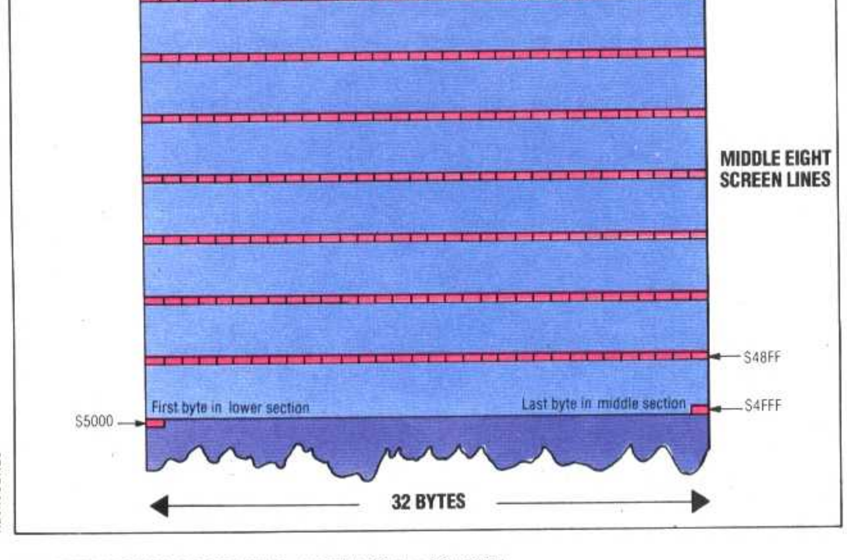
Screen Addressing

For memory-mapping purposes the Spectrum's 24-line screen is divided into three sections of eight screen lines; the details of the addressing of the middle section are shown here. Each line of the screen is divided into eight hi-res rows of 32 bytes, each bit of which corresponds to one dot pixel. Bytes are numbered sequentially along a hi-res row, and from each row in a screen line to the corresponding row in the screen line below: thus, the bytes in the eight top rows of the eight lines of one screen section have consecutive addresses. The next address is that of the first byte in the second row of the first screen line; all the bytes in the eight second hi-res rows in this screen section are addressed sequentially from this address, and so on. The address of the first byte in the top row of the first screen line of one section is one plus the address of the last byte in the eighth row of the eighth line of the preceding section. The following program for the 48K Spectrum illustrates this by POKEing a line into each byte of the hi-res screen in turn:

```
50 LET SCRNSTART=16384
60 LET SCRNEND=22527
100 FOR B=SCRNSTART TO
    SCRNEND
200 POKE B,255
300 NEXT B
```


UPPER SECTION
MIDDLE SECTION
LOWER SECTION
First byte in middle section
Last byte in upper section
$4800
$4900
$47FF
$481F
MIDDLE EIGHT SCREEN LINES
$48FF
First byte in lower section
Last byte in middle section
$5000
$4FFF
32 BYTES
KEVIN JONES

the subroutine UNDER. With D set to 0, UNDER copies the previously saved screen contents back to the screen from the table, wiping out the sprite on the screen. The program then reads the sprite position and key value, tests the key value, calls the routine to prepare for movement in the appropriate direction and then jumps to SAVSCR to save the screen contents and print the sprite on the screen, just as before.

To understand the two routines ABOVE and BELOW that prepare for the sprite's up and down movement, we must first look at the odd way in which the Spectrum matches memory addresses to screen locations. This is explained in chapter 24 of the Spectrum manual. If you look at the addresses in hexadecimal you will see that for the 256 characters in each section of the screen the low byte of the address of each of the eight bytes that make up each character is the same as the character number within the block, while the high byte of the address increases by one when we move one line of pixels down the screen. For this reason, the eight rows of pixels for one character have addresses in the range $4000 to $47FF in the top third of the screen, from $4800 to $4FFF in the middle third of the screen, and from $5000 to $57FF in the bottom third of the screen. (The '$' sign means that numbers are in hexadecimal notation — some assemblers require the use of a hash (#) symbol instead.)

The subroutine BELOW expects a screen address in the HL register pair, calculates the address of the byte that is immediately below this position on the screen, and leaves this new value in HL. If you look at the screen addresses in binary you will see that when the three low bits of H are 111 the next row of pixels down is in a different character block. BELOW tests for this first; if we are still in the same character block all that needs to be done is to add 1 to H. If we are in a different character block we have to add $20 (since there are 32 characters on a line) to L. If the new value of L is between 0 and $1F (the three high bits 000) this means we are in a different screen section. The value of HL is the current screen address.

If we are still in the same screen block we have to subtract 7 from H. You will find this becomes easier to understand if you work through the code and see what it does to the addresses shown in the table. ABOVE is similar to BELOW, but calculates the address of the pixel above the screen position.

The subroutines LMOVE and RMOVE shift the pixel bit pattern in the table left and right. Again, they are very similar, so we will just look at how LMOVE works. The accumulator is loaded with the bit position pointer, which is a single-byte number in the range 0 to 7, corresponding to the numbering of bits in a byte. 1 is then subtracted from the accumulator value to effect the move; this will also result in a number in the range 0 to 7 unless the original value of the bit position pointer was 0, in which case the accumulator will hold 255. Use of the AND 7 instruction will now ensure that the value in the accumulator remains in the 0 to 7 range. We then have a loop for the eight rows of pixels in the sprite. For each row, we load the two bytes of the table containing that row of pixels into the DE register pair and perform a 16-bit rotate left on DE. If this does not move a bit of the sprite off the top end of D into the bottom end of E we can simply store the shifted sprite pattern back into the table and go on to the next row of pixels. If we have moved a bit of the sprite from the top of D to the bottom of E we need to exchange D and E before storing them back in the table. At the end of the routine we must also subtract 1 from HL so that the sprite will be printed one position to the left.

The final subroutine in the program is PRSPRT, which does the actual printing of the sprite on the screen. This consists of two nested loops, one for the eight rows of pixels in the sprite, controlled by the C register, and one for the two bytes the row of pixels is split between, controlled by the B register. The important part of the routine is the central section that stores the bits of the sprite pattern on the screen without disturbing those bits already on the screen that should not be covered by the sprite. We have the screen address in the HL register pair and the sprite table address in the IX register. PRSPRT takes a byte of the pixel pattern of the sprite, and ORs it with what is already on the screen so that we end up with the black dots from the sprite pixel pattern superimposed on the previous screen contents.

This program is not comprehensive — in particular the maximum size of a sprite is limited to eight by eight pixels, only one sprite is allowed, and the sprite does not carry its own colours around with it. However, if you understand how this program works you will be able to extend it to include extra features. Even without modification, it is a very useful addition to many BASIC and machine code programs.