Chapter Four

The Video Display

A large part of the BBC Micro's hardware and software is concerned with producing an excellent and extremely versatile video display. Indeed it is so good that many people are buying BBC Micros to use as colour video terminals to other computers! In this chapter we will examine both the hardware and the software aspects of the video display.

As with all practical arrangements of hardware and software there is a price to be paid for every advantage gained. In the case of the video display the biggest disadvantage is the large amount of memory used for the high resolution screens. As much as 20K of RAM can be used by the video display leaving only 16K of RAM for user programs and system use. As the system can use 3-4K, an applications program can find itself left with as little as 12K of RAM. Because of this need for large amounts of memory, not all modes are available on a Model A machine. If you have a Model A machine then upgrade as soon as you can because you are missing a lot! In the rest of this chapter all modes of the video display will be discussed, including those present only on the Model B.

Not all of the display modes take so much memory. Mode 7 teletext graphics take a remarkably small 1K and can produce some very good graphics in eight colours. However, the way mode 7 works is distinctly different from all the other modes so it is given a section at the end of the chapter all to itself. Whatever mode you are using there is no doubt that the best quality display is produced by a colour monitor driven by the RGB connector. However, you can still use the highest resolution graphics on a standard colour TV set. The BBC Micro working in black and white is useful but not nearly so much fun!

The video hardware

A brief description of the video hardware was given in Chapter One but without really explaining the way that the video information was stored in memory. The BBC Micro, like many others, uses memory-mapped graphics but it uses it in a way that is very different. Most machines that generate their own video output set aside an area of memory where the ASCII (or similar) codes of the characters to be displayed are stored. As each character's code can fit into eight bits, one memory location is used for every possible display position on the screen. For example, if you have a screen of 40 characters by 20 lines then you need 40 times 20 (i.e. 800) memory locations. The way in which these memory locations are made to correspond to positions on the screen varies from machine to machine. One possible arrangement is that the first memory location corresponds to the character displayed in the top left-hand corner of the screen, subsequent memory locations corresponding to screen locations to the left of the first until the end of the line is reached, with a new line starting at the far left-hand side again (see Figure 4.1). The way that the memory is associated with the different display positions on the screen is known as the *screen memory m*ap. Obviously, if you know the screen memory map for a

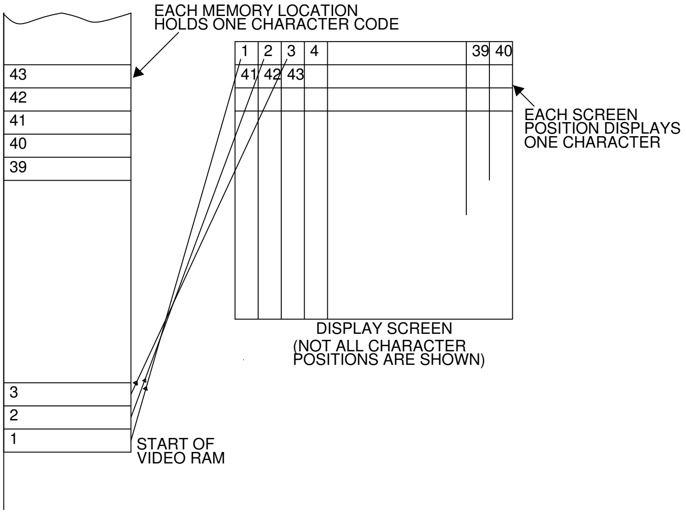
EACH MEMORY LOCATION HOLDS ONE CHARACTER CODE
43
42
41
40
39
3
2
1
START OF VIDEO RAM
1
2
3
4
39
40
41
42
43
EACH SCREEN POSITION DISPLAYS ONE CHARACTER
DISPLAY SCREEN (NOT ALL CHARACTER POSITIONS ARE SHOWN)

Fig. 4.1. The screen memory map for a 40 column screen. (Reprinted by permission of *Computing Today.)*

particular machine then you can write programs that can change the screen display by going straight to the correct memory location instead of using a PRINT or PLOT statement. This can be the quickest and sometimes simplest way of changing the screen and is often the only way of producing good moving graphics.

As mentioned earlier, the BBC Micro, in all but mode 7, uses a very different method of producing a memory-mapped screen, Instead of storing the ASCII code of the character to be displayed, the BBC Micro stores a bit pattern corresponding to the shape of the character. To make this clear it is worth considering the way other micros convert the ASCII code stored at each memory location into a character displayed on the screen.

A TV picture is built up from a series of lines and each row of characters takes a number of lines. Each character is formed from a number of dots which may be turned on or off. In this respect, the BBC Micro is no different from the rest and uses eight lines of eight dots for each character (see Figure 4.2). However, other micros produce this pattern of dots on the screen by using an extra chunk of memory that is accessible only to the video display electronics. This extra chunk of memory is normally called a *character generator* but it is nothing more than a ROM containing the information about which dots should be off, and which on, to form the image of a particular character. It is because this ROM memory is available only to the display electronics

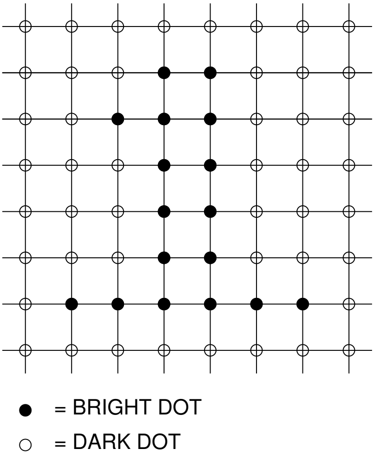

Fig. 4.2. An eight by eight dot matrix showing the character '1'. (Reprinted by permission of *Computing Today*.)

that it is normally not counted as part of the computer's memory. If you want to know how much memory is involved in a character generator all you have to do is multiply the total number of dots used to make up a character by the total number of possible characters and divide by eight. This is because the ROM has to store the dot pattern of every character that can be displayed and each dot requires one bit. For the 8 by 8 array of dots used by the BBC Micro, a ROM to generate the character set would have to be 2K bytes in size. The usual method of displaying characters on a screen using a character generator is simply to use the ASCII code stored in the computer's memory as an address to select the location in the ROM that stores the dot pattern for that character (see Figure 4.3). Instead of using this *classical* approach to video display, the BBC Micro (except in mode 7) dispenses with a character generator ROM and stores the dot pattern of the character to be displayed in RAM. The disadvantage of this method is that each screen location needs enough RAM to store all the dots for a single character - in the case of the BBC Micro this amounts to eight bytes per screen location. This means that in mode 4, for example, with 32 lines of 40 characters the total RAM required is 32 times 40 times 8 i.e. 10K bytes, and all this RAM is taken from the user RAM that you use to store programs and data. In other words, for a given screen size, the BBC Micro uses eight times the amount of screen RAM that the classical display method would require. This is because it stores the entire dot pattern for each character where the classical method stores an eight-bit code instead. The method that the BBC Micro uses is often

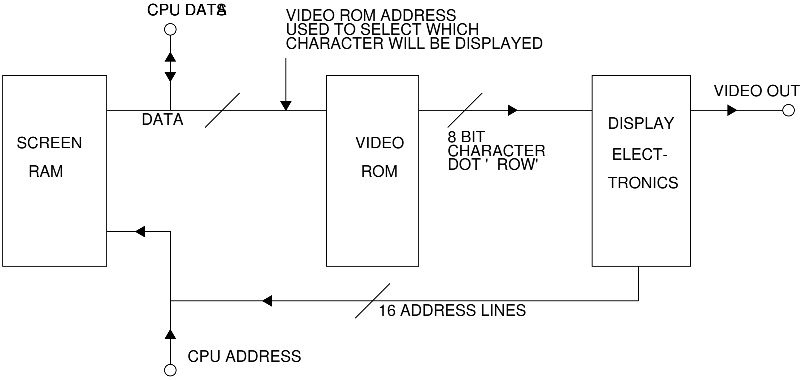
CPU DATA
VIDEO ROM ADDRESS USED TO SELECT WHICH CHARACTER WILL BE DISPLAYED
SCREEN RAM
DATA
8
VIDEO ROM
8 BIT CHARACTER DOT ' ROW'
DISPLAY ELECT-TRONICS
VIDEO OUT
16 ADDRESS LINES
CPU ADDRESS

Fig. 4.3. A 'classical' video circuit design. (Reprinted by permission of *Computing Today*.)

called a *bit-mapped* display because every bit in the screen RAM corresponds to a dot on the video screen. We can still ask for the screen memory map in this case but now it will tell us how dots on the screen correspond to bits in memory locations rather than how whole characters correspond to codes stored in memory locations.

Given the extra memory that the BBC Micro has to use to produce its display, you might be wondering what the advantages are. The main advantage is that you can produce high resolution graphics and text characters using the same hardware. Every dot on the screen corresponds to a bit in the memory location so instead of storing the dot pattern corresponding to a character, you can change individual bits in the memory to produce lines and other shapes. Also, because the same basic method is used to display characters and to produce high resolution graphics you can mix both anywhere on the screen. A second advantage is that the character set is not restricted to whatever is stored in the character generator ROM and you can therefore define new characters. These two advantages give the BBC Micro a freedom in handling both graphics and characters that is difficult to match using any other method. For comparison, the Apple uses a bit-mapped display for its high resolution graphics but uses a standard character generator for its text modes and so has difficulties in freely mixing text and graphics without extra software (shape tables). On the other hand, the PET uses a character generator for both text and graphics and so can mix them freely but the range of graphics is limited to the *graphics characters* already defined in its ROM.

Colour

The above discussion of the BBC video generator ignores the fact that each dot displayed on the screen can be any of up to sixteen colours. So far we have assumed that each bit in the video memory produces a dot on the screen. This is true in a two-colour mode such as mode 4. As each bit can be either a zero or a one, its value can select one of two colours. The colour produced by a zero bit is called the *background colour* and the one produced by a one is known as the *foreground colour*. The reason for this is that the shapes on the screen are normally formed by patterns of ones against a background of zeros. However, if you select a four- or sixteen-colour mode then one bit per dot on the screen is clearly not enough. To select one of four colours you need two bits, and to select one of sixteen colours you need four bits. Thus, in a four-colour mode (modes 1 and 5) the value of two bits in the video memory determine the colour of one dot on the screen. In the only sixteen-colour mode, mode 2, it takes the values of four bits stored in video memory to determine the colour of one dot on the screen. As a memory location can hold eight bits, a single memory location can hold the colour values of eight dots in a two-colour mode, four dots in a four-colour mode and two dots in a sixteen-colour mode. How to find the bits that correspond to a single dot is discussed in the next section on memory maps but you should now be able to see why each display mode takes the amount of memory that it does.

The screen memory map for mode 4

What the use of a bit-mapped display means for the programmer is that, unlike machines such as the PET where storing a byte in a memory location causes a complete character to appear on the screen, storing a byte in the BBC Micro's display memory causes a pattern of dots on a single line to appear. All that we need to know now is how each memory location corresponds to a screen position - in other words, the screen memory map for each mode.

For simplicity it is better to start by considering a two-colour mode such as mode 4. The best way to discover the memory map for mode 4 is via a small test program. If we start at the lowest screen address and store a byte consisting of all ones then a short line of dots will appear somewhere on the screen. If the BBC Micro uses a fairly normal screen memory map, the line should appear in either the top left or bottom right corner. If you run the following program:

```
10 MODE 4
20 ?HIMEM=&FF
30 STOP
```

then you should see a short horizontal line in the top far left-hand corner. If you don't then it's possible that it's just off the part of the screen that your TV displays and a slight adjustment of the controls should make the line visible. If this fails then try *TV 254. This will move the whole display down by two lines. The program works by first selecting mode 4 and then (in line 20) storing the hex value FF in the memory location whose address is stored in HIMEM. The variable

HIMEM stores the address of the first screen location in any mode, and FF in binary is eight ones and so produces a row of eight dots. We now know that the first (lowest) screen address corresponds to the top left-hand corner.

To find out how the rest of the screen memory map is arranged try the following program:

```
10 MODE 4
20 FOR I=0 TO 7
30 ?(HIMEM+I)=&FF
40 NEXT I
50 GOTO 50
```

This stores the hex value FF in eight consecutive memory locations. What is surprising about the result of this program is that, instead of producing a thin line eight characters long across the top of the screen, it displays a solid block about the .same size as a normal character. The screen memory map for the BBC Micro is such that the first eight memory locations form the dot matrix for the first character. The next eight form the dot matrix for the character to the right of the first and so on to the end of a line. To see the screen memory map in action try the following:

```
10 MODE 4
20 I=0
30 ?(HIMEM+I)=&FF
40 I=I+1
50 FOR J=1 TO 50
60 NEXT J
70 GOTO 30
```

You should see the screen fill up, character position by character position. You can use this program to explore the possibilities of storing graphics data directly into the screen. In most other versions of BASIC, access to memory locations is via the command POKE, which stores values in memory locations, and the function PEEK, which returns the value stored in a memory location. For this reason storing data directly to screen location is usually called POKEing the screen and, similarly, finding out what is stored in a screen location is usually called PEEKing the screen. To see that things other than solid lines can be POKEd to the screen try altering line 30 to:

```
30 ?(HIMEM+I)=RND(255)
```

and removing the delay loop formed by 50 and 60.

Using the information obtained from the above programs, we can work out a simple equation that will give the address of any screen location:

```
address=HIMEM+(X+Y*40)*8+N
```

which gives the address of the Nth line making up the character at the screen location X, Y. (N,X and Y all start from zero in the top left-hand corner,)

The screen memory map - for other modes

The memory map for any two-colour mode is easy to deduce from that of mode 4. For example, mode 3 has eighty characters to a line and 25 lines so the address of any screen location is given by:

```
address=HIMEM+(X+Y*80)*8+N
```

The corresponding expression for mode 6 with 40 characters on each of 25 lines is:

```
address=HIMEM+(X+Y*40)*8+N
```

Finally, that for mode 0 with 80 characters and 32 lines is:

```
address=HIMEM+(X+Y*80)*8+N
```

Notice that the only thing that affects the expression is the number of characters to a line. The number of lines on the screen affects the largest value of Y that can be used, of course. Modes 3 and 6 are different from the other two-colour mode in that they are text only displays. The only reason that they cannot handle graphics is that there is dead space between each line of text that cannot be affected in any way. In mode 4 a full 32 lines of character locations fill the screen completely. However, there are only 25 lines of character locations in modes 3 and 6 and these are also spread out to fill the screen. This is done by leaving a little space between each line and this is the origin of the dead space seen in each of these modes. To see this dead space try the following program:

```
10 MODE 6
20 VDU 19,0,4,0,0,0
```

The way that VDU 19 works will be discussed later but meanwhile notice that line 20 sets the background colour to blue. The dead space then shows clearly as black lines.

The complication that arises with four- and sixteen-colour modes is due to the need for more than one bit to represent each dot on the screen. How are the extra bits organised in the memory map of the other modes? The answer to this question is that the fundamental memory map outlined for mode 4 is used for all the other modes except of course that each point on the screen is now determined by a small group of bits in each memory location. For example, in mode 4 a memory location holding eight bits gives rise to eight dots but in mode 5 (a four-colour mode) the same memory location only gives rise to *four* dots. In this case each group of two bits determines which of the four colours a point will be (see Figure 4.4).

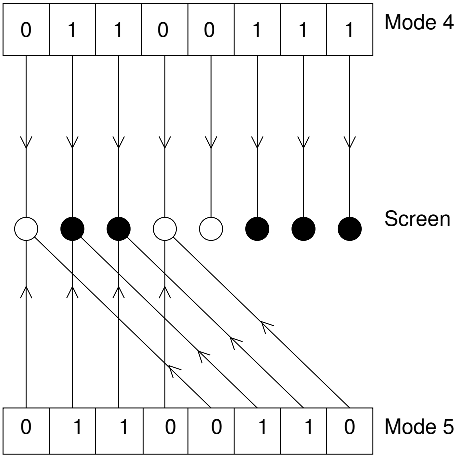

Fig. 4.4. The correspondence between memory and screen for Mode 4 and Mode 5.

The best way to investigate the memory maps of the other graphics modes is to use the programs given in the last section but change line 10 to give the required mode. In mode 5, as each block of eight memory locations now corresponds to only eight rows of four dots and each character still needs eight rows of eight dots to be displayed, it should be obvious that the storage of a single character involves two such blocks - one for the left-hand side and one for the right-hand side. Thus, the expression for the memory location corresponding to a row of dots in mode 5 (with 20 characters to a line) is:

```
address=HIMEM+(R+2*X+Y*40)*8+N
```

and in mode 1 (with 40 characters to a line):

```
address=HIMEM+(R+2*X+Y*80)*8+N
```

where X and Y are the column and line numbers of the character location, N is the number of the row of dots making up the character and R is set to 1 if it is in the right half of the character and to zero otherwise.

In the sixteen-colour mode 2 each memory location produces only two dots but the same overall pattern is maintained. Each set of eight memory locations produces a block two dots wide by eight high. Once again, a character needs an eight by eight block of dots so four of these smaller blocks are used to produce each character. If we number these smaller blocks as 0 to 3 starting at the left then the address of the memory location holding the Nth row of block B at character location X, Y is given by:

```
address=HIMEM+(4*X+B+Y*80)*8+N
```

The only question still left unanswered concerns the organisation of the bits within each memory location. In a two-colour mode, the contents of each memory location produces a row of eight dots, with the most significant bit corresponding to the left-most bit on the screen. This can be seen in Figure 4.5. In a four-colour mode the contents of a memory location control the colour of a row of four dots. The way that the bits pair to produce this row of four dots can be seen in Figure 4.6. (Notice that this is not the most obvious way to pair bits in a memory location.) Finally, the way that the eight bits in each memory location group to control the colour of two dots in a sixteen-

MSB bit 7	6	5	4	3	2	1	L.S.B. 0	
1	2	3	4	5	6	7	8	Dot number

Fig. 4.5. Bits to dots in a two-colour mode.

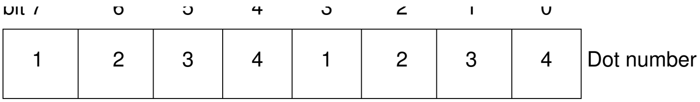

Fig. 4.6. Bits to dots in a four-colour mode.

colour mode can be seen in Figure 4.7.

All this may seem a little complicated. Compared to the way other computers work it is, but if you want to have the sort of freedom of action that the BBC Micro allows there is no other way of doing it! In practice, the use of direct memory-mapped graphics is limited to either mode 4 where it is easy, or involving assembler where everything is more difficult! Seriously though, POKEing the screen is something that is not as useful on the BBC Micro as on other machines - partly because it is more difficult except in two-colour modes and partly because the BASIC provides all sorts of features that make it unnecessary. What is more important is that a knowledge of the screen memory map allows you to find out quickly what is stored at any screen location.

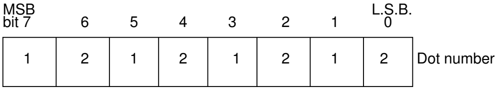

Fig. 4.7. Bits to dots in sixteen-colour mode.

PEEKing the screen

This brings us to the topic of PEEKing the screen to see what character is stored at a particular location. This is easy in machines such as the PET - all you have to do is to PEEK the screen location and this returns the ASCII code of the character stored at that position. For the BBC Micro things are not quite as easy. The first problem is that PEEKing a screen location in a two-colour mode returns the dot pattern of a row of the characters stored at the location. This is not as useful as the ASCII code because, in general, it is not enough to identify the character - for example, it is possible for two characters to have the same dot pattern in every row except one! The second problem is that for the modes that use more than two colours, even a single row of dots from a character is difficult to obtain without a number of PEEKs and quite a bit of logic.

This might make you think that screen PEEKs are not worth the trouble on the BBC machine. However, for mode 4 things are easier than they look. The general problem of deciding what character is stored at a screen location is difficult even in mode 4 but in most graphics-based applications this is more than we want to do. Instead of identifying what character from the set of all possible characters is present, it is usually enough to decide which of two or three characters is there. For example, if you are using '0' to represent one type of player and 'X' to represent another then you only have to discover if the character stored at a location is one of blank, 0 or X. This is a much easier problem as it should be possible to find a row of dots that is different in each character. If this is possible then you can tell the three characters apart by PEEKing that one row! In the case of blank, X and 0, any row will distinguish them. For example, row three corresponds to 0, 24 and 102 respectively. As we know the screen memory map for mode 4, we can write a function that will return the address of a particular row of a screen location:

```
100 DEF FNS(X,Y,N)=HIMEM+(X+40*Y)*8+N
```

FNS will return the address of the screen location corresponding to the character position X,Y and the Nth row of the character.

As an example of how to use FNS the program below prints a character on the screen at 20,10 and then prints the value of the dot pattern that makes up each row of the character.

```
10 INPUT A$
20 MODE 4
30 PRINT TAB(20,10);A$
40 FOR N=0 TO 7
50 PRINT N,?FNS(20,10,N)
60 NEXT N
70 END
100 DEF FNS(X,Y,N)=HIMEM+(X+40*Y)*8+N
```

This program can also be used to discover how any character is made up - it was used to find out the values of the third row of blank, X and 0, for example. In practice, the function FNS would typically be used in IF statements to decide what action a program should take according to what is stored at a particular location.

The character table and using the MOS to PEEK the screen

Although the BBC Micro doesn't use an external character generator

ROM, it still has to have a table of what dot patterns should be used to make each character somewhere in ROM. This character table can be found at the start of the MOS ROM, that is at address &C000. The dot pattern for each printable ASCII character is stored in this table as eight memory locations, each location corresponding to a row of dots. The first eight locations store the pattern for the ASCII blank, the next eight store the pattern for! which is the next ASCII character and this sequence continues to the last printable ASCII character, . A short program to print the patterns stored in the character table is given below.

```
10 X=&C000
20 A=?X
30 PRINT ~X;TAB(15);FNB(A)
40 X=X+1
50 IF X=8*INT(X/8) THEN PRINT
60 GOTO 20

100 DEF FNB(X)
110 LOCAL I,A$
120 A$=""
130 FOR I=7 TO 0 STEP -1
140 A$=STR$(X-2*INT(X/2))+A$
150 X=INT(X/2)
160 NEXT I
170 =A$
```

The function FNB might be useful in other programs. It converts a number to a binary number and returns the result as a string. Whenever a character is to be printed on the screen the MOS looks up the dot pattern in the table and then stores it in the correct location in the screen memory. In two-colour modes this is straight-forward and only involves transferring the bit pattern as stored in the table. There is quite a lot more work to be done in four- and sixteen-colour modes. The bit pattern stored in the table has to be used to set groups of bits in as many as 32 memory locations to the current foreground colour. This is so complicated that it is better left to the MOS! However, knowledge of where the character table is located can be used to plot dots or print other letters in the correct pattern to form very large letter displays. Apart from this application the character table could be used in reverse to discover what character was displayed at any location on the screen. This would involve comparing each of the eight memory locations that make up the character on the screen with each block of eight locations of the character table that define a character until a match is found. This is a slow and fairly difficult procedure but fortunately the MOS contains a subroutine that will carry out the search for us.

The OSBYTE call (see Chapter Three) with A=&87 will return the ASCII code of the character currently under the text cursor. The following function FNASC(X, Y) will return the ASCII code at screen location X,Y and CHR$(FNASC(X,Y)) will supply the character itself:

```
100 DEF FNASC(X,Y)
110 LOCAL C
120 X%=X
130 Y%=Y
140 A%=135
150 C=USR(&FFF4)
160 C=C AND &FFFF
170 C=C DIV &100
180 =C
```

Finding the dot pattern corresponding to an ASCII code is fast because the table is organised so that the ASCII code leads straight to the correct pattern by a simple calculation. However, going back from the pattern to the ASCII code is slower because it involves finding a match for eight bytes somewhere in the table! Even so, the User Guide claims an average time of only 120 micro-seconds to find the character!

The 6845 video generator and the ULA video processor

Now that we have a fairly full picture of the way that information is stored in the video RAM it is time to reconsider the two major components of the video circuit - the 6845 video generator and the ULA video processor. If you recall the discussion in Chapter One, you will be aware that the 6845 is responsible for supplying the correct address of the memory location that contains the bit p"ttern of the row of dots that has to be displayed on the screen. For example, in mode 4 the first visible line of the TV frame is composed of the dot pattern in the first, eighth, sixteenth, etc. screen memory locations. In addition to generating these addresses it also produces the signals that provide the timing for the TV picture and a signal that is used to produce the cursor. The operation of the 6845 is controlled by the values stored in 18 internal registers. However, these internal registers cannot be accessed directly. Instead, the 6845 has a single address register and a single data register. To write a value to any of the internal registers you have to store the number of the register in the address register and then store the value in the data register. To read a value from any register, the same procedure is followed except of course that the data register is read. In the BBC Micro, the 6845's address register is at &FE00 and its data register is at 8cFEOI. The MOS provides a way of storing information in the 6845's internal register using the statement VDU 23,0,R,X,0,0,0,0,0,0 where R is the register number and X is the value to be stored in it. To read the value stored in a register there is no choice but to use ?&FE00=reg and ?&FE01=value. You could use the 18 registers to change the mode of operation of the 6845 to produce different screen formats but because the BASIC and MOS expect to work with the particular formats corresponding to modes 0 to 7 there are lots of problems unless you intend to handle every screen function yourself. A table of 6845 registers with brief comments is given below just to give you some idea of the sort of things that can be changed. If you are really interested in using the 6845 in ' odd'ways then my advice is to get hold of a full data sheet.

Table 4.1 6845 registers

Register	*Comments*
R0	The total time taken for each horizontal scan line i.e. the horizontal sync frequency.
R1	The number of characters on a line.
R2	Position of horizontal sync pulse.
R3	Width of horizontal sync pulse.
R4	Vertical sync frequency.
R5	Vertical sync frequency.
R6	Number of character lines displayed.
R7	Vertical sync position.
R8	Interlace mode.
R9	No. of vertical dots per character.
R10	Cursor start line.
R11	Cursor stop line.
R12	(LSB) used with register 13 to specify the memory location corresponding to the first character location.
R13	(MSB) see R12.
R14	With R15 holds the address of the cursor.
R15	See R14.
R16	Light pen register.
R17	Light pen register.

The few registers that are useful to the user are made available via the MOS. For example, R14 and R15 are used by the OSBYTE call with A=86 to read the current cursor position. The cursor control registers 10 and 11 are used by VDU 23,1,0;0;0;0 which turns the cursor off and VDU 23,1,1;0;0;0;0 which restores it.

The 6845 is responsible for providing the address of the memory locations in the correct order but it is the ULA video processor that is responsible for taking the contents of the memory and converting them to dots of the correct colour. As always, it is easier to consider the two-colour case of mode 4 first. At each access a memory location provides eight bits but the TV display requires these eight bits one at a time in the correct order as the scan builds up a line across the screen. The ULA takes the eight bits from memory and feeds them out one after the other. In other words, it serialises the bits. If this was all the ULA did the BBC Micro' sgraphics facility would be severely limited. The video output of the video processor consists of the three signals R (Red), G (Green) and B (Blue). The colour displayed on the screen depends on which of the outputs are' on'For example, R on and G on produces a yellow output. All three being on produces white. You should be able to see that by taking all combinations of G and B you can produce eight different colours.

Table 4.2 Three-bit colour codes (N.B. 1 = on).

CODE	BGR	Colour
0	000	Black
1	001	Red
2	010	Green
3	011	Yellow
4	100	Blue
5	101	Magenta
6	110	Cyan
7	111	White

Now consider the problem of determining the colour of a dot displayed on the screen. In a two-colour mode each bit coming out of the serialiser could be used to select one of the eight possible colours. The only sensible way to do this is to have an extra small memory, called the *palette*, that is used to store a code for the colour to be produced when the bit is a one and another code for when the bit is a zero. The easiest code to use is a three-bit representation of which of RGB are on and which are off, as shown in Table 4.2.

Suppose, for example, that the palette has just two memory locations whose addresses are zero and one and that the code 011 is stored in zero and 101 is stored in one. Then if the output of the serialiser is fed to the palette as an address, a zero bit will produce a colour code of 011 and a one bit will produce 101. In other words, a yellow background with magenta dots will be displayed. By changing the colour codes stored in the palette any two of the eight colours can be used in a two-colour mode.

This idea extends quite easily to the four- and sixteen-colour modes. In the four-colour case we need a palette memory with four locations addressed as 00, 01, 10 and 11, each location again being capable of storing three bits of information. Now each dot on the screen is determined by two bits and this is reflected in the workings of the serialiser. Instead of changing each byte into a single stream of bits it changes each byte into two streams of bits. This is done in such a way that at any moment the two bits coming out of the serialiser are the correct two bits to determine the colour of a single dot. These two bits are used to address the palette and hence are converted into the colour codes. Obviously the four colours that appear on the screen can be selected from any of the eight available colours.

The sixteen-colour mode works in exactly the same way, the only problem being that there are only eight colours! The solution is that the extra eight colours are not really colours at all. They are just combinations of the original eight colours flashing. The palette can in fact store four bits not just the three RGB bits. The fourth bit is a *flash bit* in the sense that if it is 1 then the colour displayed on the screen alternates between its code value as stored in the palette and colour corresponding to its code value with all bits inverted. For example, if the palette held 1101, the flash bit would be set and the colour displayed would alternate between 101, magenta, and 010, green. In a sixteen-colour mode the serialiser feeds four streams of bits to be used as an address to a palette with sixteen memory locations.

After all this description it is worth summarising the details of the palette and the serialiser. The palette is a small area of memory within the video processor. Each location within the palette can store four bits which correspond to flash, B, G and R and whose state determines which of the sixteen colours is produced on the screen. The serialiser changes each byte retrieved from the video memory into either one, two or four streams of bits depending on whether the mode is a two-, four- or sixteen-colour mode. The bits forming these streams are used to address the palette RAM and so the colour codes stored in the video RAM are converted to actual colours. The relationship between the serialiser and the palette is shown in Figure 4.8.

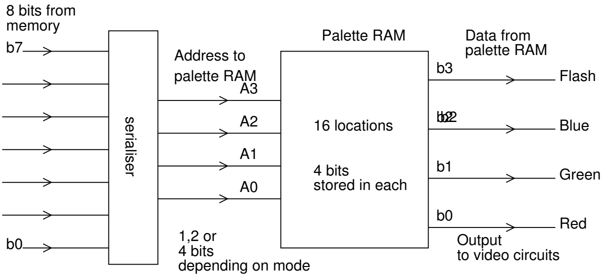

Fig. 4.8. The serialiser and palette RAM.

This changing of the colour codes stored in the video RAM to the actual colour codes produced by the palette is represented in the BBC Micro's software by the idea of logical and actual colour. Within each mode the same logical colour codes are always used. In a two-colour mode these are 0 and1, in a four-colour mode they are 0,1,2,3 and in a sixteen-colour mode they are 0 to 15. In each case these codes are simply the result of the number of bits used to control the colour of a dot in each mode; at this stage they have nothing to do with colour. They are associated with actual colours by the contents of the palette RAM. For example, if in a four-colour mode the third location of the palette RAM contained 0110, then the logical code 3 (1 I in binary) would produce a cyan dot. The contents of the palette RAM can be changed by the VDU 19 command, The form of this command is:

```
VDU 19,logical colour,actual colour,0,0,0
```

which causes 'logical colour' to produce 'actual colour' on the screen. Another way of looking at this command is that it stores the code for the actual colour in the location in the palette RAM with the address given by the code for the logical colour. For example, VDU 19,1,2,0,0,0 sets logical colour I to actual colour 2, i.e. green, or it stores the code 0010 in the palette RAM location 01 depending on how you look at it!

To read the current contents of the palette RAM you can use an

OSWORD call with A=&0B. This is described on page 462 of the User Guide and needs no further comment.

Hardware scrolling

There is one feature of the BBC Micro that is very surprising and can make use of the screen address map very difficult. When you carry out a MODE command the screen address map is set up as we have discussed and remains unaltered during the running of a program unless that program prints something that causes the screen to scroll. The action of scrolling is such a common sight on VDUs and computers that it is rare to give it a second thought. However, if you try to write a program from first principles that will scroll an entire screen you will realise what a time-consuming manoeuvre it is. Each text line of the screen must be moved up by one line. The bottom line is cleared and the top line is lost. In the BBC Micro's case, this screen shift for mode 4, if done by software, would need 10K bytes of storage to be rearranged. This would be slow, to say the least. To overcome this speed problem, scrolling is carried out by hardware which, in effect, alters the screen memory map so that the memory locations correspond to screen positions one higher. The memory corresponding to the old top line is cleared and is made to correspond to the new bottom line. In other words, following a single scroll, POKEing data into memory that was the top line produces output on the bottom line. Of course this *re-mapping* of the screen makes a non-sense of the screen mapping functions given earlier! The solution is simple - either avoid scrolling the screen following a MODE command or adjust the functions to take account of any scrolling.

To take account of scrolling it is necessary to keep a count of the number of times the screen has scrolled since the last MODE command. If the scroll count is kept in SC then the following version of FNS will work (for mode 4):

```
100 DEF FNS(X,Y,N)
110 YT=Y+SC
120 YT=YT-INT(ABS(YT)/32)*32
130 =HIMEM+(X+Y*40)*8+N
```

Notice that YT and SC are global variables and are accessible to the main program. Luckily, it is not often that the need to scroll the screen occurs in the same situation as the need to use POKE or PEEK graphics.

The way that the scrolling hardware works is quite simple. The 6845 video generator chip contains two registers, R12 and R13, which hold the address (divided by 8) of the start of the video RAM. These registers are set to the normal start of the screen following a MODE statement. However, when a scroll occurs the starting address held in the registers is increased so as to point to the start of the second line of the screen. This now becomes the new top line and every other line is moved up one. But what about the bottom line? It is now below the start of the area of memory that is displayed and so will not appear on the screen? The BBC Micro has some special electronics to overcome this problem. No matter where it starts from, the video generator always tries to display the same amount of RAM. As the highest video RAM address is always the same in any mode (&7FFF in a 32K machine and &3FFF in a 16K machine) an address produced by the video generator above the top of the video RAM area can easily be detected. When the screen display is in its initial state the video generator addresses memory from the start of the video RAM right up to the top. However, following even a single scroll, it will overshoot the top of the video RAM by exactly the amount that it has moved up. This is detected by the BBC Micro and a number is added to any such address to bring it back to the start of the video RAM. In other words, the address is made to wrap round the video RAM. This means that the previous top line isn' lost; it is now displayed (after being cleared) as .the new bottom line. The number to be added to such out-of-range addresses is different for each mode and is set by the state of the two lines C1 and C2 from VIA - A (see Chapter One).

The consequences of this hardware scrolling method are that you can set the starting point of the screen display lower by changing the values stored in R12 and R13 without any trouble but trying to increase it causes the screen to scroll. Only in this case the scroll is really a *screen* roll because the lines that appear at the bottom are not cleared first! By changing the contents of R12 and R13 by less than a whole line you can implement horizontal screen rolls. Try experimenting with the following program:

```
10 MODE 4
20 CLS
30 PRINT TAB(15,10);"HI THERE"
40 S%=HIMEM/8
50 VDU 23,0,13,I%+S% AND &00FF,0,0,0,0,0,0
60 I%=I%+1
```

```
70 IF I%>39 THEN I%=0
80 FOR J%=1 TO 1000:NEXT J%
90 GOTO 50
```

There is an easy way of disabling hardware scrolling and that is to define a text window using VDU 28. If a text window is defined then it is possible that not all of the screen will have to be scrolled. Because of this the hardware cannot be used and each line must be moved up by a software transfer. If you try this you will realise how valuable hardware scrolling is in speeding things up!

Mode 7 teletext graphics

A Mode 7 display works in a completely different way from any of the other modes. Instead of storing the bit pattern corresponding to the shape of each character to be displayed, only the ASCII code is stored. The actual bit pattern for each character is stored in an extra ROM in the video circuitry. You should recognise this as the ' classical' video circuit described at the beginning of this chapter. The only difference is that the ROM, an SAA 5050 teletext generator, produces three output signals R (Red), G (Green) and B (Blue) for an eight-colour display (with certain limitations). The advantage of using this classical arrangement is that it provides a 40 character by 25 line dispLay using only 1K of RAM. Even though only 1K of RAM is used, mode 7 provides a full upper and lower case character set and a low resolution (80 by 75) in colour! However, even though mode 7 can solve many graphics problems in less space than the other modes, it isn' t used as often as it could be. The main reason for this is that the colour control in mode 7 is by the use of control codes and the graphics take the form of block graphics characters. These are both more difficult and more restrictive than the methods used in the other modes. However, with a little understanding and practice mode 7 can be used to produce very good effects. To see the sort of results that can be achieved just look at any of the broadcast teletext pages.

As already mentioned, only the ASCII codes of the characters on the screen are stored in RAM in mode 7. This means that changing the contents of a single memory location will change the dot pattern for an entire character location. The memory map in mode 7 is:

```
memory location = HIMEM+X+Y*40
```

which is the address of the memory location corresponding to the character at X, Y. To see this in action try the following:

```
10 FOR X=0 TO 39
20 FOR Y=0 TO 24
30 ?FNS(X,Y)=ASC("A")
40 NEXT Y
50 NEXT X
60 STOP
70 DEF FNS(X,Y)=HIMEM+X+40*Y
```

Notice the use of the ASC function to store the ASCII code for the letter A in the memory location. You can use the FNS function to examine and change screen locations in mode 7 but OSBYTE call with A=k87 (see earlier) is likely to be just as fast in this case.

The colour of teletext graphics is set by the use of control codes rather than COLOUR or GCOL statements. These codes are easy to use in that they set the colour of all the teletext characters that follow until another code or a new line removes their effect. However, there is one problem in that they are not invisible on the screen. Every control code shows on the screen as a blank character the same colour as the current background. This makes changing colours in mid-line impossible without leaving a space between the two coloured zones the same colour as the current background. In other words, two areas of different foreground colours cannot meet on a line. There is, however, nothing stopping two lines of different foreground colours ' touching' . Even with this restriction it is still possible to draw very good teletext pictures. As mentioned earlier, the best way to discover more is to study the transmitted teletext pictures on the BBC (television!).

Conclusion

This chapter has tried to explore some of the hardware and software aspects of the BBC Micro' sgraphics capabilities. It has, however, barely scratched the surface of this vast topic. In particular, no mention has been made of the standard BASIC and MOS commands, such as PLOT and VDU. These are well described in the User Guide, although, of course, there is a lot to be learned through experimentation and general experience. The practical value of the information presented in this chapter about graphics memory maps for the different modes will be considered in Chapter Eight where we consider the problem of writing a screen dump program.