROCline
  280 IF key=135 THEN fix=TRUE
  290 IF key=136 THEN IF newx>15 THEN
 newx=newx-16
  300 IF key=137 THEN IF newx<1263 TH
EN newx=newx+16
  310 IF key=138 THEN IF newy>15 THEN
 newy=newy-16
  320 IF key=139 THEN IF newy<1007 TH
EN newy=newy+16
  325 IF key=65 THEN end=TRUE
  330 PROCline
  340 ENDPROC
```

It's much the same as Program IV – the new lines have the new line numbers. In essence we're repeating the main body of Program IV. Lines 105 and 137 form the new REPEAT . . . UNTIL loop. This time we won't want to finish the program every time we fix a line, so we have another logical variable, *end.*

If you compare Programs IV and V you see that the first difference between them is in the UNTIL of line 130 which has the extra condition OR *end* attached. Let's suppose we've just started the program and have moved our cursor to a position where we want to fix a line. So far everything has gone on as in Program IV. Once we press Copy though we drop out of the inner REPEAT . . .UNTIL loop (lines 110 to 130). We then change to the "standard" GCOL0 to draw the line and then revert to GCOL3 (line 135).

Line 136 contains the clever bit. We then move the starting point of our new line to the end of the line we've fixed by setting *oldx* equal to *newx* and *oldy* equal to *newy*. Next time we'll start rubber banding from the "free" end of the last line. Notice also that we set *fix* to FALSE again. If we didn't we'd just keep dropping through the inner loop, never getting anywhere. Leave it out and see! Line 137 tests to see if *end* is TRUE, dropping through to finish if so. Line 325 in PROCinput tests for the ending condition – pressing A for "abort".

Even with this rudimentary program you can have a lot of fun, but there are limitations. For instance, every line follows on from the end of the preceding one. It would be much more convenient if we could turn off the line until we'd moved the cursor into a new position. How else could we, for example, draw a box within a box? And of course, no one's perfect – we might want to delete a line.

Program VI incorporates these refinements. To do so we've introduced two new logical variables, *line* and *delete.* (Notice that the terminating conditions for our inner loop – line 70 – now involve *delete.)* If *line* is TRUE, when the cursor moves a line rubber bands to it. If it is false, the cursor moves with no attached line. We use the Return key to toggle between the two states (line 460). Line 470 then either draws the line or sets the cursor coordinates to its new position as appropriate. The actual drawing of a "fixed" line is done by line 80.

You will probably have noticed that, when using these techniques, if you draw a line exactly over another line they disappear. This is because of the GCOL3 effect of which we are taking advantage. As soon as you move the top line off however, the line underneath reappears. If you really did want to erase this line all you would have to do is press "D", which then sets delete to TRUE (line 450). You will then drop out of the inner loop and line 90 will draw the line in the background colour – that is erase it – with GCOL0,0, then restore GCOL3,1. However you came out of the inner loop lines 100 to 110 move the start coordinates to their new position and line 120 restores the logical variables *fix* and *delete* to their default values.

The program, though short, is by no means a simple one, so persevere – the ideas behind it are very important.

```
   10 REM PROGRAM VI
   20 MODE 0
   30 PROCinitialise
   40 REPEAT
   50 REPEAT
   60 PROCinput
   70 UNTIL fix OR end OR delete
   80 IF fix AND line THEN GCOL0,1:PR
OCline:GCOL3,1
   90 IF delete THEN GCOL0,0:PROCline
:GCOL3,1
  100 oldx=newx
  110 oldy=newy
  120 fix=FALSE:delete=FALSE
  130 UNTIL end
  140 *FX4,0
  150 VDU 4
  160 END
  170 REM ===========================
  180 DEF PROCinitialise
  190 VDU 23,224,24,24,24,255,255,24,
24,24
  200 end=FALSE: line=FALSE :fix=FALS
E :delete=FALSE
  210 oldx=640:oldy=512
  220 newx=640:newy=512
  230 MOVE oldx,oldy
  240 *FX4,1
  250 VDU 5
  260 GCOL3,1
  270 ENDPROC
  280 REM ===========================
  290 DEF PROCline
  300 MOVE oldx,oldy
  310 PLOT13, newx,newy
  320 ENDPROC
  330 REM ===========================
  340 DEF PROCinput
  350 MOVE newx-8,newy+16:VDU 224
  360 key=GET
  370 MOVE newx-8,newy+16:VDU 224
  380 IF line THEN PROCline
  390 IF key=135 THEN fix=TRUE
  400 IF key=136 THEN IF newx>15 THEN
 newx=newx-16
  410 IF key=137 THEN IF newx<1263 TH
EN newx=newx+16
  420 IF key=138 THEN IF newy>15 THEN
 newy=newy-16
  430 IF key=139 THEN IF newy<1007 TH
EN newy=newy+16
  440 IF key=65 THEN end=TRUE
  450 IF key=68 THEN delete=TRUE
  460 IF key=13 THEN line=NOT line
  470 IF line THEN PROCline  ELSE old
x=newx:oldy=newy
  480 ENDPROC
```

Chapter 11: Teletext Tales

ONE of the most useful features of the BBC Micro is Mode 7, the teletext mode. This is the one the micro enters automatically when you switch it on or press the Break key. It's unlike any of the other modes of the micro and is in fact controlled by a special chip used only for teletext. Because of this, the way it is used differs completely from the way we use the other graphics modes.

The beauty of Mode 7 is that it can be used to produce colourful, interesting screen displays which don't use up large chunks of precious memory. In fact, you can get a whole screenful of teletext using just 1k of memory. This leaves you some 30k to use for your program.

Having said that, the graphics you can have in Mode 7 aren't all that advanced. But if you look at the BBC's Ceefax or ITV's Oracle you'll see that impressive displays can be produced.

As you'll probably know, the Mode 7 screen consists of 25 lines from top to bottom. Each can hold 40 characters across. Figure I shows the layout of the screen: Notice that the first line is numbered 0, the last 24. It's the same with the character spaces, which are numbered from 0 to 39, not from 1 to 40 as might be expected.

Normally when we use Mode 7 all we get is white text on a black background. These are the default colours when we enter the mode. For

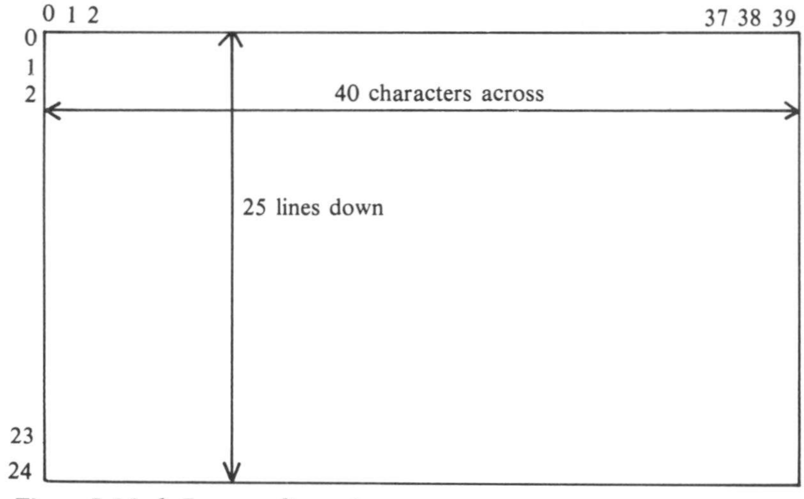


Figure I: Mode 7 screen dimensions

listings this is all right, but if we want to have a bit of colour in our Mode 7 displays, we have to tell the BBC Micro which colours we want. We do this by using some beasties called control characters. These are special characters we put on the screen to determine how the output appears for the rest of that line.

We print them on the screen just as we would the character "A" or "1". The difference is that after we've printed a control character we don't see it. A shy creature, it's just there to control the colour of the text, not to be seen itself. To print "Hello" on the screen we would normally use a line like: **10 PRINT "Hello"**. This would put the word up on screen in black and white. But suppose we wanted it in red? What we would do is print a special control character in front of the "Hello", telling the micro that whatever comes next in that line is to appear in red. The control character that produces red is the character with the number 129, so: **10 PRINT CHR$(129) "Hello"** will print the string "Hello" in red.

Figure II shows how the line is made up on screen. Notice that the control code takes up a space even though you don't see it. It blends in with the background.

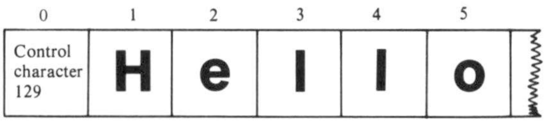


Figure II: Printing using a control character

Seven colours are available in Mode 7, each one with its own control character. Run Program I and you'll see them in action: Colourful aren't they? The FOR . . . NEXT loop has printed out a different control character at the beginning of each line bringing about different coloured text each time. These control character codes are shown in Figure III.

Remember that although we can't see it, the control character at the beginning of each line does take up one character space as demonstrated in

Number	Colour
129	Red
130	Green
131	Yellow
132	Blue
133	Magenta
134	Cyan
135	White

Figure III: Control codes

```
   10 REM PROGRAM I
   20 MODE 7
   30 FOR code = 129 TO 135
   40 PRINT CHR$(code) "COLOUR CODE "
;code
   50 NEXT
```

Program II. This produces the same display as Program I with the added row of numbers at the top. Notice that the character space 0 appears empty on

```
   10 REM PROGRAM II
   20 MODE 7
   30 PRINT"0123456789"
   40 FOR code = 129 TO 135
   50 PRINT CHR$(code) "COLOUR CODE "
;code
   60 NEXT
```

each line. It's not actually empty, there's an invisible control character there which determines the colour for the rest of the line.

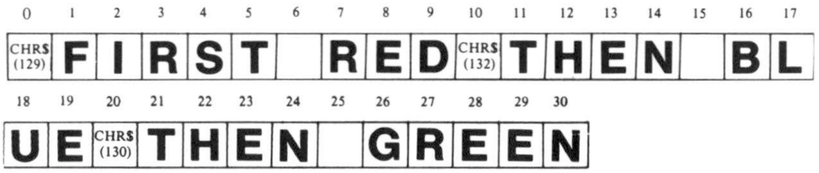
You can have more than one colour of text on a line as Program III shows. If you look at line 30 you'll see how this is done. First of all the control

```
   10 REM PROGRAM III
   20 MODE 7
   30 PRINT CHR$(129) "FIRST RED" CHR
$(132) "THEN BLUE" CHR$(130) "THEN GR
EEN"
```

character 129 is printed. We don't see this, but it decides the colour of whatever comes next in the line. Hence FIRST RED appears in red.

After this we print another control character, this time the one that produces blue. This overrides the previous command to print in red and so THEN BLUE appears in blue. Not content with having only two colours on the line, there comes the control character for green and the last string appears in green.

Thus we can have more than one colour on a line by using more than one



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
CHR$ (129) F I R S T R E D CHR$ (132) T H E N B L
18 19 20 21 22 23 24 25 26 27 28 29 30
U E CHR$ (130) T H E N G R E E N

Figure IV: Multicoloured line layout

control character in that line. Each control character's influence lasts until either the end of that line or until it meets another control character which takes over command. You'll notice that each control character still takes up one space, leaving gaps in the text. Figure IV shows what's happened.

As I said earlier, the control characters only have influence to the end of the line they appear on, or until another one takes over command. When you finish one line and start printing on the next the text will be in the default colour, white, unless you give the micro a different instruction by using another control character.

```
   10 REM PROGRAM IV
   20 MODE 7
   30 PRINT CHR$(131) "THIS WORKS ON
ONE LINE"
   40 PRINT"BUT NOT ON THE NEXT"
```

Program IV shows this in action. The first line is in yellow because of the control character 131. However, the influence of 131 only lasts until the end of that particular line. When the program comes to print the next line it finds no control character telling it what to do so it prints the text in white. If you want yellow text in the second line you have to put in the appropriate control code at the beginning as in program V.

```
   10 REM PROGRAM V
   20 MODE 7
   30 PRINT CHR$(131) "THIS WORKS ON
ONE LINE"
   40 PRINT CHR$(131)"AND ON THIS LIN
E, NOW"
```

But what about the background, must it always be black? No, the control characters can be used to change the colour of the background. Program VI shows how this is done. The secret lies in control character 157. This tells the micro that the previous control character on that line (in this case control character 130) is to be the background colour. So in line 30 the CHR$(130) selects the colour green and the following CHR$(157) tells the computer that

```
   10 REM PROGRAM VI
   20 MODE 7
   30 PRINT CHR$(130) CHR$(157) CHR$(
129) "RED LETTERS ON A GREEN BACKGOUN
D"
```

this is to be the background colour. Then CHR$(129) tells the micro that the foreground colour is to be red and so we get the red letters on a green background.

One thing to notice is that we have to set the foreground to a different colour to the background after using a CHR$(157). Otherwise the micro uses the same foreground and background colours with unimpressive results. Try leaving out the CHR$(129) in line 30 and you'll see what I mean. As green lines go, it's very nice, but it doesn't tell you much!

Run Program VI again and notice that we've now used three control characters, each of which takes up one space. Program VII shows this. The CHR$(130) has taken up one space (and is invisible against the black

```
    10 REM PROGRAM VII
    20 MODE 7
    25 PRINT"01234567890123456789"
    30 PRINT CHR$(130) CHR$(157) CHR$(
129) "RED LETTERS ON A GREEN BACKGOUN
D"
```

background). The background now changes to green and the remaining two colour codes take up two more spaces, this time matching the green background. The result is that the red text doesn't start until three spaces in from the side. Things like this have to be kept in mind when you plan your Mode 7 screen layouts.

The same control characters are used to get the background colours as are used to get the foreground colours. The only difference is the following CHR$(157) which sets the background to the colour of the previous foreground code. Program VIII shows the available background colours.

```
10 REM PROGRAM VIII
20 MODE 7
25 PRINT"01234567890123456789"
30 FOR code = 129 TO 135
40 PRINT CHR$(code) CHR$(157)
50 NEXT
```

You might notice that there is no control character for black, the default background colour. If we've changed background colour and want to revert to black again all we have to do is use CHR$(156) which switches to the black background again.

As you might guess from the above, we can change background colours in mid-line just as we can the foreground text colours. Program IX shows this in action. Line 30 does all the work. The first CHR$(157) switches the background colour to red, as determined by the preceding CHR$(129). The next CHR$(157) in that line has the control character CHR$(131) in front of it and hence the background is yellow. The control characters, although invisibly blending with the background colours, still take up room, as Program IX illustrates.

```
   10 REM PROGRAM IX
   20 MODE 7
   25 PRINT"0123456789012345678901234
56789"
   30 PRINTCHR$(129) CHR$(157) CHR$(1
31) "ONE BACKGROUND" CHR$(131) CHR$(1
57) CHR$(132) "ANOTHER BACKGROUND"
```

So far we've covered the use of control characters to change both the foreground and background colours. If you've seen any teletext displays at all then you probably won't be surprised to learn that there's a control character that will make the text flash. This is control character 136. If you put a CHR$(136) in a line then the text after it will flash, the foreground colour alternating between itself and the background colour. Program X shows how

```
   10 REM PROGRAM X
   20 MODE 7
   25 PRINT"0123456789012345678901234
56789"
   30 PRINT CHR$(136) CHR$(129) CHR$(
157) CHR$(131) "ONE BACKGROUND" CHR$(
131) CHR$(157) CHR$(132) "ANOTHER BAC
KGROUND"
```

it works. The effect can be turned off with a CHR$(137). Program XI shows both in action. Notice the amount of space used by the invisible control characters.

```
   10 REM PROGRAM XI
   20 MODE 7
   30 PRINT CHR$(136) CHR$(129) CHR$(
157) CHR$(131) "FLASHING" CHR$(137) C
HR$(131) CHR$(157) CHR$(132) "NON FLA
SHING"
```

As well as flashing text, another useful feature of Mode 7 teletext displays is that of double height characters. Using the control character 141 makes all the subsequent text in that line appear as double height. Program XII makes use of this. Line 40 is exactly the same as line 30. If it wasn't we would only get the top half of our double height text. Try leaving out line 30 or 40 when you run the program and you'll see what I mean.

```
10 REM PROGRAM XII
20 MODE 7
30 PRINTCHR$(141)"DOUBLE HEIGHT"
40 PRINTCHR$(141)"DOUBLE HEIGHT"
```

Of course the other control characters can be used with the double height letters. This is shown in Program XIII which produces large red text:

```
   10 REM PROGRAM XIII
   20 MODE 7
   30 PRINT CHR$(141) CHR$(129) "DOUB
LE HEIGHT"
   40 PRINT CHR$(141) CHR$(129) "DOUB
LE HEIGHT"
```

Just as we can use CHR$(141) to turn on the double height effect, so there is a control code which returns the micro to normal height text. This is CHR$(140) which is seen in action in Program XIV showing how double height and normal text can be mixed.

```
   10 REM PROGRAM XIV
   20 MODE 7
   30 PRINT CHR$(141) CHR$(129) "DOUB
LE HEIGHT" CHR$(140)"NORMAL HEIGHT"
   40 PRINT CHR$(141) CHR$(129) "DOUB
LE HEIGHT"
```

As you might expect, you can use CHR$(136) to make the large characters flash, as in Program XV.

```
   10 REM PROGRAM XV
   20 MODE 7
   30 PRINT CHR$(141) CHR$(136) CHR$(
129) "DOUBLE HEIGHT"
   40 PRINT CHR$(141) CHR$(136) CHR$(
129) "DOUBLE HEIGHT"
```

You can even use different background colours as with normal sized text (Program XVI).

```
   10 REM PROGRAM XVI
   20 MODE 7
   25 PRINT"01234567890123456789"
   30 PRINT CHR$(141) CHR$(136) CHR$(
130) CHR$(157) CHR$(129) "DOUBLE HEIG
HT"
   40 PRINT CHR$(141) CHR$(136) CHR$(
130) CHR$(157) CHR$(129) "DOUBLE HEIG
HT"
```

So we've covered the basics of using the teletext mode. By now you should be able to use Mode 7 to produce colourful and pleasing displays of your own, using large and flashing letters to enliven the screen. And we haven't even touched Mode 7 graphics, a void I propose to fill in the next chapter.

Finally, after making you type in all those CHR$, I'll inform you that you can use the VDU commands to pass the control characters to the micro. Program XVII, which has exactly the same result as the previous program, shows how it's done.

```
   10 REM PROGRAM XVII
   20 MODE 7
   30 VDU 141,136,130,157,129:PRINT"D
OUBLE HEIGHT"
   40 VDU 141,136,130,157,129:PRINT"D
OUBLE HEIGHT"
```

Easy, isn't it?

Chapter 12: Teletext Tales Revisited

NOW we'll go into using Mode 7's graphics capabilities. Although rather crude compared with the BBC Micro's other modes, Mode 7 can produce some entertaining displays, as a glance at Ceefax or Oracle will show. As we might expect from our earlier experiences, we access these graphics capabilities by making use of control codes. These tell the micro that instead of text it is to display block graphics. These are rectangles made up of six squares, as shown in Figure I. Each block is the size of a capital letter.

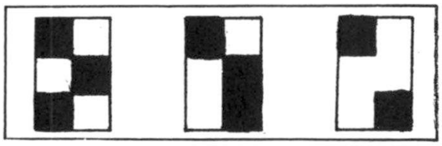


Figure I: Block graphics characters

Any of the six squares making up a block can be either on (that is, the foreground colour is shown) or off (the square is the background colour). Lighting various combinations of these squares allows the micro to have available 64 different graphics characters. Figure II shows the shapes on offer.

When we want the micro to produce block graphics we use one of the control codes shown in Table I. As you can see from this, seven codes are available. If we wanted to have red block graphics we would use **PRINT CHR$ (145)** in just the same way as previously we used **PRINT CHR$(129)** to get red text. However we have 64 block graphics characters to choose from. We have to tell the micro not only that we want block graphics and, at

Code number	**Graphics colour**
145	red
146	green
147	yellow
148	blue
149	magenta
150	cyan
151	white

Table I: Block graphics codes

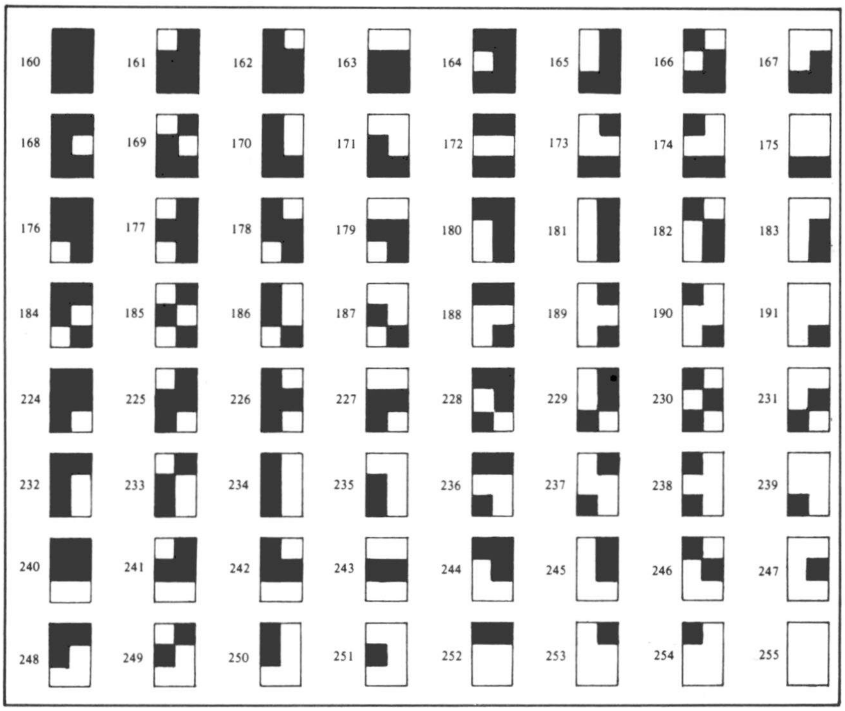


Figure II: Graphics characters and their codes

the same time, their colour, but also which character we want. This is done by using another set of control codes, those from 160 to 191 and 224 to 255.

The control codes and the character they refer to are shown in Figure II. Run Program I and you'll see them on the screen. The CHR$(145) picks red graphics while the FOR . . . NEXT loops decide which blocks appear. These shapes can be combined in many different ways to produce quite effective displays. Use PRINT and TAB along with Figure II and Table I to try your hand at Mode 7 graphics. It's not hard, you just have to remember where on

```
10 REM PROGRAM I
20 FOR code= 160 TO 191
30 PRINT code,CHR$(145)CHR$(code)
40 PRINT
50 NEXT code
60 FOR code= 224 TO 255
70 PRINT code,CHR$(145)CHR$(code)
80 PRINT
90 NEXT code
```

```
   10 REM PROGRAM II
   20 PRINT"DIFFERENT BACKGROUND COLO
UR"
   30 PRINT CHR$(130)CHR$(157)CHR$(14
8)CHR$(230)
   40 PRINT"FLASHING"
   50 PRINT CHR$(130)CHR$(157)CHR$(13
6)CHR$(148)CHR$(230)
   60 PRINT"DOUBLE HEIGHT"
   70 PRINT CHR$(130)CHR$(157)CHR$(13
6)CHR$(148)CHR$(141)CHR$(230)
   80 PRINT CHR$(130)CHR$(157)CHR$(13
6)CHR$(148)CHR$(141)CHR$(230)
```

the screen each of the characters has to go and that the control codes, although invisible, take up a space.

You'll be pleased to know that the special effects control codes also work with graphics characters. Program II shows them in action. It's not always easy to remember what the code for a particular character is. Of course they're all shown in Figure II and on page 488 of the User Guide, but these may not always be available.

Happily there is a way of calculating the control code of a character. Each of the squares that makes a Mode 7 graphics character has a value, as shown in Figure III. All you do when you want to know a character's code is to decide which squares are going to be in the foreground colour. Then total the values of those squares and add 160. The result is the control code for that character. Although it sounds complicated, it's quite simple in practice. Certainly I find it a lot easier than looking through the lists when the character I want invariably seems to hide. Figure IV shows a sample calculation.

As you might guess, we can use combinations of the various graphics blocks to produce something akin to user defined characters. If we wanted, for reasons best known to ourselves, to produce a cyan figure 8 three lines deep then we could use Program III. Here the CHR$(150) tells the micro that

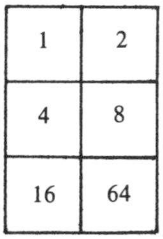


Figure III: Square values

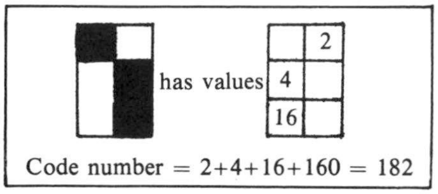


Figure IV: Calculating codes

```
    10 REM PROGRAM III
    20 PRINT CHR$(150)CHR$(183)CHR$(23
5)
    30 PRINT CHR$(150)CHR$(189)CHR$(23
8)
    40 PRINT CHR$(150)CHR$(245)CHR$(25
0)
```

we want cyan block graphics while the other six codes define which shapes are to be used. Try varying the program with your own choice of character shapes.

Now after having given you Figure II, which lists the shapes available and their codes, I'll tell you that there's a different way of getting the blocks. If you follow one of the graphics control codes with a string made up of lower case letters, numbers or punctuation marks you'll get graphics characters, not the letters or symbols you might expect. Program IV shows this in action for each of the lower case letters of the alphabet.

The full set of symbols and the graphics characters they give are shown in Figure V. Using this method we could produce our large 8 – magenta this time – in a much shorter program. Program V shows how this is done. We could even replace three lines with one by using the control codes 10 and 8 to move

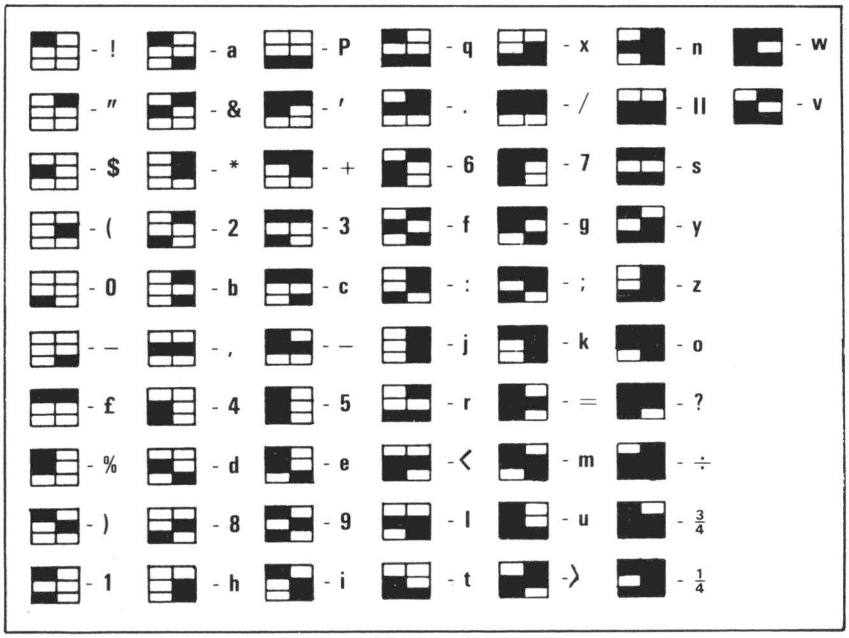


Figure V: Symbols for graphics blocks

```
    10 REM PROGRAM IV
    20 FOR x=1 TO 26
    30 alphabet$="abcdefghijklmnopqrst
uvwxyz"
    40 string$=MID$(alphabet$,x,1)
    50 PRINT CHR$(148);string$ CHR$(12
9)string$
    60 PRINT
    70 NEXT
```

```
10 REM PROGRAM V
20 PRINT CHR$(149)"7k"
30 PRINT CHR$(149)"=n"
40 PRINT CHR$(149)"uz"
```

the print cursor one space down and one space left respectively. This is shown in Program VI.

```
    10 REM PROGRAM VI
    20 PRINT CHR$(149)"7k"CHR$(10)CHR$
(8)CHR$(8)CHR$(8)CHR$(149)"=n"CHR$(10
)CHR$(8)CHR$(8)CHR$(8)CHR$(149)"uz"
```

You'll notice that the whole thing is getting more and more like a user defined character, and you may be wondering if we could replace all those CHR$s with a string of VDUs as we did at the end of the last chapter. The answer is yes, we can – and Program VII shows how it's done. It's much neater, but for the rest of the programs I'll stick to using CHR$. I think it makes things a little clearer.

```
    10 REM PROGRAM VII
    20 VDU 149,183,235,10,8,8,8,149,18
9,238,10,8,8,8,149,245,250,10
```

If you're wondering what that last control code does, it just puts the prompt on the next line. Try leaving it out and see what happens. Because the 149 code is still working on that line, the prompt is interpreted as a block graphics character with ugly results. Before you read on, just make sure that you understand what we've covered above. Have a go at producing your own

Mode 7 shapes on screen, using any of the methods outlined above. You'll find that, with a little practice, it's quite easy to do.

Having got so far, now's the time to tell you that we've only dealt with one of the two forms of block graphics available in Mode 7. We've been using contiguous block graphics. We could also have used something called separated graphics. Program VIII, a variant of Program IV shows the difference.

```
   10 REM PROGRAM VIII
   20 FOR x=1 TO 26
   30 alphabet$="abcdefghijklmnopqrst
uvwxyz"
   40 string$=MID$(alphabet$,x,1)
   50 PRINT "CONTIGUOUS SEPARATE"
   60 PRINT CHR$(148);string$,CHR$(15
4);string$
   70 NEXT
```

As you can see, contiguous graphics have all the squares that make them up joined together. Separated graphics have all the six squares which make up that character separated by the background colour. It's as though there's a border around each square, and consequently between adjoining characters on the same line. You get the "skeleton" of the block.

Contiguous block graphics are the default – the micro will use them unless you tell it differently. You can switch to separated graphics by using the control code 154 and back to contiguous graphics with control code 153. Which you use depends on what effect you want. At times separated graphics look much neater than contiguous ones. Try out some of the earlier programs using CHR$(154) to get separated graphics and see what you think.

While you've been playing around with Mode 7 you may have come up against the fact that the control codes, although invisible, take up a space on the screen. Though this doesn't matter too much in text, it can mess up graphics displays. You don't want gaps all over your beautiful artwork. Run Program IX and you'll see what I mean. The program gives the 20 shapes required, but that gap in the middle spoils the effect. That's where the control

```
 10 REM PROGRAM IX
 20 PRINT CHR$(147);
 30 FOR repeat=1 TO 10
 40 PRINT CHR$(247);
 50 NEXT repeat
 60 PRINT CHR$(148);
 70 FOR repeat=1 TO 10
 80 PRINT CHR$(251);
 90 NEXT repeat
100 PRINT
```

code 148 is hiding.

You might have wondered if there was a way round this effect and there is, using yet another set of control codes. Control code 158 has the effect of holding the graphics. What this means is that when you use that code in a line the micro will carry on printing out the required block graphics characters. However when it comes to another control code, instead of leaving a space as normal it fills that space in. It does this with the same graphics character as the last one it used. So by using the hold graphics code the ugly gap of Program IX can be avoided. Program X shows how this is done.

Eagle-eyed readers will have spotted that, although the lines are joined, there are 12 of the first character and 10 of the second. The control spaces

```
 10 REM PROGRAM X
 20 PRINT CHR$(147);
 30 FOR repeat=1 TO 10
 40 PRINT CHR$(247);
 50 NEXT repeat
 60 PRINT CHR$(158)CHR$(148);
 70 FOR repeat=1 TO 10
 80 PRINT CHR$(251);
 90 NEXT repeat
100 PRINT
```

have been filled in with the previous block graphics character. This holding effect can be switched off. The control code 159 returns the rest of the line to the normal way of things, where control codes appear as spaces.

And that's nearly the end, apart from one last control code, 152. This conceals the display. Anything that comes in a line after a 152 is automatically the background colour. This, of course, means that you can't see the rest of the line. Printing something on the screen that you can't see may sound a bit daft but it does have its point. By printing a message on the screen invisibly and then overprinting the 152 control code with a space you can make things appear as if by magic. In a way, it's the Mode 7 version of VDU

Control code	Result
152	Conceal display
153	Contiguous graphics
154	Separated graphics
158	Hold graphics
159	Release graphics

Table II: Graphics control codes

19. Program XI shows you the conceal display control code in action. You don't see the messages until the code is overwritten with a space.

```
    10 REM PROGRAM XI
    20 CLS
    30 PRINT TAB(1,11)CHR$(130)CHR$(15
2)"GREEN"
    40 PRINT TAB(1,12)CHR$(131)CHR$(15
2)"YELLOW"
    50 PRINT TAB(1,13)CHR$(132)CHR$(15
2)"BLUE"
    60 PRINT TAB(1,23)"PRESS SPACE FOR
 NEXT COLOUR"
    70 FOR X=11 TO 13
    80 WAIT$=GET$
    90 PRINT TAB(2,X)" "
   100 NEXT
```

And that's the end of our tour of Mode 7. Table II sums up these last control codes. As you can see, the basics of the mode are quite simple, but the effects obtained using the control codes can be complicated. And the great advantage is that you aren't using up lots of precious memory. Have fun!

Chapter 13: SOUND – Basic Principles

ALL the various sounds and noises that come from your micro while playing games have probably given you some idea of the scope of its sound-producing abilities. These range from derogatory noises when you lose a game, through simple tunes and even onto imitating musical instruments. Yet they come from a single speaker.

All the marvellous sound effects are produced with just two Basic commands, SOUND and ENVELOPE. It's the skilful use of these that gives the BBC Micro its musical abilities. We'll deal with both as my tale unfolds, but for the time being let's start with the simplest, the SOUND command.

This can be viewed as a single keyword, SOUND, followed by four numbers separated by commas. The SOUND keyword tells the micro to make a noise. The numbers decide how long the noise will last and what it will sound like. The structure of the command is **SOUND W,X,Y,Z** where W, X, Y and Z represent the figures that are used to control what sort of sound is actually going to be produced.

We'll deal with each of these parameters in turn, but first let's make a noise. After all that's what all this is about. Type the following into your micro: **SOUND 1,–15,53,20.** Now press the RETURN key and you should get a single note. It'll last for one second, be as loud as your micro can make it and, for the musically inclined, will be middle C in pitch (more or less). Don't worry how it's done for the moment. Just try it out and see that the SOUND command, with those four figures after it, does produce a note. Later we'll be playing around with this basic note in order to give examples of how the SOUND command can be controlled by these four numbers.

Now let's deal with each of them in turn and see how they affect it. Remember the structure is: **SOUND W,X,Y,Z.** The figure we put in the place of the W decides which channel will be used to produce the noise that the SOUND keyword tells the micro to make.

The BBC Micro's sound generator has four channels, each of which can produce a note simultaneously. This means that you can have four notes playing at the same time, one on each channel. Actually doing that lies in the future. For the time being let's just stick with producing one note at a time. As I've said, there are four channels and they can be selected by making W equal to 0, 1, 2 or 3. Let's ignore channel 0, the "special effects" one, and just deal with channels 1, 2 and 3. Any of these can be used to produce a note.

Program I has each of them in turn making the noise we made before. The SOUND command is just the same except that each time round the FOR . . . NEXT loop the channel number, W, varies. At first it's 1 (for channel 1 to be

```
10 REM PROGRAM I
20 FOR channel=1 TO 3
30 SOUND channel,-15,53,20
40 NEXT channel
```

```
10 REM PROGRAM II
20 SOUND 1,-15,53,20
30 SOUND 2,-15,69,20
40 SOUND 3,-15,85,20
```

```
10 REM PROGRAM III
20 FOR loudness = -15 TO 0
30 SOUND 1,loudness,53,20
40 FOR x =1 TO 3000:NEXT x
50 NEXT loudness
```

used), and then 2 (for channel 2) and 3 (for channel 3).

This, however, may not convince you that three separate channels have been used, as it just sounds like one note. Well, then, for the Doubting Thomases among you there's always Program II, which produces a different note on each of the three channels at the same time – giving us, in fact, a chord. You might notice that we have selected each channel by the figure we've placed in the W position. The different notes have been produced by varying the Y parameter, of which more later.

The next parameter to consider is X. The number we place in this position can vary between 0 and –15 and determines how loud the note will be. Strangely enough the loudest note is produced by –15, the quietest by –1. Putting 0 in the place of X produces silence. This can be very useful as sometimes you may need silence to separate two notes. In fact 0 loudness is used in a later example.

Program III produces the original note again, the loop varying the value of X so that the note gets quieter each time around. Don't worry too much about line 40. It's just there to slow things down a little and produce an appreciable gap between the notes. Notice how the sound seems to fade away into the distance. You've got your first sound effect!

Leaving the parameters that select the channel and the loudness, we'll move onto the figures that go in the place of Y in the SOUND command. It's these that determine the pitch of the note produced. Pitch is just a term used to describe whether a note is high or low. A note that is high in pitch will have you squeaking if you try to sing it. One that is low in pitch will have you "singing in your boots", as my old music master used to say.

It's easy to vary the pitch of the notes produced by replacing Y with a figure

```
10 REM PROGRAM IV
20 FOR pitch=0 TO 255
30 SOUND 1,-15,pitch,20
40 NEXT pitch
```

```
10 REM PROGRAM V
20 FOR pitch=0 TO 255 STEP 4
30 SOUND 1,-15,pitch,20
40 NEXT pitch
```

```
10 REM PROGRAM VI
20 FOR duration=0 TO 254
30 SOUND 1,-15,53,duration
40 SOUND 1,0,53,duration
50 NEXT duration
```

between 0 and 255. The lowest note you can get on the micro is with Y equal to 0, the highest with Y equal to 255. If you can stand it, Program IV will demonstrate the notes available, using a loop to vary the Y parameter from 0, the lowest note, to 255, the highest. It lasts over four minutes, the note slowly creeping upwards, so it's no disgrace to press Escape! Incidentally, you might like to reverse the loop to get a descending series of notes. I leave that to you.

You might find Program V more bearable. This does the same thing, only the notes go upwards in steps of four, technically known as semi-tones. These sound much more natural to our ears. More about them later.

The final parameter is Z, which determines the length of each note. This can vary from 0 to 254 in value. The figure 20 in the place of Z gives a note of one second's duration. You could say that each unit was worth one twentieth of a second. If you put 200 in the duration parameter you would get a note lasting 10 seconds. If, for peverse reasons of your own, you make Z either –1 or 255 then the note will continue to sound indefinitely. Try it – you can always press Escape when you've had enough. It makes Program IV seem interesting by comparison!

Program VI again uses a loop, this time to vary the length of the note from nothing to 12.7 seconds. You'll notice that by having 0 in the X position, line 40 produces a sound that has zero loudness. This is just to provide a gap between the example notes. Notice how it gives the impression of something slowing down.

So using these four parameters we can vary the notes produced by the SOUND command. Program VII does just this, using a loop to produce a series of varying notes. Each of the four parameters W, X, Y and Z, controlling channel, loudness, pitch, and duration respectively, are set randomly. The result is "computer music", strangely soothing if listened to for a while.

Try varying the ranges of the RND expressions to see what happens to the type of "music" produced. It's great fun, and you can learn a lot about the SOUND command by playing around.

```
   10 REM PROGRAM VII
   20 REPEAT
   30 pitch=RND(255)
   40 loudness=-1*RND(16)+1
   50 duration=RND(255)-1
   60 channel=RND(3)
   70 SOUND channel,loudness,
pitch,duration
   80 UNTIL FALSE
```

Chapter 14: Matters of Note

NOW for musical scales. Not that I intend to give you a music lesson, I wouldn't dare, but I will show you the basics of producing simple tunes with your micro. In the last chapter we examined the SOUND command and learnt how to use it to produce noises. Each of these separate noises is technically called a note and by putting together a series of notes we get a tune.

It should be pointed out that some series of notes are more tuneful than others – as you may have found with the random music generator! This is because the human ear has come to expect a certain consistency in the notes that make up tunes. It wants them to go up and down, or change pitch, by regular amounts.

Try running Program I. This goes upwards in a series of steps called semitones. It does this by adding four to the pitch parameter of the SOUND command each time round the loop. These semitones are the natural building blocks of western music. When tunes go up and down in pitch (higher or lower, soprano or bass) they tend to do so in semitones or groups of semitones. Even though Program I will probably have you pressing Escape before its end, you'll find it a lot more acceptable than Program II which goes up in steps of one, a quarter of a semitone.

Since we've been talking about semitones you might expect that we'll be coming to whole tones and this we do with Program III. This plays a series of notes, each successive note being raised in pitch by one tone (or two semitones). So increasing the pitch parameter by four raises the note by a semitone; increasing it by eight produces a note one whole tone higher. Higher

```
10 REM PROGRAM I
20 FOR pitch=0 TO 255 STEP4
30 SOUND 1,-15,pitch,20
40 SOUND 1,0,pitch,20
50 NEXT pitch
```

```
10 REM PROGRAM II
20 FOR pitch=0 TO 255 STEP 1
30 SOUND 1,-15,pitch,20
40 SOUND 1,0,pitch,20
50 NEXT pitch
```

```
10 REM PROGRAM III
20 FOR pitch=0 TO 255 STEP8
30 SOUND 1,-15,pitch,20
40 SOUND 1,0,pitch,20
50 NEXT pitch
```

```
10 REM PROGRAM IV
20 pitch=50
30 REPEAT
40 SOUND 1,-15,pitch,20
50 READ increase
60 pitch=pitch+increase
70 UNTIL increase=0
80 DATA 8,8,4,8,8,8,4,0
```

notes require different combinations of four and eight to be added to the pitch parameter.

The trouble with the previous programs is that although they worked in tones and semitones, the natural building blocks of music, they were boring. They leave you with that "waiting for the other boot" feeling! This is because our ears not only expect tunes to be composed of notes which vary in pitch by tones and semitones, but also they prefer certain selections of the available notes. Try running Program IV. Much more satisfying, isn't it? It has a complete feeling about it.

After the first note is played the following notes are increased in pitch by the amounts shown in the DATA statement. As you'll expect by now, these increments are all fours and eights, but they are in an order that sounds pleasing to the ear. This is known as a scale, and most tunes are made up of combinations of notes picked from one scale.

To recap, nearly all the tunes you know consist of series of notes whose pitch changes in multiples of tones and semitones. The notes go up and down in discrete bundles made up of these measures and most tunes confine themselves to a selection of the available notes. What this means for the BBC Micro is that if you're trying to write a tune you know that after you've picked the start note all the other notes will have the start note's pitch parameter varied by factors of four or eight.

Of course the channel, loudness and duration of the note may vary as well but in this chapter we'll concentrate on the pitch. It's much easier to use in practice than to describe. Try running Program V. This produces a fairly uninspiring version of a well-known tune. It's hardly wonderful music but you can see from the DATA statement how the pitch parameter of the notes varies by factors of four or eight from the pitch parameter of the first note.

Once I had the first note it was just a matter of figuring out whether the next

```
10 REM PROGRAM V
20 REPEAT
30 READ note
40 IF note=0 THEN END
50 SOUND 1,-15,note,10
60 UNTIL FALSE
70 DATA 30,30,38,26,30,38,0
```

note was up or down and adding or subtracting the fours and eights as necessary. Then I just put them in a DATA statement and let the loop read the pitch and play the tune. You can play your own tunes by putting in your own DATA statements. The problem is that you have to figure out which DATA statements produce which notes. Program VI will help you with this. It is a very simple one which allows you to write micro music by ear.

It begins by asking you for the pitch and duration parameters of the tune's first note. The micro then plays this and asks you if you want to keep it. If you do, it will save the note and ask you if you want to play the tune. The program carries on like this, asking for the parameters of notes, playing them and, if required, saving them. When you finally decide to hear the tune it will play it and display the DATA statement values used.

It's simple to write tunes using this program. When you want to play them just modify PROCreadnote and PROCplaystring. Of course this has been a fairly limited treatment. Only one of the BBC Micro's four channels has been used, and we haven't varied the loudness at all. Also we've avoided playing more than one note at a time and producing harmonies.

All this, and more, will follow.

```
  10 REM PROGRAM VI
  20 counter=1
  30 PROCnote
  40 PRINT:PRINT
  50 PRINT "Data Statements:"
  60 PRINT:PRINT
  70 PRINT tune$
  80 END
  90 DEF PROCtune
 100 REPEAT
 110 PROCreadnote
 120 PROCplaystring
 130 UNTIL counter>=LEN(tune$)
 140 ENDPROC
 150 DEF PROCnote
 160 CLS
 170 INPUT"Pitch of note?" pitch
```

```
  180 PRINT:PRINT
  190 INPUT"Duration of note?" durati
on
  200 PRINT:PRINT
  210 SOUND 1,-15,pitch,duration
  220 PRINT:PRINT:PRINT
  230 PROCkeepnote
  240 ENDPROC
  250 DEF PROCkeepnote
  260 PRINT"Keep this note?"
  270 PRINT "Press Y and Return for Y
ES"
  280 PRINT "Any letter and Return fo
r NO"
  290 PRINT:PRINT
  300 INPUT key$
  310 IF key$="Y" OR key$="y"
                THEN PROCstring
                   ELSE PROCnote
  320 ENDPROC
  330 DEF PROCstring
  340 tune$=tune$+STR$(pitch)+","
         +STR$(duration)+","
  350 CLS
  360 PRINT "Play the tune now?"
  370 PRINT:PRINT
  380 PRINT "Press Y and Return for Y
ES"
  390 PRINT "Any other letter and Ret
urn for NO"
  400 INPUT key$
  410 IF key$ ="Y" OR key$ = "y"
         THEN PROCtune ELSE PROCnote
  420 ENDPROC
  430 DEF PROCplaystring
  440 pitch=VAL(pitch$)
  450 duration=VAL(duration$)
  460 SOUND 1,-15,pitch,duration
  470 SOUND 1,0,pitch,0
  480 ENDPROC
  490 DEF PROCreadnote
  500 pitch$=""
  510 REPEAT
  520 IF MID$(tune$,counter,1)<>","
         THEN pitch$=pitch$
```

```
            +MID$(tune$,counter,1)
  530 counter=counter+1
  540 UNTIL MID$(tune$,counter-1,1)="
,"
  550 duration$=""
  560 REPEAT
  570 IF MID$(tune$,counter,1)<>","
          THENduration$=duration$
             +MID$(tune$,counter,1)

  580 counter=counter+1
  590 UNTIL MID$(tune$,counter-1,1)="
,"
  600 ENDPROC
```

Chapter 15: Channels, queues and Basic

SO far our treatment of the SOUND command has been fairly limited. Ignoring the special effects channel, the BBC Micro has three sound channels available but we've mostly avoided using more than one at a time. When we did the result was a mess – remember the random music generator!

To make more interesting noises we have to use more than one channel at a time. To allow us to do this without causing chaos BBC Basic allows the SOUND command to be extended to control the flow of notes through the three channels. For the time being, though, let's just stick to one channel and play three notes one after the other.

```
10 REM PROGRAM I
20 SOUND 1,-15,60,20
30 SOUND 1,-15,68,20
40 SOUND 1,-15,76,20
```

Program I isn't very exciting but there's a lot going on which repays closer attention. The three notes play one after the other in the order of the SOUND statements. The first note plays for a count of 20, then the second and when that finishes the third sounds. All very logical. Let's try Program II and see what happens. You'll notice that it's the same as Program I except that each SOUND command is followed by a PRINT statement.

Now you might think that the first note will be played followed by the PRINT statement putting something on screen, then the second note and so on. Have a go and see. What actually occurred was that the three messages were printed on the screen before the first note had finished playing. Then the second note played, then the third. The micro appeared to have jumped around the program lines instead of following them one by one in line number order as usual.

Appearances are deceptive, for the micro read each line in turn and

```
10 REM PROGRAM II
20 SOUND 1,-15,60,20
30 PRINT "NOTE1"
40 SOUND 1,-15,68,20
50 PRINT "NOTE2"
60 SOUND 1,-15,76,20
70 PRINT "NOTE3"
```

followed its instructions before moving onto the next one and obeying that. What happened was that the program read line 20 and started playing the first note. It then read line 30, printed the message, and had a look at line 40 which told it to produce the next note.

The trouble was that there was already a note playing on that channel and (in the very fast world of the microprocessor) it would continue playing for some time. So as not to hold up the program, the micro put the sound it had created on a special queue for that channel and carried on with the next instruction which is to print the second message. This it did and, because BBC Basic is so fast, it did it before the first note had stopped playing. It then came to line 60 which told it to play yet another note.

Because it hadn't finished the first yet, nor gone on to the second, it popped that note on the queue and went on to line 70. The micro printed the last message and, as there were no more lines, the program stopped running. It did not stop making the noises, however, but carried on working its way down the queue. The important point is that the micro continues to play the notes in the queue even after it has stopped processing a program's Basic commands. In this sense the queues – each channel has its own – are independent of the program.

From this you can see that it's possible for a program to rush through its Basic statements, finish whatever it was meant to do and the micro will carry on playing the sounds in the queues regardless. This can at times be embarrassing. Imagine a game that sounds a fanfare every time you zap an alien. Suppose you actually manage to get a few of them so that the fanfares build up in the queue. Then you get hit, it's your last life and the game is over.

The program might order the micro to make a "losing" type of noise on the same channel. This it will do and, since there are still fanfares to be played, it will pop it on the queue. Instead of the losing noise when you get hit, you get a succession of fanfares until the queue reaches the losing sound. In this sense the queue can be a bit of a nuisance, causing the sounds to be out of step with the program.

While the example above is fairly trivial it does show that silly things can happen. It would be nice if there were a way of emptying the notes from the queue to give another note priority. This would allow an "important" sound to take precedence by wiping all the waiting sounds from the queue and so playing it immediately. In our example this would mean that the losing sound would be played immediately.

In fact, there is a way that you can get a SOUND command to take priority over others in the queue. It involves a different way of using the parameter that selects the channel. This, if you remember, is the first number following the SOUND command. Up until now it has only been a single digit – 1, 2 or 3. By using this channel parameter as a four figure hexadecimal number you can do all sorts of clever things with the simple SOUND command.

It is not as complicated as it sounds, especially as we'll only be concentrating on two of the figures. Previously we've used the SOUND command in the format **SOUND W,X,Y,Z** where W selects the channel, X the loudness, Y the pitch and Z the duration of the note. In its expanded role as a hexadecimal number, the channel selection parameter can be looked on as **&TUVW** where T, U, V and W represent figures.

The T parameter we shall ignore for the time being. Similarly, the U parameter, which controls the synchronisation of notes, will be left to a later date. All we are left with is V, which can have the values 0 or 1, and W which is the channel selection number and is used exactly as before – except that it comes at the end of a four digit number.

Previously, when we have been using SOUND, the micro has only come across W in the first position after SOUND and so has assumed that T, U and V were all 0. What this means is that where we have used **SOUND 1,–15,60,40** we could equally have used **SOUND &0001,–15,60,40.** The result would be the same and the SOUND command would work as we have come to expect.

If we want a SOUND command to be obeyed immediately, overriding any other notes that might be in the queue, we change the V parameter from 0 to 1. When the program gets to this line it will obey that command immediately, stopping any note that's playing and "flushing" the queue.

```
10 REM PROGRAM III
20 SOUND 1,-15,60,20
30 SOUND 1,-15,68,20
40 SOUND&0011,-15,76,20
```

Let's see what happens if we change the last line of Program I to make Program III. It seems as though the micro only plays the last note. What's actually happened is that the micro has read line 20 and started to play a note. It then read the next line and put that note on the queue and got to the last SOUND instruction. This had the V parameter set to 1 so the micro immediately cleared the queue, stopped playing the note that it was playing and played the final note. This happens so fast that you only hear the final sound.

Program IV slows things down a little with a delay procedure. Run it and you will hear all three notes. The second line tells the micro to make a noise and, since there is nothing in the queue, it does this. While this is still playing, the micro whirls through the first of the delay loops and when it gets to the second SOUND command it puts this on the queue. It then goes on to the second delay loop.

In the meantime, the first note has reached the end of its alloted time and stops. The second note comes off the queue and starts playing but never reaches the full length specified by its duration parameter. This is because the micro finishes the second delay loop and reads the next SOUND command. This has a 1 for its V parameter so the computer immediately stops playing its present note and obeys that SOUND command straight away. Hence the

```
 10 REM PROGRAM IV
 20 SOUND 1,-15,60,20
 30 PROCdelay
 40 SOUND 1,-15,68,20
 50 PROCdelay
 60 SOUND&0011,-15,76,20
 70 END
 80 DEF PROCdelay
 90 FOR N=1 TO 1000:NEXT
100 ENDPROC
```

shortened second note.

By making the V parameter a 1 you can ensure that SOUND command gets priority over all other sounds playing on that channel. In effect, it cuts short the note that is playing. We could use this in the games program mentioned earlier to make the "losing" SOUND command get rid of all the fanfares in the queue and play immediately. Also it can be used to ensure that noises are synchronised with whatever a program is doing by flushing the queues as in Program V.

```
 10 REM PROGRAM V
 20 SOUND 1,-15,60,20
 30 PRINT "NOTE1"
 40 SOUND 1,-15,68,20
 50 PRINT "NOTE2"
 60 SOUND 1,-15,76,20
 70 PRINT "NOTE3"
 80 FOR N=1 TO 100
 90 PRINT"SOUNDING"
100 NEXT
110 SOUND &11,0,0,0
```

What has happened is that the micro has read the first SOUND command and played a note and then put the other two notes on the queue for channel 1. Then the loop prints "sounding" on the screen and when it has finished the micro reads the last SOUND command. This has a V parameter of 1, so it is executed immediately, flushing the queue at the same time. However, since all the other parameters are 0 there is just silence and the sound's ending coincides with the ending of the printing loop. You might notice from the last line that there is no need to put in the first two zeroes after the ampersand (&).

So far we've covered how the SOUND commands can be stored in queues and how they can appear to operate independently of Basic. We have also seen that this can lead to problems where sounds can be out of phase with what the Basic program is doing and how to remedy this by flushing the queues. We have not mentioned how many sounds can be held in each queue and what happens when they are full. Try Program VI and see what happens.

From what happened in Program I you might expect that it would sound the first note while obeying all the PRINT statements. That is, NOISE1, NOISE2, . . . NOISE9 would appear on the screen while the first note sounded. When that note had finished, the other notes on the queue would take their turn.

What does take place is that the first note sounds with NOISE1 to NOISE6 appearing on the screen. When the first note is finished, the second starts playing and NOISE7 is printed. The third note coincides with NOISE8, the fourth with NOISE9. The reason for this is that although the Basic can carry on quickly by putting the SOUND commands on the queues these queues only have a limited capacity.

The manual says that each channel can hold four sounds as well as the one currently playing. When the channels are full up, the micro will happily work its way through all the Basic statements until it comes to the next SOUND command. Here it comes to a halt until there's a place available on the queue. This means that the program hangs until the note that's playing reaches the end of its duration parameter and the next note can come off the queue. Then the SOUND command that's caused the delay can be processed and popped onto the queue allowing the program to continue.

This is the explanation for what happened in the last program. The program happily played the first sound and popped the rest on the queue until it could do no more and the program came to a stop. When the first note

```
 10 REM PROGRAM VI
 20 SOUND 1,-15,60,20
 30 PRINT "NOISE1"
 40 SOUND 1,-15,68,20
 50 PRINT "NOISE2"
 60 SOUND 1,-15,76,20
 70 PRINT "NOISE3"
 80 SOUND 1,-15,60,20
 90 PRINT "NOISE4"
100 SOUND 1,-15,68,20
110 PRINT "NOISE5"
120 SOUND 1,-15,76,20
130 PRINT "NOISE6"
140 SOUND 1,-15,60,20
150 PRINT "NOISE7"
160 SOUND 1,-15,68,20
170 PRINT "NOISE8"
180 SOUND 1,-15,76,20
190 PRINT "NOISE9"
```

stopped playing the sounds shuffled along the queue and made room for the SOUND command that was causing the delay. The program then obeyed the next PRINT command, putting NOISE7 on the screen and went on to the next line. Here it came to a halt at the next SOUND statement and had to wait until the note stopped playing and so on.

The trouble is that, by my calculations, the program plays one sound and puts five on the queue before it grinds to a halt at the seventh SOUND command and has to wait to print NOISE7. This seems to be one more than the manual would allow. Still, the point remains the same. When a channel's queue is full the next SOUND command for that channel will cause the program to wait.

Chapter 16: All Together Now

SO far we've covered the production of notes on the BBC Micro using the SOUND command, and learned about the sound channels and how to flush them. In this chapter we'll be covering the problems of getting two or more notes to sound at the same time. This means synchronising the notes on different channels.

Try out Program I. This plays three notes at the same time. This is called a chord. If it sounds nasty, then it's a discord. Peters' First Law states that it is much easier to produce discords than harmonious chords! I doubt if anyone has ears sharp enough to tell, but, because it takes time for the micro to interpret a line of Basic, the second and third notes start a split second after each other. Since they have the same duration, this means they will end in a ragged manner, first note one, then note two, then note three.

```
10 REM PROGRAM I
20 SOUND 1,-15,60,20
30 SOUND 2,-15,76,20
40 SOUND 3,-15,88,20
```

Program II has a couple of delay loops in it to accentuate this effect. This is supposed to lessen the clarity of chords, though how anyone can tell beats me. Still, BBC Basic has a way to overcome this undetectable effect. It's not as pointless as you might think, as it also comes in useful when you're writing tunes for the micro, helping you make sure that the right notes are played at the right time.

You'll remember that the channel parameter can be treated as a four digit hexadecimal number &TUVW. W was the channel number with values of 0, 1, 2 or 3 while V was the flushing control which was set to 1 to clear a channel's sound queue. T we ignored, and will continue to do so until we get to

```
10 REM PROGRAM II
20 SOUND 1,-15,60,20
30 FOR N=1 TO 1000:NEXT N
40 SOUND 2,-15,76,20
50 FOR N=1 TO 1000:NEXT N
60 SOUND 3,-15,88,20
```

U Parameter	Effect	Notes in chord
0	Plays without reference to other channels	1
1	Waits for note on one other channel	2
2	Waits for notes on two other channels	3
3	Waits for notes on all channels	4

Figure I: Synchronisation parameter values

the ENVELOPE command. It's the U parameter that decides how notes are synchronised.

If you think about it then you'll realise that for two or more notes to play at the same time each must be on a separate channel. The U parameter in a SOUND command tells the micro that it is not to play that note until there is a certain number of other channels with notes.

When the required number of channels containing notes is reached then all of them start playing at the same time. The value of U decides how many other channels must have notes before the chord is played. If U is 0, then the micro plays the note when it can, without reference to other channels. If it is 1, the micro will hold that note until there is another note available on another channel. If it is 2 it will wait until two other notes are available, that is all three notes will start simultaneously. If it is 3, it will wait for three other notes when all four of the micro's channels will be in use. The values are shown in Figure I. Try Program III to see it in action.

```
10 REM PROGRAM III
20 SOUND &0201,-15,60,40
30 FOR N=1 TO 1000:NEXT N
40 SOUND &0202,-15,76,40
50 FOR N=1 TO 1000:NEXT N
60 SOUND &0203,-15,88,40
```

What happens is that the micro reads the first SOUND statement and, because it has 2 in the U parameter, it waits for two notes to be available on other channels. When it has these it plays all three notes together. The delay loops make for a slight pause before the chord is played. This shows that the micro really is waiting for the other two notes. Since the duration parameters are the same, the notes all stop playing at the same time. If this weren't the case there would be a ragged ending. Setting the U parameter only

```
10 REM PROGRAM IV
20 SOUND &0301,-15,60,40
30 SOUND &0302,-15,76,40
40 SOUND &0303,-15,88,40
```

synchronises the start of the notes.

Can you figure out why nothing seems to happen when you run Program IV? Don't think your computer has gone wrong. What happens is that the micro reads the U parameter of the first SOUND statement. As this is 3 it will not play that note until sounds are available from three other channels. Unfortunately the program only gives two other notes so the micro can't play. Hence the silence.

However, the notes in the channel don't just disappear. They are still in the queues waiting for a note on the fourth channel. Give them one by entering **SOUND &0300,–15,88,40** and pressing Return. Now that the fourth channel has a note, all four will sound.

Before you run the other programs in this chapter it would be wise to press Escape to clear the sound queues of any left-over notes. If you don't, you might get some strange results. You have been warned. One point to bear in mind is that when the micro comes across a U parameter that makes it search the channels for other notes, it will quite happily accept ones that are already playing. Try Program V.

```
10 REM PROGRAM V
20 SOUND &0101,-15,60,40
30 SOUND &0102,-15,76,40
40 SOUND &0203,-15,88,40
50 SOUND &0200,-15.88.40
```

At first glance you might think it will play the first two notes as the U parameter ties them together. Then it will hang up as the next two notes have a U parameter of 2 and need a third before they can sound. Playing the program will soon show you that this isn't so, as all four notes are played at once. This is because the micro comes to the last two SOUND statements and starts searching the other channels for a note. Since they're already playing notes and carry a U parameter, it will quite happily accept either of them to make up the third note. And so the U parameter's condition is fulfilled and the note plays.

However, if we put in a line of Basic as in Program VI you'll notice that only the first two notes are played and that the third and fourth are left in the queues. Enter one of the sound commands from the first two notes and you'll hear these left-over sounds along with the one you've entered. Remember, if you're getting funny effects it's probably because you've got some notes left in the queues. Get rid of them with Escape.

Finally, Program VII combines the synchronisation parameter U with the flushing parameter V, which we covered in the last chapter. This allows us to play three chords in succession by pressing the key. Notice that the final SOUND statements which flush the chords are synchronised to prevent a ragged ending, though I don't know who would be able to tell if they weren't.

```
   10 REM PROGRAM VI
   20 SOUND &0101,-15,60,40
   30 SOUND &0102,-15,76,40
   40 FOR N=1 TO 4000:NEXT N
   50 SOUND &0203,-15,88,40
   60 SOUND &0200,-15,88,40
```

```
   10 REM PROGRAM VII
   20 SOUND &0201,-15,52,254
   30 SOUND &0202,-15,68,254
   40 SOUND &0203,-15,80,254
   50 PRINT"Press key for next chord.
"
   60 pause$=GET$
   70 SOUND &0211,-15,60,254
   80 SOUND &0212,-15,76,254
   90 SOUND &0213,-15,88,254
  100 PRINT"Press key for next chord.
"
  110 pause$=GET$
  120 SOUND &0211,-15,68,254
  130 SOUND &0212,-15,84,254
  140 SOUND &0213,-15,96,254
  150 PRINT"Press key to end."
  160 pause$=GET$
  170 SOUND &0211,0,0,0
  180 SOUND &0212,0,0,0
  190 SOUND &0213,0,0,0
```

Chapter 17: Not the Sound of Music

THE sound effects channel is the channel you select by making the first parameter of the SOUND command equal to zero. This means that soon you'll be using lots of SOUND commands such as **SOUND 0,–15,2,40.** With channel 0 you can create all sorts of weird and wonderful effects – especially when you start to use the ENVELOPE command which we'll come to in the next chapter.

In all there are eight basic sound effects available on channel 0. You can get them by making the pitch parameter of a channel 0 command equal to a number between 0 and 7. Type in Program I, run it and you'll get a conducted tour through the various noises. Figure I summarises the result each value of the pitch parameter has on a channel 0 SOUND command.

So to use the sound effects channel we use a SOUND command such as **SOUND 0,–15,pitch,20** where pitch has a value between 0 and 7. The value you give to pitch decides what kind of sound effect you get. As you'll see from Figure I, putting in values of 0, 1 or 2 all give something called "periodic

```
   10 REM PROGRAM I
   20 REPEAT
   30 FOR pitch=0 TO 7
   40 CLS
   50 PROCnoise(pitch)
   60 NEXT pitch
   70 UNTIL FALSE
   80 END
   90 DEF PROCnoise(pitch)
  100 PRINT TAB(5,5) "This is noise n
umber ";pitch;"."
  110 IF pitch=3 OR pitch=7 THEN PRIN
T TAB(5,15)"This can be varied by usi
ng" TAB(5,17)" a sound on channel 1"
  120 PRINT TAB(5,20) "Press a key fo
r next noise."
  130 SOUND 0,-15,pitch,255
  140 Wait=GET
  150 SOUND &10,0,0,0
  160 ENDPROC
```

Value of P	Noise produced on channel 0
0	High frequency periodic
1	Medium frequency periodic
2	Low frequency periodic
3	Periodic – frequency depends on pitch of channel 1
4	High frequency white
5	Medium frequency white
6	Low frequency white
7	White – frequency depends on pitch of channel 1.

Figure I: Pitch values on channel 0

noise". If pitch is equal to 0 you get high frequency periodic noise. Enter **SOUND 0,–15,0,40** and you'll see (or, rather, hear) what I mean.

As you might guess, values of pitch of 1 and 2 give medium frequency and low frequency periodic noise respectively. Try **SOUND 0,–15,1,40** and **SOUND 0,–15,2,40** if you don't believe me. To hear all three in order run Program II.

```
10 REM PROGRAM II
20 FOR pitch=0 TO 2
30 SOUND 0,-15,pitch,20
40 NEXT pitch
```

This should help you hear the difference between the three. If you want you can use a selection of periodic noises to produce something approaching a "Close Encounters" noise. Program III does this by playing each of the periodic noises in turn over and over, the duration of each note getting less each time round the REPEAT . . . UNTIL loop.

```
10 REM PROGRAM III
20 time=20
30 REPEAT
40 FOR pitch=0 TO 2
50 SOUND 0,-15,pitch,time
60 NEXT pitch
70 time=time-2
80 UNTIL time=0
```

Notice that it is time and not TIME in the above program. TIME refers to the internal clock of the micro and could cause some funny results if used! For the moment let's ignore what happens when you make the pitch equal to 3 and go straight on to what happens when you make it 4, 5, or 6. If pitch has these values, channel 0 produces "white" noise. If you want to know what that is then enter **SOUND 0,–15,4,100.** This will give you five seconds of high frequency white noise. It won't come as a suprise to learn that **SOUND 0,–15,5,100** will give you five seconds of medium frequency white noise and **SOUND 0,–15,6,100** five seconds of low frequency white noise.

Try Program IV which plays all three one after the other for a gradually decreasing time. It's very like Program III, so don't type it all in again. Just use the Copy key and change the lines you have to:

```
10 REM PROGRAM IV
20 time=20
30 REPEAT
40 FOR pitch=4 TO 6
50 SOUND 0,-15,pitch,time
60 NEXT pitch
70 time=time-2
80 UNTIL time=0
```

Now what does all that white noise remind you of? A steam engine? Try Program V. This uses the white noise pitch parameters in a REPEAT . . . UNTIL loop to make the sounds of an accelerating engine. The REPEAT . . . UNTIL FALSE loop takes over when the engine is at full speed.

```
 10 REM PROGRAM V
 15 REM STEAM ENGINE
 20 time=20
 30 REPEAT
 40 FOR pitch=5 TO 6
 50 SOUND 0,-15,pitch,time
 60 NEXT pitch
 70 time=time-2
 80 UNTIL time=10
 90 REPEAT
100 SOUND 0,-15,5,time
110 SOUND 0,-15,6,time
120 UNTIL FALSE
```

Still, enough of this nostalgia. What about the two values we've ignored? What do they do? It's quite simple really, if you think about it. We've been using the bit of the SOUND command that we normally use for selecting the pitch (higher or lower) to pick the kind of noise we want, white or periodic.

So how do we change the pitch of the noise we are making? Can we make the white noise and the periodic noise go up and down in pitch like normal notes on the other sound channels? The answer is that we can, using values of 3 and 7 in the pitch parameter of a channel 0 SOUND command.

If you put these values in the pitch parameter then the pitch of the noise that is played depends on the pitch of a note that is playing on sound channel 1. In other words, by using 3 or 7 you can use a SOUND command on channel 1 to alter the pitch, higher or lower, of periodic and white noise, respectively. Program VI shows this being done with the pitch parameter in line 30 set at three. Run it and see what happens.

```
10 REM PROGRAM VI
20 FOR pitch=100 TO 200
30 SOUND 0,-15,3,20
40 SOUND 1,0,pitch,20
50 NEXT pitch
```

As you'll hear, the pitch of the noise goes upwards. This is because the FOR . . . NEXT loop is raising the note produced by the channel 1 SOUND command in line 40. The pitch of the note produced by line 30's channel 0 SOUND command varies with the pitch of the note from channel 1.

If you look carefully at line 40 you'll see that the amplitude has a value of 0 which means that you can't hear what is being played on that channel. It still varies the noise on channel 0, however. In fact it's a good thing that you can't hear the channel 1 note. Run Program VII and you'll see what I mean.

A mess, isn't it? Now try Program VIII which, like Program VI, plays a rising sequence of periodic noises.

Both of the duration parameters of the SOUND commands are the same. What would happen if they were different? Try changing the duration

```
10 REM PROGRAM VII
20 FOR pitch=100 TO 200
30 SOUND 0,-15,3,20
40 SOUND 1,-15,pitch,20
50 NEXT pitch
```

```
10 REM PROGRAM VIII
20 FOR pitch=50 TO 100
30 SOUND 0,-15,3,40
40 SOUND 1,0,pitch,40
50 NEXT pitch
```

parameter in line 40 to 20 or 10 or 60 and see what happens. Can you explain the odd effects? I'll give you a clue if you just stand in the queue!

If you make the pitch parameter 7 instead of 3 in the last three programs you'll see that channel 1 has the same effect. Only this time it's a rising stream of white noises that are produced.

Finally, let's have some sound effects. Run Programs IX and X and you'll see what you can do with 3 and 7 as the pitch parameters. The first uses periodic noises to produce a motorbike or moped sound:

```
 10 REM PROGRAM IX
 20 REM MOTOR CYCLE
 30 FOR pitch= 100 TO 120
 40 SOUND 0,-15,3,5
 50 SOUND 1,0,pitch,5
 60 NEXT pitch
 70 SOUND 1,0,pitch,80
 80 SOUND 0,-15,3,80
 90 FOR pitch= 120 TO 100 STEP -1
100 SOUND 0,-15,3,5
110 SOUND 1,0,pitch,5
120 NEXT pitch
130 SOUND 1,0,pitch,20
140 SOUND 0,-15,3,20
```

The second uses white noise to imitate an unsuccessful rocket!

```
 10 REM PROGRAM X
 20 REM ROCKET
 30 FOR pitch= 100 TO 200 STEP 4
 40 SOUND 0,-15,7,5
 50 SOUND 1,0,pitch,5
 60 NEXT pitch
 70 SOUND 1,0,pitch,80
 80 SOUND 0,-15,7,80
 90 FOR pitch= 200 TO 100 STEP -4
100 SOUND 0,-15,7,5
110 SOUND 1,0,pitch,5
120 NEXT pitch
130 SOUND 0,-15,6,60
```

Chapter 18: Addressing the Envelope

SO far we've thoroughly explored the SOUND command, its intricacies and vagaries. We have managed to produce some interesting sounds on the way and, I hope, had a lot of fun in the process. However we haven't explored all of the BBC Micro's sound capabilities. We've yet to use the ENVELOPE command.

Now this is a formidable looking beast, being followed as it is by 14 numbers or parameters. The numbers and what they do are shown in Table I, but don't let them put you off. As long as you keep your nerve and take things step by step you won't come to any harm.

Why have an ENVELOPE in the first place, you might ask. After all, we've been doing quite well with the nice, comparatively simple SOUND command. Why complicate matters? The answer is that although we can do a lot with SOUND, the ENVELOPE command allows us to do a whole lot more!

The note that we get from the BBC Micro with the SOUND command

Parameter	Range	Meaning
N	1 to 4	Envelope label
T	0 to 127	Length of each step in 100th of seconds.
	(+ 128)	Added to stop auto-repeat
PI1	−128 to 127	How the pitch will change with each step in the first part
PI2	−128 to 127	Pitch change per step in the second part
PI3	−128 to 127	Pitch change per step in the third part
PN1	0 to 255	The duration of section one, measured in steps of T
PN2	0 to 255	The duration of the second section in steps of T centiseconds length
PN3	0 to 255	The duration of the third section PN3 steps, each of T centiseconds
AA	126	These values affect the loudness of the note and are fixed for this article
AD	0	
AS	0	
AR	−126	
ALA	126	
ALD	126	

Table I: Values and meaning of the ENVELOPE parameters

tends to be rather "electric" and not very exciting. Try **SOUND 1,–15,100,200** Well, it's a sound, but not a very interesting one. Now let's vary this sound using an ENVELOPE command to define an envelope. Type in **ENVELOPE 1,1,70,16,2,2,0,0,126,0,0,–126,126,126** then **SOUND 1,1,100,200** and notice the difference.

The SOUND command is exactly the same as the previous one except that the loudness parameter of –15 has become a 1. This figure 1 just tells the micro to look for an envelope which has been typed in as envelope 1. I must point out that the envelope must have been defined before the SOUND command can use it. The micro then obeys the SOUND command, but the note that it plays is influenced by the ENVELOPE command we defined previously. In this case, the rather boring noise we produced earlier has now become the sound of an alarm clock. I don't want anyone dozing off while they're reading this chapter!

One thing to notice is that it is the SOUND command that makes the noise. You can type in ENVELOPEs until you're blue in the face but they won't make a noise. All they do is alter the noises made by any SOUND command that refers to them. The envelope defined by an ENVELOPE command varies the sound produced by a SOUND command (provided the SOUND command refers to the envelope by the number in its loudness parameter).

Now let's have a look at the parameters that follow the ENVELOPE command: **ENVELOPE N,T,PI1,PI2,PI3,PN1,PN2,PN3,AA,AD,AS,AR, ALA,ALD** The beast still looks pretty formidable, but we'll go through it step by step. I have used the same parameter names as you'll find in the User Guide and some of the books, so you can cross reference.

You'll be pleased to know that we're ignoring the last six parameters, the ones that begin with A. These affect the loudness, or amplitude, of a note and we'll cover them in the next chapter. So we're left with the first eight parameters. These label the envelope, decide how long its effects are going to last and vary the pitch of the note.

Now why would you want to vary the pitch of a note? It seems odd that you should specify the pitch in a SOUND command and then use an envelope to vary it. The reason is that in real life notes are never the pure, steady sound that we get from the BBC Micro's sound channels. What we call the pitch of a note is just an average. The actual note "wobbles" around that value. It's this wobbling that lets us tell the difference between the middle C on a piano and the middle C on a violin. The average pitch is the same, but the wobbles vary. (I beg the forgiveness of any musical genius who may read the above.)

The ENVELOPE command allows us to approximate these wobbles and so make the noises our micro produces sound like a saxophone or an alarm clock. I should point out that the loudness also wobbles, but we'll leave that until later. Now let's take a look at the first parameter following the ENVELOPE command. As you'll see from Table I, this is called N and can have values of 1 to 4. N is a reference number. You decide what number you want to refer to the envelope by and put that number in N. Then when you want a sound to be influenced by that envelope you just put that number in the loudness parameter of that SOUND command.

If you look at the alarm clock again you'll see that we made the N in the envelope equal to 1 and put a 1 in the loudness parameter of the following sound statement. We could have used 2 as the label, or 3, or 4. From this you'll see that you can have up to four envelopes ready for use.

The next parameter we come to is T which, as you might guess, stands for time. As you'll see in a moment, the ENVELOPE command works in a basic unit called a step. It will affect a SOUND command in one way for a specified number of steps, then in another way for another number of steps and so on. What T does is to allow you to choose how long each of these steps will be. It can have values from 0 to 127, and is measured in hundredths of a second. If we have T equal to 100 then each step is a hundred centi-seconds long, which, if my maths is correct, makes each step last a second.

Looking at the alarm clock envelope again, you'll see that T is equal to 1, so each step lasts for just one hundredth of a second. So N just labels the envelope while T decides how long each of its steps will last. Now let's explore the next six parameters which cause the wobbles in the pitch of the note, allowing it to make all manner of amazing sounds. There are three stages or sections to the wobble, or change, in pitch. The pitch of the note can go up, go down, or stay the same in each section.

Figure I shows the effect of a hypothetical envelope on a hypothetical sound. The horizontal line is the pitch of the note as defined in the SOUND command. The wavy line shows the notes actually played under the influence of the ENVELOPE command. As you can see, there are three stages: the

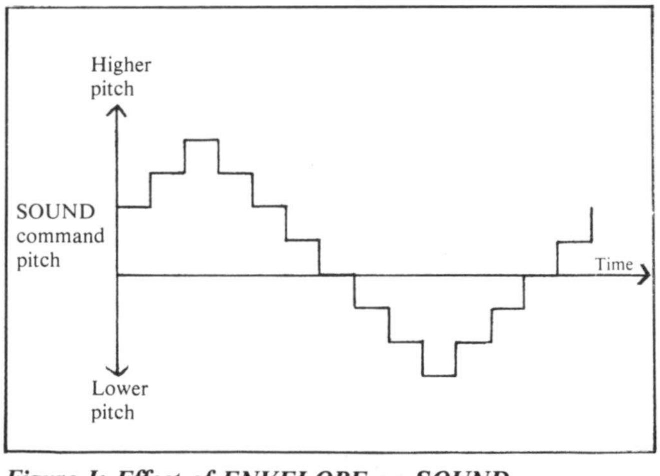


Figure I: Effect of ENVELOPE on SOUND

pitch rises in the first stage, goes down in the second, and rises again in the third. Of course it could have risen in all three stages, or gone down in all three, but let's just stick to it going up, down, and up again.

These ups and downs are caused by the parameters PI1, PI2, and PI3 in the ENVELOPE command. PI1 is the increase or decrease of pitch for each step in stage one. Similarly PI2 sets the increase or decrease per step for stage two and PI3 that for stage three. The units of pitch are the same as we used in the SOUND command. Eight of them make up a range of one tone, four of them a semitone. The length of each stage is determined by PN1, PN2, and PN3. The value of PN1 determines that the first stage will consist of PN1 steps, each of length T. Similarly the second stage is of length PN2 steps, again of length T.

As you might guess, the third stage is of length PN3 steps, each step lasting for T steps. Figure II shows all this. It's just another version of Figure I with the parameters put in. Let's have a look at this in practice. Type in and run Program I.

Exciting, isn't it? Can you understand the ENVELOPE command of line 20 and how it affects the noise made by the SOUND command of line 30? Looking at line 20 we first of all have the ENVELOPE keyword, then comes the number 1. This is the label that the envelope will be known by. Then comes 100, which means that each step of the envelope will take 100 centi-seconds or one second. As we can see, PI1 has the value of 4. This means that for each

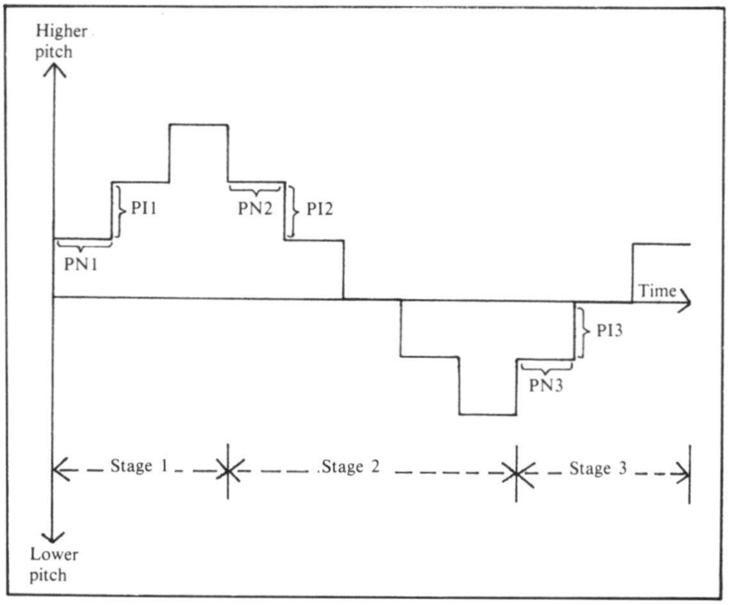


Figure II: ENVELOPE parameters

```
   10 REM PROGRAM I
   20 ENVELOPE 1,100,4,0,0,10,0,0,126
,0,0,-126,126,126
   30 SOUND 1,1,50,200
```

step of one second the note produced by the SOUND command will rise in pitch by one semitone.

Looking at the value for PN1 we see that there will be 10 steps. Each will last for one second and every second the note will rise by a semitone. Run it again and see that this is the case. You'll notice that we've made the parameters of the other two stages equal to zero to stop things getting too complicated. The last six parameters effect the amplitude of the note. We'll deal with this in the next chapter, and for the moment just stick to the six values in line 20.

You might also notice that the envelope lasts for 10 seconds (PN1 times T) and the SOUND command lasts for 10 seconds. What happens if they don't coincide so neatly? Try Program II and see.

```
   10 REM PROGRAM II
   20 ENVELOPE 1,100,4,0,0,11,0,0,126
,0,0,-126,126,126
   30 SOUND 1,1,50,200
```

In this program you'll see that PN1 has become 11 so we should expect the envelope to last for 11 seconds. However as the note is only being played by the SOUND statement for 10 seconds, the last step never gets taken. The reverse is the case in Program III.

```
   10 REM PROGRAM III
   20 ENVELOPE 1,100,4,0,0,5,0,0,126,
0,0,-126,126,126
   30 SOUND 1,1,50,200
```

As you can see, PN1 has now become 5. The SOUND statement is the same, so the note will play for 10 seconds, but the envelope will only last for five seconds (five steps, each of one second). What happens during the last five seconds? As you can hear, the envelope auto-repeats. That is, when it comes to the end of its steps, it pauses for a couple of steps (one for each of the other two stages), then starts again and continues until the SOUND statement runs out of puff after 10 seconds. This auto-repeat can be very useful for producing sound effects, but it can also be a nuisance. Happily, it can be switched off by adding 128 to whatever value of T you've put in the ENVELOPE command.

In Program IV to switch off the auto-repeat I've added 128 to the value of

```
   10 REM PROGRAM IV
   20 ENVELOPE 1,228,4,0,0,6,0,0,126,
0,0,-126,126,126
   30 SOUND 1,1,50,200
```

T, previously 100. This makes for a T value of 228. Each step is still one second long, the extra value justs ensures no auto-repeat. Have a go at Program IV and see what I mean.

The auto-repeat has been switched off so the sound increases for six steps then stays at that pitch for the remaining four seconds of the note. Right, let's see about doing something with the second stage of the pitch envelope. Let's give PI2 (pitch increment two) a value of –4 and PN2 a value of 5. Run Program V and see what happens.

```
   10 REM PROGRAM V
   20 ENVELOPE 1,228,4,-4,0,5,5,0,126
,0,0,-126,126,126
   30 SOUND 1,1,50,200
```

As you might have guessed, the whole thing lasts for 10 seconds (PN1+PN2 steps, each of one second duration). The pitch of the note goes up one semitone for each of five steps, then it goes down one semitone for five steps. Program VI gives values to PI3 and PN3 and so we make use of the third stage of the pitch part of the envelope.

```
   10 REM PROGRAM VI
   20 ENVELOPE 1,228,4,-4,4,3,3,3,126
,0,0,-126,126,126
   30 SOUND 1,1,50,200
```

As you may have realised, the envelope lasts for nine seconds while the sound lasts 10. This creates an uneven effect, the last note lasting two seconds. Program VII shortens the note played by the SOUND statement to compensate for this.

```
   10 REM PROGRAM VII
   20 ENVELOPE 1,228,4,-4,4,3,3,3,126
,0,0,-126,126,126
   30 SOUND 1,1,50,180
```

You can figure out how long the envelope lasts by adding up PN1, PN2, and PN3 and multiplying the result by T. That is you add up the total number of steps and multiply them by the time each step takes. And that's really all there is to the pitch part of the ENVELOPE command. It's just a matter of

```
   10 REM PROGRAM VIII
   20 ENVELOPE 1,255,4,-4,0,3,3,0,126
,0,0,-126,126,126
   30 SOUND 1,1,50,255
```

deciding how many steps you want in each stage and what happens to the note for each step. It's quite simple really. Practice using envelopes and they'll soon stop seeming so difficult. One thing that you should be wary of though, is shown by Program VIII.

You might think that as you start by going up three steps and then go down three steps you'll start and end on the same note. However this isn't the case – the sound ends up one semitone below the beginning pitch. Figure III shows what's happened. The ENVELOPE takes effect straight away and the first note played is a semitone above the pitch of the note in the SOUND statement. It then goes up two more semitones and drops three. The final note is, as you can hear, a semitone below the first. Program IX rectifies this by having one step less in stage two than in stage one. The final note is now the same pitch as the first.

```
   10 REM PROGRAM IX
   20 ENVELOPE 1,255,4,-4,0,3,2,0,126
,0,0,-126,126,126
   30 SOUND 1,1,50,255
```

And that's the end of our first excursion into the ENVELOPE command. It's not all that hard when you get used to it, and you can do amazing things

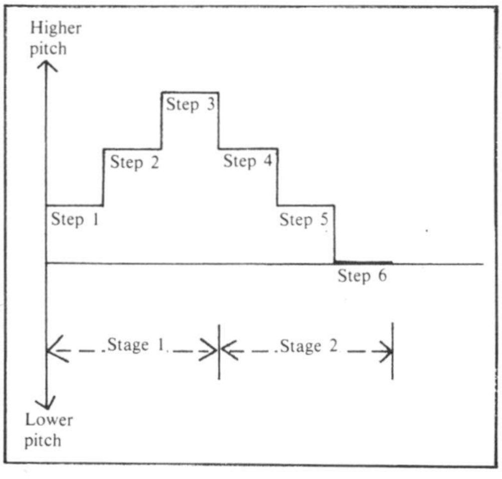


Figure III: The ups and downs of ENVELOPE

with sound envelopes. Try Program X and listen to the four different sounds produced. If you look at the listing you'll see that the SOUND command is the same in each case, only the envelope differs.

```
   10 REM PROGRAM X
   20 count=0
   30 ENVELOPE 1,255,4,-4,4,3,3,3,126
,0,0,-126,126,126
   40 ENVELOPE 2,127,4,-4,4,3,3,3,126
,0,0,-126,126,126
   50 ENVELOPE 3,20,4,-4,4,3,3,3,126,
0,0,-126,126,126
   60 ENVELOPE 4,1,4,-4,4,3,3,3,126,0
,0,-126,126,126
   70 REPEAT
   80 envelope = 1 + count MOD4
   90 PRINT:PRINT
  100 PRINT "Envelope number ";envelo
pe
  110 PRINT:PRINT
  120 PRINT "Press key for next envel
ope."
  130 SOUND 1,envelope,50,255
  140 count=count+1
  150 WAIT$=GET$
  160 SOUND &11,0,0,0
  170 UNTIL FALSE
```

Chapter 19: Envelopes licked

IN the last chapter we explored the ENVELOPE command and saw how we could use it to vary the pitch of a note. We studiously ignored the last six of the 14 parameters that follow ENVELOPE. Now we'll be dealing with these last six and seeing how the values we give them can affect the amplitude, or loudness, of a note.

The full set of ENVELOPE parameters is:

```
ENVELOPE N,T,PI1,PI2,PI3,PN1,PN2,PN3,
AA,AD,AS,AR,ALA,ALD
```

Let's go straight on to Program I and see what happens if we put values in two of these amplitude parameters. The note played by the SOUND

```
   10 REM PROGRAM I
   20 ENVELOPE 1,100,0,0,0,0,0,0,30,0
,0,0,120,0
   30 SOUND 1,1,50,80
   40 SOUND 1,0,0,0
```

command in line 30 gets louder (in four steps) and then stops. This is a result of the parameters we put in the last part of the envelope definition. Table I gives a list of these parameters and their effects and ranges. I have stuck to the standard parameter name abbreviations for the sake of uniformity.

You might ask why have these amplitude parameters anyway – surely they just make things more complicated? The answer is that they do, but they also allow the BBC Micro's sound generator to mimic musical instruments. When a note is played on a violin or piano it doesn't have the same loudness all the time. It builds up from silence to a maximum and then gets quieter again. Each instrument has a different amplitude envelope, as this characteristic increase and decrease in loudness is called. Some achieve their maximum loudness rapidly, then fade away quickly. Others take relatively longer to reach their full power and then die away gently.

Until now all we have used to control the loudness of a sound was the amplitude parameter of the SOUND command. As you'll remember, this is the second one after the SOUND. The note started at the loudness specified by that parameter and stayed there until it finished. It was simple, but it wasn't

Parameter	Range	Meaning
AA	−127 to 127	Amplitude change per step in the attack phase.
AD	−127 to 127	Amplitude change per step in the decay phase.
AS	0 to −127	Amplitude decrease during each step of sustain phase.
AR	0 to −127	Amplitude decrease during each step of release phase.
ALA	0 to 126	Target (maximum) value achieved during attack phase.
ALD	0 to 126	Target (minimum) value achieved during decay phase.

Table I: Amplitude parameters of ENVELOPE

like real life. The amplitude part of the ENVELOPE command was designed to overcome this. It divides the time the note plays into four sections, each with a different characteristic.

Figure I shows this schematically. The first section is the attack phase. In

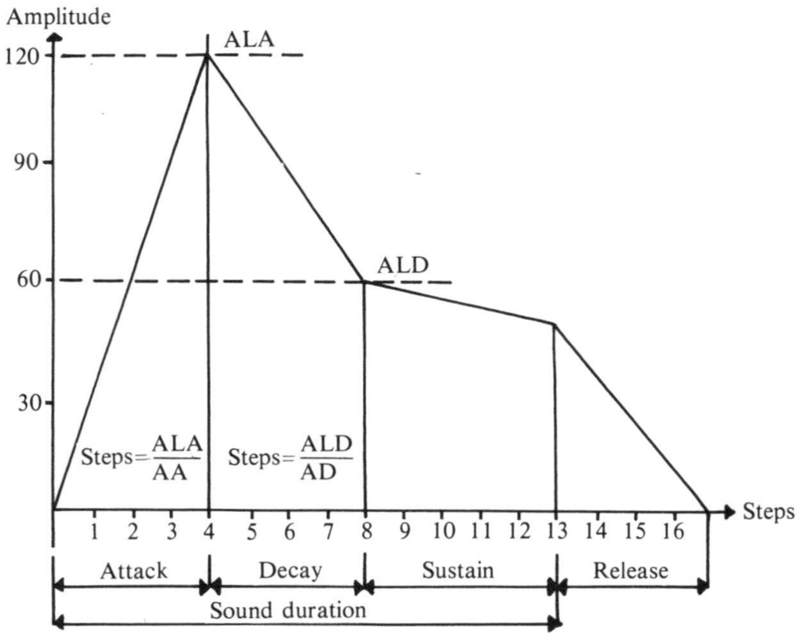


Figure I: Amplitude parameters in action

this the note builds up from nothing to its maximum loudness. The parameters ALA and AA control this phase. After the note has reached its peak, it enters the decay phase, where it goes into a gradual decline. The parameters ALD and AD govern this. The final two sections are the sustain phase and the release phase, controlled by AS and AR respectively.

We'll ignore these last two for the time being and go back to Program I to see what caused the changes in loudness we heard. Notice that it is only the loudness which is changing. The pitch of the note stays exactly the same. If you look at the envelope defined by line 20, you'll see it has the label 1 and each step lasts for one second. The six pitch parameters are all 0, so they don't have any effect on the note.

The only parameters that have a value assigned to them are ALA and AA, the ones that govern the attack phase. The maximum loudness the note can reach is fixed by ALA. This can have values between 0 and 126, corresponding to the range of 0 to −15 allowed by the SOUND command. As you can see, the amplitude envelope allows for much finer volume control.

In Program I, I've set the level at 120, mainly because it's loud and divides easily. The rate at which the loudness of the note increases to get to this maximum level is fixed by the parameter AA. This can have values from −127 to 127, is normally positive and is the change of amplitude per step. It corresponds to the gradient of the attack phase shown in Figure I. For each time step forward the volume of the note increases by AA.

In Program I the value is 30. For each time step forward the volume of the note will increase by 30 until it reaches the level of 120. The number of steps it takes to do this is 120 divided by 30, which give the answer four. We can actually hear the four steps as the program runs. Since each step lasts for one second the whole thing takes four seconds, which is the value I've made the note sound for in line 30. Don't worry about the SOUND command in line 40 it's just a dummy note, there to catch garbage. You'll learn its significance later.

Run Program II and you'll hear it has eight steps. The sound gets louder,

```
   10 REM PROGRAM II
   20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,0,0,120,0
   30 SOUND 1,1,50,160
   40 SOUND 1,0,0,0
```

then fades away because there are now values for the decay phase of the amplitude envelope. The target value ALD is now 0 (no volume) and the drop in amplitude per time step, AD, is now −30.

Figure II shows what has happened. Notice that the peak amplitude lasts for two steps. This "doubling" effect occurs where two phases join and can

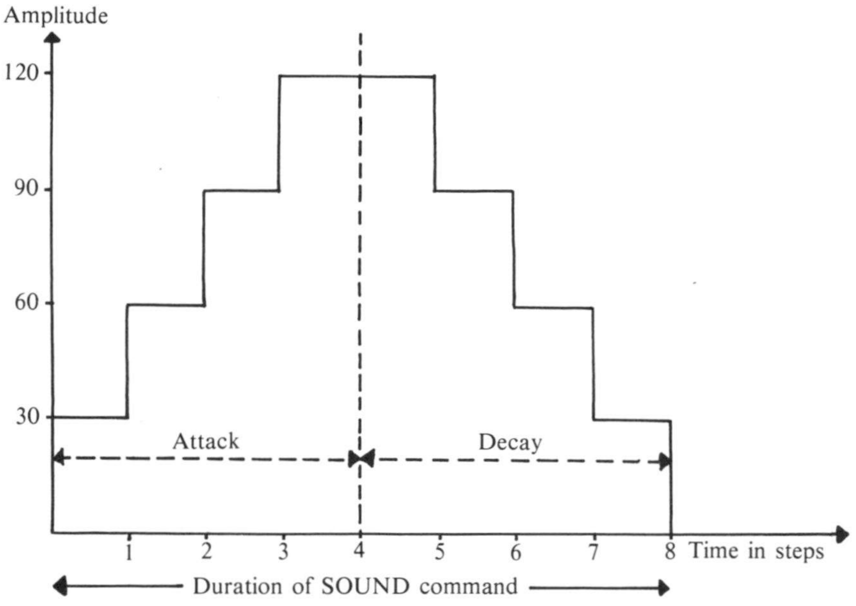


Figure II: Program II's amplitude envelope

lead to some unexpected results if you don't keep it in mind. Still that's for later when you experiment with envelopes for yourself. Remember though, if you think that something has lasted a step more than it ought it's probably because of the "join" of two phases.

You'll notice that I have increased the duration of the SOUND command in line 30 to eight seconds, exactly the length of the envelope. You might wonder what would happen if the duration of the SOUND command was shorter or longer than that dictated by the parameters of the ENVELOPE command governing the sound.

Program III shows what happens if it's shorter. Here the duration of the

```
   10 REM PROGRAM III
   20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,0,0,120,0
   30 SOUND 1,1,50,80
   40 SOUND 1,0,0,0
```

sound is only four seconds, while looking at the parameters of the envelope would lead us to expect it to last eight seconds as before. What happens is that the sound lasts for four seconds, so only the first four seconds of the envelope get a chance to work. The rest, in this case the decay phase, is ignored.

Program IV shows us what happens if the duration of the sound exceeds that of the parameter. As you might expect, the note carries on at the final

```
    10 REM PROGRAM IV
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,0,0,120,0
    30 SOUND 1,1,50,200
    40 SOUND 1,0,0,0
```

volume it reached, finishing off the duration parameter of the SOUND command. The trouble is that since the final loudness of the sound is 0 we can't hear it! Still, it is there, playing away silently until the ten seconds are up. If you don't believe me, make the SOUND command in line 40 produce a real sound on the same channel. You won't hear it until the ten seconds are up.

Now the value of ALD doesn't always have to be 0. We can have any value between 0 and 126. In Program V, ALA has the value of 60. This is the target volume for the decay phase and it is reached in steps of –30 (AD). With Program V the duration of the SOUND is such that it finishes at the same

```
    10 REM PROGRAM V
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,0,0,120,60
    30 SOUND 1,1,50,120
    40 SOUND 1,0,0,0
```

time as the amplitude reaches 60, after six seconds. The trouble is that this means you can't hear the note at its final volume as you cut off the sound just as it reaches it.

Program VI is exactly the same, except that it lasts for nine seconds. Now

```
    10 REM PROGRAM VI
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,0,0,120,60
    30 SOUND 1,1,50,180
    40 SOUND 1,0,0,0
```

you can hear the final decrease in volume. Notice again that the sound continues at the final loudness reached until all of the duration parameter of the SOUND command is used up.

But what of the sustain phase, governed by the parameter AS? What does this do? It's quite simple really. All that it does is use up the rest of the duration of the SOUND command. The attack and decay phase both use up part and the sustain phase lasts for whatever period, if any, is left.

Now that we know how long the sustain phase lasts, let's see what it does by running Program VII, which gives AS the value of −15. As you can hear, the volume increases to 120 and decreases down to 60 as you might expect

```
    10 REM PROGRAM VII
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,-15,0,120,60
    30 SOUND 1,1,50,200
    40 SOUND 1,0,0,0
```

from the parameters. This uses up six seconds of the 10 that the note will play for. During the remaining four seconds of the note the amplitude envelope enters the sustain phase. Here the AS parameter is −15. This, as you might guess, means that the volume decreases by a factor of 15 for each time step. In this program the volume in the sustain phase goes down from 60 by 15 each second. It gets to 0 volume just as the time runs out.

Program VIII is the same as Program VII except that the duration of the

```
    10 REM PROGRAM VIII
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,-15,0,120,60
    30 SOUND 1,1,50,240
    40 SOUND 1,0,0,0
```

note caused by the SOUND statement is now 12 seconds. The volume still reaches 0 after ten seconds but the note keeps playing, silently, for the last two seconds. Again, if you don't believe me put a real note on channel 1 in line 40 and it won't sound until the note in line 30 has had its full 12 seconds. Of course you don't have to arrange it so that AS eventually reduces to 0 volume. In Program IX it has the value −5 so the volume doesn't have a chance to reach 0 before the time runs out.

```
    10 REM PROGRAM IX
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,-5,0,120,60
    30 SOUND 1,1,50,240
    40 SOUND 1,0,0,0
```

Looking at Program X two things are apparent. Line 40, the garbage collecting line, has gone and the value of AS is now −10. A quick calculation will show that the attack and decay phases will take six seconds. A decline of −10 for each of the six remaining one second steps of the sustain phase should take the volume to 0 just as the note finishes playing. Try it and see! The note

```
    10 REM PROGRAM X
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,-10,0,120,60
    30 SOUND 1,1,50,240
```

carries on beyond the 12 seconds you'd expect from the SOUND command.

What's happened is that the sound generator has come to the end of the sustain phase and entered the release phase. This is a rather weird construction which is independent of the duration parameter of the SOUND command. The envelope reaches the end of the sustain phase when the duration of the note runs out. The sound generator then searches around for something to do next. If a note is waiting on the same channel it will play it. If not it will carry on playing the last note until the next note comes along.

Never mind that the duration set by the SOUND command has been used up. The release phase carries on regardless. During this release phase the volume of the note can be made to fade away by giving AR a negative value. It will carry on decreasing by this amount per time period until it eventually reaches 0 or another note is placed on that channel. The eventual target volume is 0, the decrease per step is AR and the whole phase is independent of the duration parameter of the note.

```
    10 REM PROGRAM XI
    20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,0,0,120,60
    30 SOUND 1,1,50,240
```

As you can see from Program XI, if AS and AD are both 0 and there is no other note on that channel queue, the note carries on and on. This is because there's no decline to 0 volume in the release phase. Hence the dummy note I've put in line 40 of the previous programs – without it the note is endless and would confuse all the examples.

As I said, it's a strange part of the amplitude envelope. It's mainly meant to mimic the dying away of a note on a musical instrument. You'll notice that if there's a note following it on the same channel the envelope never enters the release phase. It just gets on with the next note. There are times when we might want to give a note a release phase, despite other notes behind it in the same queue.

We can force the note to enter its release phase, despite any following notes, by putting a dummy note after it such as:

```
SOUND &1001,0,0,0
```

This makes the T parameter of the SOUND command equal to one, which forces the previous note to enter the release phase. See the second chapter on sound if you don't know where the T parameter goes or why we've got a & in the SOUND command.

I find the sustain and release phases a nuisance most of the time, so I just use the values of −127 for each. This effectively switches them off because as soon as the note enters either phase the volume is decreased to 0, whatever it was before. Program XII shows this in action.

```
   10 REM PROGRAM XII
   20 ENVELOPE 1,100,0,0,0,0,0,0,30,-
30,-127,-127,120,60
   30 SOUND 1,1,50,240
```

And that's the end. I've covered the SOUND and ENVELOPE commands and their parameters. The rest is up to you.

YOU may be able to program, but are you getting the most from your BBC Micro? Does it sit silently in the corner, displaying forlorn messages in black and white? If so, then you're wasting the BBC Micro's huge potential.

Getting Started in Sound and Graphics is the answer.

In the same straightforward style as their highly praised series in *The Micro User,* Michael Noels, Paul Jones and Nigel Peters fully explore the computer's astonishing sound and graphics capabilities.

Assuming only a rudimentary knowledge of Basic the authors thoroughly explain all the programming methods involved, ranging from the simple to the most complex.

From triangles to teletext, straight lines to multiplane images, squeaks to sonatas – you'll find all the techniques you need in this easy to understand book.

ISBN 0-948104-00-7

9 780948 104008

£5.95