CHAPTER TEN

SPRITE GRAPHICS

Most arcade games feature a series of animated characters which movearound the screen. One or more of the characters are controlled by the player. On the BBC Micro, the only help the operating system gives to anyone trying to produce such graphics is the provision of theu ser-definable character set. Using VDU23 it is easy to produce eight-by-eight pixel shapes which can be moved around the screen.

However, this system has its limitations. Firstly, the shape producedcan only be in two colours, background and foreground; secondly, eight-by-eight is too small for most purposes; and thirdly, this method is far too slow for a fast action-packed arcade game.

The first and second problems can be solved by combining more than onecharacter to make up an object, but this makes the animation even slower.It is slow because time is taken up by the characters having to be converted from the eight-byte format of the user-defined character to theform in which they are actually stored in the screen memory. Worse still,the way a character is stored varies between the different screen modes.

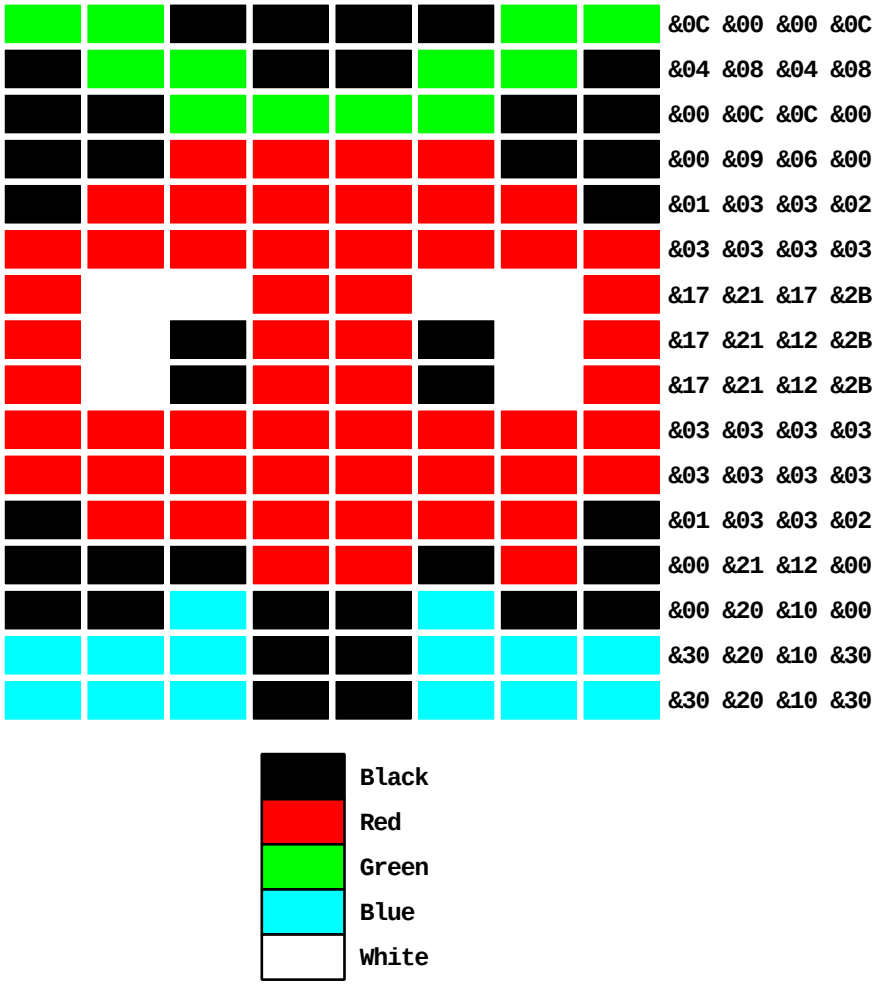
However, as most games will only use one graphics mode, it should bepossible to code the character into the relevant format for thatparticular moe when writing the program. This ready-defined characterwould be able to contain all the colours available in the mode and couldbe any size. This shape could then be stored directly on the screen in afraction of the time taken by the operating system to do the same job. These predefined characters are called SPRITES. As most arcade games workin

Mode 2, I am going to show how to use a complete sprite system from machine code in this mode.

Remember that the methods about to be detailed will not work across the Tube.

Before we embark on a complex machine code routine, we should try an experiment in BASIC − this, as we have seen, is always a good idea when writing machine code routines.

A sample sprite



A BASIC sprite routine We are going to use the sprite shown above as an example. The codng forits storage as a Mode 2 sprite is shown. Because of the way in which the

screen is laid out this coding will only work if the sprite starts on thefirst pixel of a screen memory byte. That is, the furthest left pixel ofthe sprite must be on an even-numbered pixel horizontally. If we wanted toplace it a single pixel to the right or left, we would have to totally re-code it.

However, we can easily move the sprite left and right two pixels at atime; that way, we are moving it one byte at a time. If we made the spritemove this distance every fiftieth of a second (which is the rate at whichthe image on a TV or monitor is updated), the image would appear to bemoving smoothly.

However, if we wanted the sprite to move slower than that, we wouldeither have to put up with noticing that the sprite jumps two pixels at atime, or we would have to define two sprites, one in each position, andalternate between them. This is common practice in arcade games and veryoften the two sprites are slightly different. For example, it is quiteeffective to use two sprites of a man with his legs in differentpositions. This will make him appear to walk when the sprites are placedalternatey on the screen.

For movement up and down, we need to place the bytes from the shapetable into the screen memory in different positions. As we shall see, thisis not too difficult. We can move the sprite up and down a pixel at a timewithout having to recode the sprite shape table.

The BASIC program below will place our example sprite in the top left-hand corner of the screen. The data statements at the end contain thecoded data for the sprite laid out as above. Remembe, this is an 8-by-16sprite, so it is stored as 4 bytes (= 8 pixels) wide and 16 bytes high.

```
10 MODE2
20 VDU23,1,0;0;0;0;
30 FOR A%=0 TO 1
40 FOR B%=0 TO 7
50 FOR C%=0 TO 3
60 READ D%
70 ?(&3000+A%*640+B%+C%*8)=D%
80 NEXT,,
```

```
   90 GOTO90
20000 DATA&0C,&00,&00,&0C
20010 DATA&04,&08,&04,&08
20020 DATA&00,&0C,&0C,&00
20030 DATA&00,&09,&06,&00
20040 DATA&01,&03,&03,&02
20050 DATA&03,&03,&03,&03
20060 DATA&17,&2B,&17,&2B
20070 DATA&17,&21,&12,&2B
20080 DATA&17,&21,&12,&2B
20090 DATA&03,&03,&03,&03
20100 DATA&03,&03,&03,&03
20110 DATA&01,&03,&03,&02
20120 DATA&00,&21,&12,&00
20130 DATA&00,&20,&10,&00
20140 DATA&30,&20,&10,&30
20150 DATA&30,&20,&10,&30
```

Notice in line 70 that the first eight rows of the sprite are placed from address &3000 onwards and the second eight rows are placed 640 bytes further on because they are on the next text line. Notice also that the position ALONG a row, C%, is multiplied by eight in line 70 because the columns take up eight bytes each. All this is to get around the problem caused by the complex way in which the screen is laid out. If we want to be able to move the sprite UP and DOWN a pixel at a time we are going tohave to find a way of knowing when to add 640 to the address to get us tothe next text line.

For our final sprite routine, we are going to assume that the data for the shape of the sprite is already stored in a section of memory (perhaps in an array) in the format we used in the Basic example. We can then copythis sprite table into the screen memory at whatever screen position wewant the sprite.

Before we can write a complete sprite routine, we should write an experimental version in BASIC. The easiest way to do this is as a procedure.

```
 1000 DEF PROCsprite(L%,X%,Y%,W%,H%)
```

Here L% is the location of the first byte of the sprite

shape table, X% and Y% are, respectively, the X and Y coordinates of the top left-handcorner of where we want the sprite to appear on the screen, and W% and H% are the width and height of the sprite. To make the program simpler, wewill take the X coordinate and the width W% as being in bytes (i.e. they give the number of pixels divided by two). Thus the X coordinate can take values from 0 to 79. The Y coordinate and height H% will be in pixels.

First we need to find the screen memory address of the top left-handcorner of where we want the sprite to appear. We need to split the Ycoordinate up into the text row number (from 0 to 31) and the number ofpixels down within that row (from 0 to 7). Because each row contains eightpixels vertically, bits 0 to 2 of the Y coordinate will be the number ofpixels down within the row and the other five bits will be eight times therow number. So, if we take Y% DIV 8, this will give us the text row number. As one row takes up 640 bytes of memory, we must multiply this by640 then add &3000 (which is the address of the start of the screen). Then we add eight times the X coordinate and finally the least significant three bits of the Y coordinate (Y% MOD 8). This will give us the addressof the first byte of the screen memory that we will need to change to place the sprite on the screen. We will, however, need to keep the Y% MOD8 part of the address separate as it will tell us how far down within the text row we are. So we end up with something like this:

```
1010 A%=&3000+(Y%DIV8)*640+X%*8
1020 Y%=Y%MOD8
```

Notice that in line 1020 we have altered Y% so that, instead of containingthe complete Y coordinate, it now only holds the Y coordinate within thetext row.

We now have all the information we need to fix the address of the topleft-hand corner of the sprite's intended position on the screen and so place the first row of pixels of the sprite on the screen. By adding eight to the current address (in A%+Y%) each time, we will move two pixels (=one byte in

Mode 2) to the right. We can continue to copy the table intoevery byte of the screen memory, moving one byte through the table eachtime, until we have copied the number of bytes specified by the width W%.We don't need the X coordinate again so we can use X% to count bytes.

```
1030 FOR X%=0 TO W%-1
1040 ?(A%+Y%+X%*8)=?L% 1050 L%=L%+1
1060 NEXT
```

We have now placed the first row of pixels of the sprite on the screen.Now we need to go to the next row of pixels. We can do this by adding 1 toY%. However, if Y% then equals eight, we need to move to the next text rowby adding 640 to A%. At the same time we can check to see if the addresshas passed through &8000. If so, the sprite has dropped off the bottom ofthe screen and we want the remainder of it to appear at the top of the screen to produce a 'wrap around' effect. To do this we need only subtract&5000 from A% to move back to the corresponding position at the top of thescreen.

```
1070 Y%=Y%+1
1080 IF Y%=8 THEN Y%=0:A%=A%+640:IF
     A%>&7FFF THENA%=A%-&5000
```

Next we need to check whether we have finished drawing the sprite. Theeasiest way to do this is to subtract one from the height (H%) after eachrow until H% is 0.

```
1090 H%=H%-1
1100 IF H%>0 THEN 1030
1110 ENDPROC
```

We now have a complete BASIC sprite routine.

However, this routine would be very awkward for realistic animation asit provides no means of removing the sprite again. To do this, we need to Exclusive-Or the sprite with the screen to put it on and then do the sameagain to remove it. This is the

same technique that can be used from BASICwith the VDU23 user-defined characters. For this we need to alter ourprogram. Line 1040 should now read:

```
1040 ?(A%+Y%+X%*8)=?(A%+Y%+X%*8) EOR ?L%
```

Here again, then, is the complete BASIC sprite routine.

```
1000 DEF PROCsprite(L%,X%,Y%,W%,H%)
1010 A%=&3000+(Y%DIV8)*640+X%*8
1020 Y%=Y%MOD8
1030 FOR X%=0 TO W%-1
1040 ?(A%+Y%+X%*8)=?(A%+Y%+X%*8) EOR ?L%
1050 L%=L%+1
1060 NEXT
1070 Y%=Y%+1
1080 IF Y%=8 THEN Y%=0:A%=A%+640:IF
     A%>&7FFF THENA%=A%-&5000
1090 H%=H%-1
1100 IF H%>0 THEN 1030
1110 ENDPROC
```

You may have noticed that the BASIC program is a little awkwardly written. Even so, it has been specifically written to be easy to code into machine code.

At this stage in our exploration it is hardly necessary to give the corresponding line numbrs of the sections discussed. You may find ituseful to refer to the above listing for the next section

A machine code sprite routine

For our machine code version we must use a subroutine instead of a procedure. We need a way to pass the parameters (such as the location of the table, X and Y coordinates, etcetera) to the routine. We can use the X and Y registers and some zero page locations to hold these parameters. On entry to the subroutine let's specify that the X and Y registers should initially contain the X and Y coordinates at which we want to place the sprite; that &72 should contain the width (W%) of the sprite; &73 should contain the height (H%) of the sprite; and

that &75 and &76 should contain the start address of the area of memory where we have stored the sprite shape table.

The fist thing we need to do is to calculate the screen memory address (A% in our BASIC example). In the BASIC example this was calculated like this:

```
1010 A%=&3000+(Y%DIV8)*640+X%*8
```

Let's deal with this in reverse order. The first task, then, is to calculate X%*8. The answer to this could be larger than 255 (since X% can go up to 79) so we will need two bytes to hold the answer. We are going to store the final answer to A% at &70 and &71 so let's start by putting X% in &70 and zero in &71. Thus we can treat X% as a two-byte number and multiply it by two, three times, so as to finally multiply it by eight. X% is initially contained in the X register, so the code for the multiplication by eight looks like this:

```
STX &70
LDX #0
STX &71
ASL &70
ROL &71
ASL &70
ROL &71
ASL &70
ROL &71
```

Notice that as we don't need the X coordinate again the X register is free for other use.

We have now calculated the X%*8 part of A%, so our next task is the multiplication by 640. Conveniently, the 1.2 Operating System ROM contains a 'times 640' table. This is stored starting at address &C375. It is stored as 32 entries each two bytes long (high byte then low byte). So the contents of &C375 and &C376 are 0 (for 640 x 0), the contents of &C377 and &C378 are 640 (for 640 x 1) and so on. (If you don't have a 1.2 Operating System then it is relatively easy to write a BASIC program that calculates such a table and stores it in an array.) So, to add the

result of (Y% DIV 8)*640 to A% (stored at &70 and &71) we do this:

```
TYA
AND #&F8
LSRA
LSR
TAX
LDA &C376,X
CLC
ADC &70
STA &70
LDA &C375,X
ADC &71
CLC
ADC #&30
STA &71
```

The first section masks off the bottom three bits of Y% then divides by four. This leaves the accumulator containing the equivalent of (Y%DIV8)*2. This is because each byte in the times 640 table takes up TWO bytes. Thus the result of (Y%DIV8)*640 can be looked up using the X register as a pointer and added to the rest of A% in &70 and &71. Notice that, in the last three commands, we have added &30 to the high byte. This is the equivalent of adding &3000 to the whole number. So now we have the equivalent of A% in &70 and &71. Notice that this is onlyu the address in the screen memory of the first character position at which the sprite is to be placed, not the actual byte within that character position.

The next line in our BASIC example was:

```
1020 Y%=Y%MOD8
```

This represents how far down the character position the sprite is to be placed. Converting this to machine code is very easy. We want only the least significant three bits of the Y register, so we do:

```
TYA
```

```
AND #7
STA &74
```

We have copied this answer into &74 as we are going to need to use it again. The next section of the BASIC program was a loop from 0 to W%-1 for the width of the sprite. Notice that in line 1040 we have to add to A% (now in &70 and &71) the current contets of &74 (Y%) and X%*8. The easiest way to do this in machine code is to first load the Y register with the contents of &74, then add eight to it each time around th loop. If we do this, then the values the loop takes do not matter so long as it is executed the correct number of times.

```
1040 ?(A%+Y%+X%*8)=?(A%+Y%+X%*8) EOR ?L%
```

So, we can load the X register with the width (W%) and decrement it each time round the loop until it is zero. Because we have A% in &70 and &71 and the rest of the expression in the Y register, we can use post-indexed indirect addressing to access the screen, i.e. LDA (&70),Y will be the equivalent of ?(A%+Y%+X%*8).

Next we need to load a byte from the sprite shape table into the accumulator. Because at this stage we are using both the X and Y registers, we cannot easily use any form of indexed addressing. What we do instead is use absolute addressing. In this addressing mode the address is stored as two bytes after the command byte. The first byte is the low byte, the second is the high byte. By changing these two bytes, we can use this addressing mode for looking into tables. This method is not, strictly speaking, 'legal' as it would not work if the routine were in ROM, but it does save on memory and speed.

The next section of our program will look like:

```
.row    LDY &74
        LDX &72
```

This sets the Y register to the position of the first pixel of the sprite row and puts the width W% in the

X register. Because we jump back to here at the start of each new row of the sprite, we need the label ROW.

```
.byte    LDA &FFFF
```

Above is the command that looks into the sprite shape table: the two bytes after it will be modified by the program as it runs, so the number &FFFF is unimportant. After each byte of the sprite is placed on the screen we will need to jump back to this command; so again a label, BYTE, is needed. However, before we can carry on we need to back-track to the very beginning of the routine because we need to place the address of the first byte of the sprite shape table at BYTE+1 and BYTE+2 (low, high) in place of the 'dummy' address we have specified in the assembly code. Thus the program will start by copying the first byte of the table into the screen memory.

We have specified that when calling this routine, the address of the first byte of the table should be stored at &75 and &76, so the first two lines of the program need to be:

```
.sprite  LDA &75
         STA byte+1
         LDA &76
         STA byte+2
```

Going back to where we left off, we have just loaded a byte from the sprite shape table. So now we will need to EOR this byte with the relevent byte of the screen memory and place the result back on the screen:

```
         EOR (&70),Y
         STA (&70),Y
```

Then we must add eight to the Y register to move one byte (two pixels) to the right:

```
        TYA
        CLC
        ADC #8
        TAY
```

The next line of the BASIC program was

```
1050 L%=L%+1
```

So we need to add one to the sprite shape table pointer. This is held, remember, in the two bytes after the LDA command at BYTE so we can do this:

```
        INC byte+1
        BNE nocarry
        INC byte+2
.nocarry DEX
        BNE byte
```

Notice that we then carry on doing one row of the sprite, moving from left to right, until X has reached zero and we have completed a row. The next two lines of the BASIC program were:

```
1070 Y%=Y%+1
1080 IF Y%=8 THEN Y%=0:A%=A%+640:IF
     A%>&7FFF THEN A%=A%-&5000
```

See if you can spot how this relates to the following machine code:

```
        INC &74
        LDA &74
        CMP #8
        BNE notline
        LDA #0
        STA &74
        LDA &70
        CLC
        ADC #&80
        STA &70
        LDA &71
        ADC #2
```

```
        STA &71
        CMP #&80
        BCC notline
        SEC
        SBC #&50
        STA &71
.notline ...
```

Lastly, we need to subtract one from H% and branch back to the label ROW if H% is larger than zero, or return from the subroutine, with the sprite completed, if not.

```
.notline DEC &73
        BNE row
        RTS
```

We now have a complete assembly language sprite drawing routine.

Listing 1

```
.sprite  LDA &75       DEFPROCsprite
                       (L%,X%,Y%,W%,H%)
         STA byte+1
         LDA &76
         STA byte+2
         STX &70       A% =
         LDX #0
         STX &71
         ASL &70       X% * 8
         ROL &71
         ASL &70
         ROL &71
         ASL &70
         ROL &71
         TYA           + (Y%DIV8)
         AND #&F8
         LSR A
         LSR A
         TAX
         LDA &C376,X   * 640
         CLC
         ADC &70
         STA &70
```

```
                LDA &C375,X
                ADC &71
                CLC
                ADC #&30        + &3000
                STA &71
                TYA
                AND #7          Y%=Y%MOD8
                STA &74
.row            LDY &74
                LDX &72         FOR X% = 0 TO W%-1
.byte           LDA &FFFF       ?(A%+Y%+X%*8)=
                                ?(A%+Y%+X%*8) EOR ?L%
                EOR (&70),Y
                STA (&70),Y
                TYA
                CLC
                ADC #8
                TAY
                INC byte+1      L%=L%+1
                BNE nocarry
                INC byte+2
.nocarry        DEX
                BNE byte
                INC &74         Y%=Y%+1
                LDA &74         IF Y%<>8
                CMP #8
                BNE notline     THEN 'notline'
                LDA #0          Y%=0
                STA &74
                LDA &70         A%=A%+640
                CLC
                ADC #&80
                STA &70
                LDA &71
                ADC #2
                STA &71
                CMP #&80        IF A%<&8000
                BCC notline     THEN 'notline'
                SEC
                SBC #&50
                STA &71
.notline        DEC &73         H%=H%-1
                BNE row         IF H%>0 THEN 'row'
                RTS             ENDPROC
```

This routine will only Exclusive-Or a sprite with the screen memory; it will not move it for us.

Moving sprites

To move a sprite we have to use the routine twice: first to remove the old image and then to replace it with the new image. It would make the routine easier to use if it would do all this for us. Another point is that, to make a man walk, for example, we would have to replace the previous image with a different image. Ideally our routine should be able to handle this also. Further, we should not have to tell the routine the dimensions and location of the shape table for each sprite every time we want to use it.

The last problem is solved relatively easily by assigning each sprite shape table an arbitrary number by which we can refer to it. First we reserve a section of memory in which to place a table, distinct from the sprite shape table itself, containing all the information on each sprite − an information table. The easiest way to do this is with a DIM statement − DIM sprites 255. We need four bytes of information for each sprite − two bytes for the ADDRESS of its shape table; where the data for the actual sprite shape is stored, and one byte each for HEIGHT and WIDTH of the sprite − so this command will reserve a table large enough for the information on 64 sprites. If you are not going to use this many sprites, the information table can be smaller. (Notice that we are using one table to point to a series of other tables. This is a very useful technique.)

For convenience let's assume that each sprite takes up four consecutive bytes in this information table: its two-byte shape table address (low-high) followed by its width and then its height (two separate bytes). We can now assign each sprite a number from 0 to 63 where sprite 0 is the one whose data is in the first four bytes of the information table, sprite 1 is the one whose data is the next four bytes of the information table, and so on.

Now we can enter our new, improved sprite drawing routine with just three parameters − the

sprite's X and Y coordinates on the screen in the X and Y registers, and the sprite's number in the accumulator. Therefore we must now alter the beginning of our sprite routine to handle this.

We will need the X register to point into the sprite information table so we need to save the current contents of the X register. We might as well save this as a two-byte number at &70 and &71 ready for the multiplication by 8. So the modified beginning of the sprite routine starts like this:

```
.sprite  STX &70
         LDX #0
         STX &71
```

Next we need to use the sprite number (which we have said must be in the accumulator when the routine is called) as a pointer for the information table. As each entry takes up four bytes we need to multiply the accumulator by four then transfer it to the X register ready to point into the information tabke.

```
         ASL A
         ASL A
         TAX
```

Now we can transfer the four bytes of information about the sprite (sprite shape table address, width and height) from the information table into the relevant locations ready for the rest of the program.

```
         LDA sprites,X
         STA byte+1
         LDA sprites+1,X
         STA byte+2
         LDA sprites+2,X
         STA &72
         LDA sprites+3,X
         STA &73
```

Notice that, by taking the address of the sprite shape table directly from the sprite information table and placing it in BYTE we don't need to use

locations &75 and &76 any more.

The rest of the routine is the same as before (Listing 1 on page 228), starting with:

```
ASL &70
ROL &71
ASL &70
...
...
```

We now only have to give the number of the sprite we want to place on the screen, but we still have to move the sprites by putting them on the screen and then taking them off again. This problem can be solved by keeping track of which of the sprites are being shown on the screen at any one time. The best way to do this is to have a number assigned to each moving sprite (or 'film') on the screen. This number should not be confused with the number we assigned to each sprite shape table. If, for example, we had a game with a user-controlled man and eight monsters, the man could be film 0 and the monsters could be films 1 to 8. Any of these nine films (0 to 8) could actually appear as any of the sprites we have shape tables for. And the man would probably alternate between two different sprite shapes so that he would appear to walk.

What we will do is keep a further set of tables that contain the number of the sprite shape that was last used for each film and where on the screen it was placed. To do this we will have three tables.

The table OLDS will contain the number of the last sprite used and OLDX and OLDY will contain the last X and Y coordinates for each of the films. Each film will use one byte of each of these three tables. Thus the details for film 0 will be at the first byte of each table, the details for film 1 will be at the second byte of each table, etcetera. So at the beginning of our program we will need:

```
DIM olds 255, oldx 255, oldy 255
```

If fewer films are needed then the size of the arrays can be reduced accordingly. So that we know

that none of the films are in use at the start of the program, let's set all the entries in OLDS to 255. So a 255 in OLDS tells us that the relevant films is not yet active. This means that we can't have a sprite shape numbered 255.

```
FORA%=0TO255
olds?A%=255
NEXT
```

We can now write a new routine, which will use the subroutine SPRITE, that will move a film on the screen. We can enter this routine with just the number of the film we want to move, the number of the new sprite shape we want to use for the film, and the X and Y coordinates to which we want to move the film. For convenience let's keep the sprite number in the accumulator, the X and Y coordinates in the X and Y registers, and store the film number in &75. The first job the new routine must do is to save the contents of the three registers as they will not be needed immediately. We could push these on the stack but this is slow. It is quicker to store them in zero page:

```
.move    STA &76
         STX &77
         STY &78
```

Next we need to load the X register with the film number so that we can look into the film tables. We then need to check if this is a new film (whether the relevant entry in OLDS contains 255). If so, we don't need to remove an old image and can go straight to the new image routine.

```
         LDX &75
         LDA olds,X
         CMP #255
         BEQ newfilm
```

If we find that we need to remove an old image we can load the data about the old image from the film tables. Note that because of the shortage of regis-

ters we have to temporarily save the contents of the accumulator (which contains the byte from OLDS) in yet another zero page location.

```
        STA &79
        LDA oldy,X
        TAY
        LDA oldx,Y
        TAX
        LDA &79
```

We can now call SPRITE to remove the old image. Notice that we then need to reload the X register with the current film number, as this has been lost.

```
        JSR sprite
        LDX &75
```

We now have to place the new sprite on the screen and at the same time place the data on the new sprite back in the film tables ready for the next animation. Note that the new sprite shape number and the new X and Y coordinates have to be reloaded from zero page where we stored them temporarily.

```
.newfilm LDA &78
        STA oldy,X
        TAY
        LDA &77
        STA oldx,X
        LDA &76
        STA olds,X
        LDX &77
```

Note again the problems caused by the lack of registers.

Next we need to call SPRITE again. In fact, it is better to place the whole film move routine directly before the sprite routine so that the next command will be the start of the sprite routine. This saves a JSR command and an RTS command (See full listing below).

We now have a complete sprite routine. For clarity here is a complete assembler listing of it. To use it from BASIC use the command PROCassemble at the beginning of the program to initialise it and then call MOVE with A%, X%, Y% and &75 set correctly, as explained earlier. (Listing two below.)

Listing 2

```
10000 DEFPROCassemble
10010 DIMsprites 255,olds 255,oldx 255,oldy 255
10020 FORA%=0TO255
10030 olds?A%=255
10040 NEXT
10050 DIMZ%200
10060 FORpass%=0TO2STEP2
10070 P%=Z%
10080 [OPTpass%
10090 .move    STA &76      \ Store info
10100          STX &77      \ on new sprite
10110          STY &78      \ temporarily
10120          LDX &75      \ in zero page
10130          LDA olds,X
10140          CMP #255
10150          BEQ newfilm
10160          STA &79
10170          LDA oldy,X   \ Remove old
10180          TAY          \ sprite if nec.
10190          LDA oldx,X
10200          TAX
10210          LDA &79
10220          JSR sprite
10230          LDX &75
10240 .newfilm LDA &78      \ Replace
10250          STA oldy,X   \ with
10260          TAY          \
10270          LDA &77      \ new sprite
10280          STA oldx,X   \ and store
10290          LDA &76      \ new sprite in
10300          STA olds,X   \ arrays
10310          LDX &77
10320 .sprite  STX &70      \ Main sprite
10330                       \ routine.
10340          LDX #0
```

```
10350          STX &71
10360          ASL A
10370          ASL A
10380          TAX          \ Get info
10390          LDA sprites,X \ on sprite shape
10400          STA byte+1   \ from info
10410          LDA sprites+1,X \ table
10420          STA byte+2
10430          LDA sprites+2,X
10440          STA &72
10450          LDA sprites+3,X
10460          STA &73      \ Calculate screen
10470          ASL &70      \ addr of top of
10480          ROL &71      \ LH corner
10490          ASL &70      \ of sprite
10500          ROL &71
10510          ASL &70
10520          ROL &71
10530          TYA
10540          AND #&F8
10550          LSR A
10560          LSR A
10570          TAX
10580          LDA &C376,X
10590          CLC
10600          ADC &70
10610          STA &70
10620          LDA &C375,X
10630          ADC &71
10640          CLC
10650          ADC #&30
10660          STA &71
10670          TYA
10680          AND #7
10690          STA &74
10700 .row     LDY &74      \ Plot row
10710          LDX &72      \ of pixels
10720 .byte    LDA &FFFF    \ Plot 1 pair of
10730          EOR (&70),Y  \ pixels (1 byte)
10740          STA (&70),Y
10750          TYA          \ move right to
10760          CLC          \ next pair of
10770          ADC #8       \ pixels
10780          TAY
```

```
10790          INC byte+1   \ set shape table
10800          BNE nocarry  \ pointer to
10810          INC byte+2   \ next byte
10820 .nocarry DEX          \ next pair of
10830          BNE byte     \ pixels until
10840                       \ row complete.
10850          INC &74      \ move down to
10860          LDA &74      \ next row of
10870          CMP #8       \ pixels
10880          BNE notline  \ If nec. move
10890          LDA #0       \ down to
10900          STA &74      \ next text row
10910          LDA &70
10920          CLC
10930          ADC #&80
10940          STA &70
10950          LDA &71
10960          ADC #2
10970          STA &71
10980          CMP #&80
10990          BCC notline  \ If nec. 'wrap'
11000          SEC          \ back to top
11010          SBC #&50     \ of screen
11020          STA &71
11030 .notline DEC &73      \ Move down a row
11040          BNE row      \ until sprite
11050          RTS          \ complete
11060 ]
11070 NEXT
11080 ENDPROC
```

This routine has several limitations. Firstly, it cannot be used from a second processor because it uses direct screen access rather than the official Acorn commands. For a routine to work with the second processor, only the operating system commands must be used for input and output, but this slows down a machine code program considerably. For most purposes it is better to leave this routine in the main processor and call it from the second processor.

Secondly, notice that as the Y register is used to hold 8 times the X coordinate within the sprite, the largest width of sprite the routine will handle is 32

bytes. Vertically there is no limit. Also note that by using a differently coded sprite shape table, this routine may be used in any 20K mode. However, the number of horizontal positions the sprite may be positioned at will still only be 80.

Now that we have our sprite routine we must see how it can be used. Let's write a routine to animate a small man around the screen under the control of the Z, X, /, and : keys. We will use the man we used for the BASIC example. First we must have the sprite shape table as data.

```
20000 DATA&0C,&00,&00,&0C
20010 DATA&04,&08,&04,&08
20020 DATA&00,&0C,&0C,&00
20030 DATA&00,&09,&06,&00
20040 DATA&01,&03,&03,&02
20050 DATA&03,&03,&03,&03
20060 DATA&17,&2B,&17,&2B
20070 DATA&17,&21,&12,&2B
20080 DATA&17,&21,&12,&2B
20090 DATA&03,&03,&03,&03
20100 DATA&03,&03,&03,&03
20110 DATA&01,&03,&03,&02
20120 DATA&00,&21,&12,&00
20130 DATA&00,&20,&10,&00
20140 DATA&30,&20,&10,&30
20150 DATA&30,&20,&10,&30
```

Our main program must first go into Mode 2 and turn off the cursor then assemble the machine code. Next it must load the sprite shape table into a reserved block of memory (in this case an array) and place the information about the size of the sprite and location of the shape table in the memory, in SPRITES. We are only going to use one film (film zero) so we can set the film number stored at &75 permanently to zero.

```
10 MODE 2
20 VDU23,1,0;0;0;0;
30 PROCassemble
40 DIM man% 63
50 FOR A% = 0 TO 63
```

```
 60 READ man%?A%
 70 NEXT
 80 !sprites = man%
 90 sprites?2 = 4
100 sprites?3 = 16
110 ?&75 = 0
```

Notice that the command READ man%?A% is legal. This is because expressions using the operators ?, ! and $ are treated as variable names. Next we set the starting position of the man into X% and Y% and set the sprite shape number in A% to 0.

```
120 X% = 0
130 Y% = 0
140 A% = 0
```

Next we must place the man on the screen.

```
150 CALL move
```

Next we need to alter X% and Y% according to which keys are being pressed.

```
160 IF INKEY-98 AND X%>0 X%=X%-1
170 IF INKEY-67 AND X%<76 X%=X%+1
180 IF INKEY-105 AND Y%<240 Y%=Y%+4
190 IF INKEY-73 AND Y%>0 Y%=Y%-4
```

Notice the checks to prevent our man from wandering off the edge of the screen. Next we need to go back to line 150 to remove his old image and plot his new one.

```
200 GOTO150
```

If we now add in the main sprite routine assembly code (Listing 2) the program should work.

The flicker licker

If you try the program just given, however, you will find that although it is very fast it is also very flickery.

To understand why this is we must first look at the way in which a monitor or TV works. A TV set

works on the principle of a beam of electronics which are fired at a screen. The screen is covered with a substance that glows where the beam hits it. The beam obviously cannot be aimed at the whole screen at one time yet we need the whole screen to appear to glow all the time. In fact what happens is that the beam scans in a series of horizontal lines starting at the top and working down. It completes a full screen (or 'frame') every fiftieth of a second. The result is that the eye is fooled into seeing a permanent picture. This means that if part of our computer's copy of a picture is changed it does not appear to change on the screen itself until the beam reaches that point on the screen. So the fastest you can animate a computer image without jarring the eye is 50 times a second. This, however, is also fast enough to fool the eye into seeing continuous motion.

So, our program needs to move the man exactly 50 times a second for him to move smoothly. To move the man we are taking him off the screen and then putting him back on again near where he was. This, of course, means that there is a short space of time when the man is not on the screen at all. If this blank period happens to coincide with the beam's scanning that point the man will just disappear for a whole frame. This results in the flicker that our example program suffers from.

We need, then, a method of synchronising our program with the scanning of the beam in the VDU. When the beam reaches the bottom of the screen it has to move back to the top again ready for the next frame. While this is happening the screen is 'blanked'. This vertical blanking takes about a fifth of the fiftieth-of-a-second frame cycle.

We should update the screen only during this blanking period. The computer sends a signal (call a synchronisation pulse) to the VDU approximately in the middle of this blanking to tell the VDU to move the beam back to the top again. At the same time as this occurs an interrupt is generated by the computer. When the operating system processes this it subtracts one from the contents of location &240. If we wait until the contents of location &240 change

we wait until the synchronisation pulse occurs, and we can then redraw our picture which will be ready to be 'looked' at when the scanning beam reaches it. *FX19 conveniently does this for us. So we can improve our program by inserting the line:

```
195 *FX19
```

If you try this, however, you will find that now there is a region at the top of the screen where the sprite disappears completely. This is because the synchronisation produced by *FX19 occurs roughly in the middle of the blanking period. The time between this and the beam's starting to draw the top of the screen is too small to move the sprite in. If the sprite is at the top of the screen the beam arrives at the sprite position before it has been replaced, so the sprite vanishes.

Unfortunately there is no simple solution to this problem. In many cases it is easiest to ignore the flicker completely. In most slow-moving games, such as PACMAN or DONKEY KONG, where the sprites move only every other frame the flicker will not be much of a problem. Some games, however, could be improved considerably by the removal of flicker.

To do this we need some means of synchronising the sprite movement with the beginning of the blanking period rather than the middle as produced by *FX19. We need an interrupt to occur at the very beginning of the blanking period.

To do this we can use *FX19 to start an interrupt timer in one of the VIAs which will count for just long enough for it to create an interrupt when we need it. Before we can get into the details of this, however, we need to solve another problem. If we are using several films they must all move at the same time − during the blanking period, not when the commands for each of them to move is sent. For this reason we will need a buffer, for each film, which contains the X and Y coordinates and the new sprite shape number for the next move of that film.

The easiest way to do this is to have three tables

NEWX, NEWY and NEWS. As with the three tables for the old positions each film uses one byte of each table. The main program can then set these tables to hold the movements required. When the interrupt occurs, the movements can be processed in one go. We may not want to move all the films every 50th of a second, so any one we don't want to move will have 255 in its corresponding byte of NEWS. Once each film has been moved its entry in NEWS will be set to 255 ready for the NEXT move. We could use the move routine we already have for this and add to it, but it is easier to rewrite it to handle all the sprites. Firstly we will need the actual sprite routine from LISTING 2, starting at the label SPRITE. Next we need to know how many films are in use at any moment. This can be stored in &77. (Remember that the sprite routine only uses locations &70 to &74.) Now we can start.

If the contents of &77 are, say, 7, then we will refer to bytes 0 to 6 in each of the tables, so we need to load the contents of &77 into the X register and decrement them. While we are at it we need to check whether X was 0 and, if so, end the routine. So the start of the routine will look like this:

```
.rts     RTS

.films   LDX &77
.next    DEX
         CPX #255
         BEQ rts
```

Notice that it is more efficient to place the RTS command before the start of the routine as this ensures that even if the rest of the routine is long the branch range won't be exceeded. The label NEXT can be jumped to, to process the next film.

Next we need to check whether this film needs moving. Remember we said that static films would have 255 in NEWS (the table holding the new sprite shape number for each film).

```
         LDA news,X
         CMP #255
```

```
        BEQ next
```

We now check whether there is an old sprite to remove from the screen and, if so, remove it.

```
        LDA olds,X
        CMP #255
        BEQ newfilm
        STA &76
        LDA oldy,X
        TAY
        LDA oldx,X
        STX &75
        TAX
        LDA &76
        JSR sprite
        LDX &75
```

Now we have to place the new sprite on the screen and store the data on it in the relevant bytes of OLDS, OLDX and OLDY.

```
.newfilm LDA news,X
        STA olds,X
        STA &76
        LDA newy,X
        STA oldy,X
        TAY
        LDA newx,X
        STA oldx,X
        STX &75
        TAX
        LDA &76
        JSR sprite
```

Finally we have to store 255 in the relevant bytes of NEWS (so that the program does not repeat this move in the next frame), and go back for the next film.

```
        LDX &75
        LDA #255
        STA news,X
        JMP next
```

Notice the similarities between this version of MOVE and the previous one.

Next we need a routine that will place a command into the buffer. For this purpose we will use &78 to hold the current film number, and the accumulator and the X and Y registers to hold the sprite number and X and Y coordinates respectively, as before. Thus, to instruct the sprite routine to move a film, you must place the film number in &78, the number of the sprite shape you want to use in the accumulator, and the new X and Y coordinates in the X and Y registers. Then you must call the routine MOVE.

```
.move   STX &76
        LDX &78
        STA news,X
        TYA
        STA newy,X
        LDA &76
        STA newx,X
        RTS
```

Now we have come to the difficult bit. We need to intercept the interrupt which *FX19 uses and use it to start an interrupt timer in the system VIA. We will need an initialisation routine − first, to copy the IRQ1 vector into a spare vector, and second, to alter it to point to our own interrupt routine. While doing this we need to disable the interrupts so that no interrupt can occur when the vector is changed.

```
.init   SEI
        LDA &204
        STA &230
        LDA &205
        STA &231
        LDA #irq MOD256
        STA &204
        LDA #irq DIV256
        STA &205
```

We also need to disable the TIMER 1 interrupts and

the analogue-to-digital converter interrupts in the system VIA. These tend to cause trouble by interrupting at awkward moments. Doing this means the analogue-to-digital converter interrupts and the centisecond clock will not work. However, if you want a clock, say for a game, it is easy to add into the IRQ routine a section which counts video sync interrupts (which occur 50 times a second) and increments a location every fiftieth interrupt so that it counts in seconds. It also means that though INKEY will work GET will not.

```
LDA #&50
STA &FE4E
CLI
RTS
```

(See chapter 4 for more information on the registers in the VIA.)

Now that we have initialised the interrupts we need a routine to handle them. This routine must appear totally 'transparent' to the operating system − it mustn't interfere with the operating system at all. For this reason it must not change any of the registers (A, X or Y). When the operating system processes an interrupt it stores the accumulator in location &FC and leaves the other two registers as they were when the interrupt occurred. If then jumps to the vector IRQ1.

We also have the problem that if another interrupt occurs while we are processing the current one, the routine will be entered again. This must not disturb the first entry. For these reasons both the contents of location &FC and the X and Y registers must be pushed onto the stack. This saves them so that they can be recovered before returning control to the operating system.

```
.irq    LDA &FC
        PHA
        TXA
        PHA
        TYA
```

```
        PHA
```

Next we must check that the interrupt that has occurred is the one we want. We can look at the VIA for the answer to this.

```
        LDA #2
        BIT &FE4D
        BEQ notsync
```

We now have to deal with the occurrence of a video sync interrupt. We must set TIMER 2 in the system VIA to count for just long enough for the scanning beam in the monitor to reach the vertical blanking period, and then produce an interrupt. So that we can make the fine adjustments later, we use the variable T% to set the time. T% can then be tuned by trial and error to get the best flicker-free picture.

```
        LDA &FE4B
        AND #&DF
        STA &FE4B
        LDA &FE4E
        ORA #&20
        STA &FE4E
        LDA #T%MOD256
        STA &FE48
        LDA #T%DIV256
        STA &FE49
```

Lastly we need to exit from the routine, remembering to replace all the registers as they were. It is VITAL that, when dealing with interrupts, the registers A, X and Y (and any variables or locations in memory that the main program may be using) hold the same values as they did when the interrupt occurred, before the interrupt-servicing routine passes control back the operating system.

```
.exit   PLA
        TAY
        PLA
        TAX
        PLA
```

```
        STA &FC
        JMP (&230)
```

Now we must check to see if a TIMER 2 interrupt has occurred.

```
.notsync LDA #&20
        BIT &FE4D
        BEQ exit
```

Then, when TIMER 2 interrupt occurs we must clear the interrupt status of the VIA ready for the next interrupt.

```
        STA &FE4D
```

We are now ready to move the films. Moving the films will take most of the blanking period so the synchronisation interrupt will happen right in the middle of the films. For this reason we must ensure that the interrupts are enabled while the films are moved. However, we must not have changed the state of the interrupts when we exit the routine. The easiest way to ensure this is to push the processor status register on the stack before changing the interrupt status and then pull it back off again just before we exit the routine.

```
        PHP
        CLI
        JSR films
        PLP
        JMP exit
```

We now have the complete routine. For simplicity of use we can dimension all the arrays inside the procedure that assembles the machine code. These arrays need not be longer than the maximum number of sprite shape tables and films that we are going to use, so we might as well specify these two numbers as procedure parameters. At the same time we can set the number of films in use (stored at &77) to the maximum number of films. If, at some point, we want to use fewer films than this number,

then we can alter the contents of &77 after using the procedure. While we are bout it we may as well clear OLDS and NEWS, ready for use, by filling them with 255. We also need to set T% to a suitable value. This value you can experiment with to get the best results.

Here, then, is the complete sprite routine with films and flicker-prevention.

Listing 3

```
10000DEFPROCassemble(NF%,NS%)
10010?&77=NF%
10020NF%=NF%-1
10030NS%=NS%*4-1
10040T%=18000
10050DIMsprites NS%,olds NF%,oldx NF%,oldy
NF%,news NF%,newx NF%,newy NF%
10060FORA%=0TONF%
10070olds?A%=255
10080news?A%=255
10090NEXT
10100DIMZ%400
10110FORpass%=0TO2STEP2
10120P%=Z%
10130[OPTpass%
10140.sprite  STX &70       \ draw a sprite
10150         LDX #0        \ given sprite
10160         STX &71       \ NOT & XY
10170         ASL A         \ coordinates
10180         ASL A
10190         TAX
10200         LDA sprites,X
10210         STA byte+1
10220         LDA sprites+1,X
10230         STA byte+2
10240         LDA sprites+2,X
10250         STA &72
10260         LDA sprites+3,X
10270         STA &73
10280         ASL &70
10290         ROL &71
10300         ASL &70
10310         ROL &71
10320         ASL &70
```

```
10330         ROL &71
10340         TYA
10350         AND #&F8
10360         LSR A
10370         LSR A
10380         TAX
10390         LDA &C376,X
10400         CLC
10410         ADC &70
10420         STA &70
10430         LDA &C375,X
10440         ADC &71
10450         CLC
10460         ADC #&30
10470         STA &71
10480         TYA
10490         AND #7
10500         STA &74
10510.row     LDY &74
10520         LDX &72
10530.byte    LDA &FFFF
10540         EOR (&70),Y
10550         STA (&70),Y
10560         TYA
10570         CLC
10580         ADC #8
10590         TAY
10600         INC byte+1
10610         BNE nocarry
10620         INC byte+2
10630.nocarry DEX
10640         BNE byte
10650         INC &74
10660         LDA &74
10670         CMP #8
10680         BNE notline
10690         LDA #0
10700         STA &74
10710         LDA &70
10720         CLC
10730         ADC #&80
10740         STA &70
10750         LDA &71
10760         ADC #2
```

```
10770         STA &71
10780         CMP #&80
10790         BCC notline
10800         SEC
10810         SBC #&50
10820         STA &71
10830.notline DEC &73
10840         BNE row
10850.rts     RTS
10860
10870.films   LDX &77       \ Move all films
10880.next    DEX           \ currently
10890         CPX #255      \ active
10900         BEQ rts
10910         LDA news,X
10920         CMP #255
10930         BEQ next
10940         LDA olds,X
10950         CMP #255
10960         BEQ newfilm
10970         STA &76
10980         LDA oldy,X
10990         TAY
11000         LDA oldx,X
11010         STX &75
11020         TAX
11030         LDA &76
11040         JSR sprite
11050         LDX &75
11060.newfilm LDA news,X
11070         STA olds,X
11080         STA &76
11090         LDA newy,X
11100         STA oldy,X
11110         TAY
11120         LDA newx,X
11130         STA oldx,X
11140         STX &75
11150         TAX
11160         LDA &76
11170         JSR sprite
11180         LDX &75
11190         LDA #255
11200         STA news,X
```

```
11210         JMP next
11220
11230.move    STX &76       \ Now
11240         LDX &78       \ move a film in
11250         STA news,X    \ next blanking
11260         TYA           \ period given
11270         STA newy,X    \ film no in &78
11280         LDA &76       \ shape in A
11290         STA newx,X    \ coords in X & Y
11300         RTS
11310
11320.init    SEI           \ Init. all
11330         LDA &204      \ interrupts
11340         STA &230      \ needed for
11350         LDA &205      \ sprite routines
11360         STA &231
11370         LDA #irq MOD256
11380         STA &204
11390         LDA #irq DIV256
11400         STA &205
11410         LDA #&50
11420         STA &FE4E
11430         CLI
11440         RTS
11450
11460.irq     LDA &FC       \ Handle any
11470         PHA           \ interrupts
11480         TXA           \ that occur
11490         PHA
11500         TYA
11510         PHA
11520         LDA #2
11530         BIT &FE4D
11540         BEQ notsync
11550         LDA &FE4B
11560         AND #&DF
11570         STA &FE4B
11580         LDA &FE4E
11590         ORA #&20
11600         STA &FE4E
11610         LDA #T%MOD256
11620         STA &FE48
11630         LDA #T%DIV256
11640         STA &FE49
```

```
11650.exit    PLA
11660         TAY
11670         PLA
11680         TAX
11690         PLA
11700         STA &FC
11710         JMP (&230)
11720.notsync LDA #&20
11730         BIT &FE4D
11740         BEQ exit
11750         STA &FE4D
11760         PHP
11770         CLI
11780         JSR films
11790         PLP
11800         JMP exit
11810]
11820NEXT
11830ENDPROC
```

Using the mover

Now we can write a program to show how the routine is used. Firstly we will need this sprite routine and the sprite shape table data from the last example. Then we need a program that will move the sprite around the screen under control of the Z, X, : and / keys as with the last example program.

First we go into Mode 2 and remove the cursor.

```
10 MODE 2
20 VDU23,1,0;0;0;0;
```

Then we need to assemble the machine code. We are only going to use one film and one sprite so these are the parameters we specify.

```
30 PROCassemble(1,1)
```

Now we need to set up the sprite table.

```
40 DIM man% 63
50 FOR A% = 0 TO 63
60 READ man%?A%
70 NEXT
```

Now we have to set up the parameters of this sprite shape (start of sprite shape table, width and height) in the right table.

```
 80 !sprites = man%
 90 sprites?2 = 4
100 sprites?3 = 16
```

We can now initialise the interrupts.

```
110 CALL init
```

Next we need to put the man on the screen. We can set both the film and the sprite number permanently to 0. And we can set up the initial X and Y coordinates. Then we can call the sprite routine to place the man on the screen.

```
120 ?&78 = 0
130 X% = 0
140 Y% = 0
150 A% = 0
160 CALL move
```

Next we want to check to see if any keys are being pressed, and alter the X and Y coordinates accordingly.

```
170 IF INKEY-98 AND X%>0 X%=X%-1
180 IF INKEY-67 AND X%<76 X%=X%+1
190 IF INKEY-105 AND Y%<240 Y%=Y%+4
200 IF INKEY-73 AND Y%>0 Y%=Y%-4
```

Now we can go back and place the film in its new position.

This sprite routine will only move the film every fiftieth of a second. If we try to move it more often than this, some positions will just be ignored and so the sprite will sometimes jump more than one position at a go. This means, of course, that we can move the film as fast or as slowly as we like and the sprite will still appear flicker free. For our purposes we want the sprite to move one jump every fiftieth

of a second. To do this we can use *FX19 to synchronise the BASIC program with the screen.

```
210 *FX19
220 GOTO 160
```

If you try this program you will find that unlike the last program it is totally flicker-free and smooth.

Try deleting line 210. You will find that although the man moves a lot faster he appears to move jerkily. There is no flicker but the movement is not smooth. In fact the smoother speeds at which to move the man turn to be multiples of 50 moves a second. This means that the man moves exactly the same distance between each frame. So if you want to make a film move faster it is better to increase the distance it moves each time rather than to try and make more moves per second.

This program seems to be perfect for our needs. However, if you use this routine to move more than two or three films you will find that the flicker prevention begins to fail. There is no practical solution to this problem that also allows fifty moves a second for all the films over a full screen. The only way to beat it is to use a smaller amount of the screen. You will find that when flicker occurs it usually occurs only at the top or bottom of the screen. If you ensure that your sprites never move into these areas then your problems are solved. Obviously, this is not always practicable; but if, for instance, you have a ‘space invaders’ type base at the bottom of the screen it would make sense to make this film zero. This is because lower-number films are processed later (film 0 is processed last) and these will be the ones that are still being processed when the scan starts at the top of the screen so causing possible flicker. By ensuring that film 0 never goes anywhere near the top of the screen you ensure that it will not flicker. If you do have trouble, try altering the value of T% set at the beginning of the procedure as the best value for this will depend on your program. Beyond this the only advice I can give is to experiment. Do not take these routines as perfect − feel free to customise them to your needs.

Anyone for tennis?

Let's now look at an example of how to use our sprite routine in a simple two-player tennis game. We will use the complete flicker-free sprite routine with films, etcetera. We will need two sprites − a ball and a bat; and three films − one for the ball and one for each bat. We will also need an extra section of assembly code specific to the game.

Because of the length of the assembly code the program will have to be split into three sections that chain each other.

The machine code will take up about 700 bytes, so, as we are using Mode 2 we can place this at address &2D00 to &2FFF. Unfortunately, not even the assembly code, let alone the BASIC section of the program, will fit in the space between PAGE and &2D00 on a disc machine. This means that to make the program work on both types of machine we will need to LOAD the assembly code in at &3000, ASSEMBLE it to &2D00 and then CHAIN the BASIC part of the program in to &1900 or &E00, depending on the filing system. Thus the complete program consists of three sections.

The first section simply loads in the assembly code at &3000.

```
10 MODE 7
20 PAGE=&3000
30 CHAIN"TENNIS1"
```

Save this 'loader' first as TENNIS.

Now we must write the assembly code. Because we are using the sprite routine for a specific purpose we can simplify it a bit. This is important to remember − WHEN USING A GENERAL ROUTINE FOR A SPECIFIC PURPOSE ALWAYS TRY AND SIMPLIFY IT SO THAT IT ONLY DOES WHAT IS NEEDED OF IT.

Because we know how many sprites and films we need we can set up the seven tables that the routine uses part of the assembly code. The table SPRITES will need to be eight bytes long to accommodate the two sprites. Each of the other six table

will need to be three bytes long to accommodate the three films. We can also initialise the routine with no films active by setting up all the bytes of OLDS and NEWS to &FF. All this data can be added to the end of the assembly code.

```
.sprites EQUD 0
         EQUD 0
.olds    EQUW &FFFF
         EQUB &FF
.oldx    EQUW 0
         EQUB 0
.oldy    EQUW 0
         EQUB 0
.news    EQUW &FFFF
         EQUB &FF
.newx    EQUW 0
         EQUB 0
         EQUW 0
         EQUB 0
```

This simplifies the first few lines of the assembler section of the program to:

```
10 ?&77=3
20 T%=18000
30 FORpass%=0TO2STEP2
40 P%=&2000
50 [OPTpass%
60 .sprite  STX &70
..
..
```

The assembly code part of the routine is as before (see LISTING 3).

One problem is that the third section − the BASIC part of the game − will need to refer to some of the labels in the assembly code, but CHAIN clears all normal variables. The answer to this problem is to copy the four labels we are going to need into integer variables A%, B%, C% and D% which are only cleared on CTRL BREAK. Then the first four lines of the BASIC section can copy them back into the normal variables for easy use.

Let's now look at the beginning of the BASIC section of the program and ignore the rest of the assembly code for a moment. As we have said, the first four lines must be:

```
10 sprites=A%
20 init=B%
30 move=C%
40 tennis=D%
```

Note that TENNIS is the extra assembly code routine we still have to write. Before we write this we should look at the main, BASIC, part of the program.

The next thing we need to do is set the PRINT format for printing the score (see page 325 of the USER GUIDE).

```
50 @%=3
```

Now we can go into Mode 2 and set HIMEM so as to safeguard the machine code that the second section of the program has placed from &2D00 to &2FFF. We can turn off the cursor and disable the copy cursor. (This otherwise tends to produce annoying results if the cursor keys are pressed while the game is being played.)

```
60 MODE2
70 HIMEM=&2D00
80 VDU23,1,0;0;0;0;
90 *FX4,1
```

The next job is to print the scores at the top of the screen in appropriate colours.

```
100 COLOUR1:COLOUR130
110 PRINT"SCORE:0    SCORE:0  "
```

Now we must draw the tennis court.

```
120 GCOL0,2:MOVE0,0:MOVE1279,0
130 PLOT85,0,12:PLOT85,1279,12
140 MOVE0,988:MOVE1279,988
```

```
150 PLOT85,0,976:PLOT85,1279,976
160 GCOL0,3:FORA%=24TO972STEP32
170 MOVE632,A%:DRAW632,A%+12
180 MOVE640,A%:DRAW640,A%+12
190 NEXT
```

We have two players, so we need two variables to store their scores. The best way to do this is with an array.

```
200 DIMscore(1)
```

Next we must set up the designs for the two sprites. For this game we only need very simple sprites. The bat can be made of a red rectangle two pixels (one byte) by 24 pixels (24 bytes). The ball will be a cyan square 2 pixels (1 byte) by four pixels (4 bytes). This mean that between them the two sprites need 28 bytes of storage. We can reserve these with the DIM command and set up the sprite with some simple BASIC.

```
210 DIMsp%28
220 FORA%=0TO23:sp%?A%=3:NEXT
230 sp%!24=&3C3C3C3C
```

Next we need to put the data on the sizes, and addresses, of these two sprite shapes in SPRITES.

```
240 !sprites=sp%+&18010000
250 sprites!4=sp%+24+&4010000
```

This makes the bat sprite 0 and the ball sprite 1. We can now initialise the sprite routine (the films have already been turned off in the assembler section).

```
260 CALLinit
```

We are now into the main section of the game. Let's specify some of the zero page locations we will use.

Locations &70 to &78 are used by the sprite routine. Let's say that &7E and &7F give the current X and Y coordinates of the ball. We can then say that &79 contains the current direction of the

ball horizontally − if it is 1, then the ball is travelling to the right, if it is 255 the ball is travelling to the left. Location &7A can contain the vertical speed of the ball. Because we can position the ball more accurately vertically than horizontally we will make the ball travel left and right at a constant speed while travelling up and down at a variable speed. This will give a range of speeds and directions the ball can take. Location &7A will contain the number of pixels moved down for each position moved along. Positive numbers will be moving down, negative numbers will be moving up.

We now need to initialise these ready for a serve. We shall say that the end that the ball appears at is random and that if it appears on the left it will subsequently travel right, and vice versa. So we now have to decide which end the ball starts. We can set the horizontal direction to either 1 or -1 first and use this to set the X coordinate of the ball either to 0 or 79. The Y coordinate will always be 12 so that the ball starts at the top.

```
270 ?&79=RND(2)*2-3
280 ?&7E=(?&79=255)*-79
290 ?&7F=12
```

We can now set the vertical speed of the ball randomly between 0 and 3. Having done this we must place the ball on the screen for the players to see. The ball is film 0.

```
300 ?&7A=RND(4)-1
310 ?&78=0
320 A%=1
330 X%=?&7E
340 Y%=?&7F
350 CALLmove
```

The bats will be at either side of the screen and will only move up and down. We can say that the Y coordinate for the left-hand bat is in &7C and for the right-hand bat is in &7D. We will start both bats halfway up the screen and, again, we need to place them on the screen. The left-hand bat is film 1 and

the right hand bat is film 2.

```
360 ?&7C=124
370 ?&7D=124
380 ?&78=1
390 A%=0
400 X%=0
410 Y%=?&7C
420 CALLmove
430 ?&78=2
440 X%=79
450 CALLmove
```

Now we have to start the game. We are going to do all the moment of the sprites from machine code. We still have to write the routine (TENNIS) which handles all the movement of ball and bats until one of the player misses the ball. When this happens the routine will exit with the number of the player who won that volley (0 for left and 1 for right) in &7B. So, we can call this routine, then update the score, then have a slight pause, and then serve the next ball. This gives us:

```
460 CALLtennis
470 score(?&7B)=score(?&7B)+1
480 PRINTTAB(6,0);score(0);TAB(17,0);
    score(1)
490 FORA%=1TO10000:NEXT
500 GOTO270
```

Now we must go back to the second section of the game and write TENNIS. The first thing this must to is wait for the vertical sync so as to make the routine move the films every fiftieth of a second. Then we must move the three sprites to their original positions.

```
.tennis  LDA #19
         JSR &FFF4
         LDA #0
         STA &78
         LDA #1
         LDX &7E
```

```
LDY &7F
JSR move
```

This moves the ball.

```
INC &78
LDA #0
LDX #0
LDY &7C
JSR move
INC &78
LDA #0
LDX #79
LDY &7D
JSR move
```

This moves the two bats.

Next, we have to alter &7E and &7F to point to the next position of the ball. To move it horizontally all we have to do is add the horizontal direction and the sign will take care of which way the ball moves. We can do the same sort of thing for the vertical direction.

```
LDA &7E
CLC
ADC &79
STA &7E
LDA &7F
CLC
ADC &7A
STA &7F
```

Next we must check to see if the ball has hit the top wall of the court. If so, then the Y coordinate will be less than 12 so that we want the ball to bounce. First, it mustn't go through the wall, so we must reposition it to be just touching the wall. The horizontal direction must remain the same but the ball must begin to travel in the opposite direction vertically with the same speed as it hit the wall vertically. This means that we want to multiply the speed by -1. The easiest way to do this is to sub-

tract it from 0.

```
        CMP #12
        BCS skip1
        LDA #12
        STA &7F
        LDA #0
        SEC
        SBC &7A
        STA &7A
```

Next we must check whether the ball has hit the bottom wall of the court. We can deal with this in a similar way.

```
.skip1  LDA &7F
        CMP #249
        BCC skip2
        LDA #248
        STA &7F
        LDA #0
        SEC
        SBC &7A
        STA &7A
```

Next we need to check whether the ball has reached the left-hand side of the screen. If so, we need to check whether it has hit the bat. If the contents of &7F fall outside the range of ?&7C-4 to ?&7C+23 then the ball has missed the bat and the right-hand player has won. Otherwise, we must make the ball bounce off the bat.

```
.skip2  LDA &7E
        CLC
        ADC #4
        CMP &7C
        BCC rwin
        LDA &7C
        CLC
        ADC #23
        CMP &7F
        BCS skip3
.rwin   LDA #1
```

```
        STA &7B
        RTS
.skip3
```

To make the ball bounce, we first have to make it travel to the right. We also have to set the vertical speed of the ball to a random value to simulate the range of shots a real player could play. We must, in a moment, write the routine RAND to do this.

```
.skip3  LDA #1
        STA &79
        JSR rand
.skip4  ...
```

For RAND we need to call the random number routine from the BASIC ROM. For BASIC I this is at &AFB6 and for BASIC II this is at &AF87. This routine creates a 32-bit random number in zero page locations &0D to &11. We need a random number between -3 and 4. (The slight bias to the direction is not enough to matter and makes the coding easier.) To get this we need to take a three-bit random number between 0 and 7 and subtract 3. To get a three-bit random number we need only take any three bits of the 33-bit random number that BASIC generates.

```
.rand   JSR &AFB7
        LDA &D
        AND #7
        SEC
        SBC #3
        STA &7A
        RTS
```

We can now check for the ball reaching the right-hand side of the screen in the same way.

```
        LDA &7E
        CMP #79
        BNE skip6
        LDA &7F
        CLC
```

```
        ADC #4
        CMP &7D
        BCC lwin
        LDA &7D
        CLC
        ADC #23
        CMP &7F
        BCS skip5
.lwin   LDA #0
        STA &7B
        RTS
.skip5  LDA #255
        STA &79
        JSR rand
```

Next we must move the bats. Before we do this we can simplify matters by writing an INKEY routine. We will use the OSBYTE equivalent of INKEY with a negative number to test individual keys. For this we need to set A to &81. Y to &FF and X to the key number. We then call &FFF4 and compare X with &FF. We can do most of this in a subroutine.

```
.key    LDA #&81
        LDY #&FF
        JSR &FFF4
        CPX #&FF
        RTS
```

If we jump to this with the key number in X, then, on exit, the zero flag will be set if the is pressed and be clear if it is not.

If the left-hand bat is not already at the bottom of the screen then we can test the Z key. If it is pressed, we can move the bat down four pixels.

```
.skip6  LDA &7C
        CMP #228
        BEQ skip7
        LDX #&9E
        JSR key
        BNE skip7
        LDA &7C
        CLC
```

```
        ADC #4
        STA &7C
```

We can do the same sort of thing for moving up under control of the A key, provided that the bat isn't already at the top.

```
.skip7  LDA &7C
        CMP #12
        BEQ skip8
        LDX #&BE
        JSR key
        BNE skip8
        LDA &7C
        SEC
        SBC #4
        STA &7C
```

We can then do the same for the right-hand bat under control of the] and SHIFT keys.

```
.skip8  LDA &7D
        CMP #228
        BEQ skip9
        LDX #&FF
        JSR key
        BNE skip9
        LDA &7D
        CLC
        ADC #4
        STA &7D
.skip9  LDA &7D
        CMP #12
        BEQ skip10
        LDX #&A7
        JSR key
        BNE skip10
        LDA &7D
        SEC
        SBC #4
        STA &7D
```

Finally, we need to go back to the beginning and repeat the whole process over and again until

someone misses the ball.

```
.skip10  JMP tennis
```

This, of course, is a very simple game; but it illustrates the techniques used for much more complicated games.

Here is a listing of all three sections of the program. Type in Listing 4 and save it as TENNIS; then type in Listing 5 and save it as TENNIS1; and then type in Listing 6 and save it as TENNIS2.

By this stage you should have become reasonably familiar with assembly code. You might like to try and follow these three listings, as I have purposely omitted the comments, and work out how they work for yourself. If you get lost, refer back to the program descriptions in this chapter.

Listing 4

```
10 MODE 7
20 PAGE=&3000
30 CHAIN"TENNIS1"
```

Listing 5

```
10 ?&77=3
20 T%=18000
30 FORpass%=0TO2STEP2
40 P%=&2000
50 [OPTpass%
60 .sprite  STX &70
70          LDX #0
80          STX &71
90          ASL A
100         ASL A
110         TAX
120         LDA sprites,X
130         STA byte+1
140         LDA sprites+1,X
150         STA byte+2
160         LDA sprites+2,X
170         STA &72
180         LDA sprites+3,X
190         STA &73
200         ASL &70
```

```
210          ROL &71
220          ASL &70
230          ROL &71
240          ASL &70
250          ROL &71
260          TYA
270          AND #&F8
280          LSR A
290          LSR A
300          TAX
310          LDA &C376,X
320          CLC
330          ADC &70
340          STA &70
350          LDA &C375,X
360          ADC &71
370          CLC
380          ADC #&30
390          STA &71
400          TYA
410          AND #7
420          STA &74
430 .row     LDY &74
440          LDX &72
450 .byte    LDA &FFFF
460          EOR (&70),Y
470          STA (&70),Y
480          TYA
490          CLC
500          ADC #8
510          TAY
520          INC byte+1
530          BNE nocarry
540          INC byte+2
550 .nocarry DEX
560          BNE byte
570          INC &74
580          LDA &74
590          CMP #8
600          BNE notline
610          LDA #0
620          STA &74
630          LDA &70
640          CLC
```

```
 650          ADC #&80
 660          STA &70
 670          LDA &71
 680          ADC #2
 690          STA &71
 700          CMP #&80
 710          BCC notline
 720          SEC
 730          SBC #&50
 740          STA &71
 750 .notline DEC &73
 760          BNE row
 770 .rts     RTS
 780 .films   LDX &77
 790 .next    DEX
 800          CPX #255
 810          BEQ rts
 820          LDA news,X
 830          CMP #255
 840          BEQ next
 850          LDA olds,X
 860          CMP #255
 870          BEQ newfilm
 880          STA &76
 890          LDA oldy,X
 900          TAY
 910          LDA oldx,X
 920          STX &75
 930          TAX
 940          LDA &76
 950          JSR sprite
 960          LDX &75
 970 .newfilm LDA news,X
 980          STA olds,X
 990          STA &76
1000          LDA newy,X
1010          STA oldy,X
1020          TAY
1030          LDA newx,X
1040          STA oldx,X
1050          STX &75
1060          TAX
1070          LDA &76
1080          JSR sprite
```

```
1090          LDX &75
1100          LDA #255
1110          STA news,X
1120          JMP next
1130 .move    STX &76
1140          LDX &78
1150          STA news,X
1160          TYA
1170          STA newy,X
1180          LDA &76
1190          STA newx,X
1200          RTS
1210 .init    SEI
1220          LDA &204
1230          STA &230
1240          LDA &205
1250          STA &231
1260          LDA #irq MOD256
1270          STA &204
1280          LDA #irq DIV256
1290          STA &205
1300          LDA #&50
1310          STA &FE4E
1320          CLI
1330          RTS
1340 .irq     LDA &FC
1350          PHA
1360          TXA
1370          PHA
1380          TYA
1390          PHA
1400          LDA #2
1410          BIT &FE4D
1420          BEQ notsync
1430          LDA &FE4B
1440          AND #&DF
1450          STA &FE4B
1460          LDA &FE4E
1470          ORA #&20
1480          STA &FE4E
1490          LDA #T%MOD256
1500          STA &FE48
1510          LDA #T%MOD256
1520          STA &FE49
```

```
1530 .exit    PLA
1540          TAY
1550          PLA
1560          TAX
1570          PLA
1580          STA &FC
1590          JMP (&230)
1600 .notsync LDA #&20
1610          BIT &FE4D
1620          BEQ exit
1630          STA &FE4D
1640          PHP
1650          CLI
1660          JSR films
1670          PLP
1680          JMP exit
1690 .sprites EQUD 0
1700          EQUD 0
1710 .olds    EQUW &FFFF
1720          EQUB &FF
1730 .oldx    EQUW 0
1740          EQUB 0
1750 .oldy    EQUW 0
1760          EQUB 0
1770 .news    EQUW &FFFF
1780          EQUB &FF
1790 .newx    EQUW 0
1800          EQUB 0
1810 .newy    EQUW 0
1820          EQUB 0
1830 .tennis  LDA #19
1840          JSR &FFF4
1850          LDA #0
1860          STA &78
1870          LDA #1
1880          LDX &7E
1890          LDY &7F
1900          JSR move
1910          INC &78
1920          LDA #0
1930          LDX #0
1940          LDY &7C
1950          JSR move
1960          INC &78
```

```
1970          LDA #0
1980          LDX #79
1990          LDY &7D
2000          JSR move
2010          LDA &7E
2020          CLC
2030          ADC &79
2040          STA &7E
2050          LDA &7F
2060          CLC
2070          ADC &7A
2080          STA &7F
2090          CMP #12
2100          BCS skip1
2110          LDA #12
2120          STA &7F
2130          LDA #0
2140          SEC
2150          SBC &7A
2160          STA &7A
2170 .skip1   LDA &7F
2180          CMP #249
2190          BCC skip2
2200          LDA #248
2210          STA &7F
2220          LDA #0
2230          SEC
2240          SBC &7A
2250          STA &7A
2260 .skip2   LDA &7E
2270          BNE skip4
2280          LDA &7F
2290          CLC
2300          ADC #4
2310          CMP &7C
2320          BCC rwin
2330          LDA &7C
2340          CLC
2350          ADC #23
2360          CMP &7F
2370          BCS skip3
2380 .rwin    LDA #1
2390          STA &7B
2400          RTS
```

```
2410 .skip3   LDA #1
2420          STA &79
2430          JSR rand
2440 .skip4   LDA &7E
2450          CMP #79
2460          BNE skip6
2470          LDA &7F
2480          CLC
2490          ADC #4
2500          CMP &7D
2510          BCC lwin
2520          LDA &7D
2530          CLC
2540          ADC #23
2550          CMP &7F
2560          BCS skip5
2570 .lwin    LDA #0
2580          STA &7B
2590          RTS
2600 .skip5   LDA #255
2610          STA &79
2620          JSR rand
2630 .skip6   LDA &7C
2640          CMP #228
2650          BEQ skip7
2660          LDX #&9E
2670          JSR key
2680          BNE skip7
2690          LDA &7C
2700          CLC
2710          ADC #4
2720          STA &7C
2730 .skip7   LDA &7C
2740          CMP #12
2750          BEQ skip8
2760          LDX #&BE
2770          JSR key
2780          BNE skip8
2790          LDA &7C
2800          SEC
2810          SBC #4
2820          STA &7C
2830 .skip8   LDA &7D
2840          CMP #228
```

```
2850          BEQ skip9
2860          LDX #&FF
2870          JSR key
2880          BNE skip9
2890          LDA &7D
2900          CLC
2910          ADC #4
2920          STA &7D
2930 .skip9   LDA &7D
2940          CMP #12
2950          BEQ skip10
2960          LDX #&A7
2970          JSR key
2980          BNE skip10
2990          LDA &7D
3000          SEC
3010          SBC #4
3020          STA &7D
3030 .skip10  JMP tennis
3040 .rand    JSR &AFB7
3050          LDA &D
3060          AND #7
3070          SEC
3080          SBC #3
3090          STA &7A
3100          RTS
3110 .key     LDA #&81
3120          LDY #&FF
3130          JSR &FFF4
3140          CPX #&FF
3150          RTS
3160 ]
3170 NEXT
3180 A%=sprites
3190 B%=init
3200 C%=move
3210 D%=tennis
3220 PAGE=&1900
3230 CHAIN"TENNIS2"
```

Listing 6

```
  10 sprites=A%
  20 init=B%
```

```
 30 move=C%
 40 tennis=D%
 50 @%=3
 60 MODE2
 70 HIMEM=&2D00
 80 VDU23,1,0;0;0;0;
 90 *FX4,1
100 COLOUR1:COLOUR130
110 PRINT"SCORE:0    SCORE:0  "
120 GCOL0,2:MOVE0,0:MOVE1279,0
130 PLOT85,0,12:PLOT85,1279,12
140 MOVE0,988:MOVE1279,988
150 PLOT85,0,976:PLOT85,1279,976
160 GCOL0,3:FORA%=24TO972STEP32
170 MOVE632,A%:DRAW632,A%+12
180 MOVE640,A%:DRAW640,A%+12
190 NEXT
200 DIMscore(1)
210 DIMsp%28
220 FORA%=0TO23:sp%?A%=3:NEXT
230 sp%!24=&3C3C3C3C
240 !sprites=sp%+&18010000
250 sprites!4=sp%+24+&4010000
260 CALLinit
270 ?&79=RND(2)*2-3
280 ?&7E=(?&79=255)*-79
290 ?&7F=12
300 ?&7A=RND(4)-1
310 ?&78=0
320 A%=1
330 X%=?&7E
340 Y%=?&7F
350 CALLmove
360 ?&7C=124
370 ?&7D=124
380 ?&78=1
390 A%=0
400 X%=0
410 Y%=?&7C
420 CALLmove
430 ?&78=2
440 X%=79
450 CALLmove
460 CALLtennis
```

```
470 score(?&7B)=score(?&7B)+1
480 PRINTTAB(6,0);score(0);TAB(17,0);score(1)
490 FORA%=1TO10000:NEXT
500 GOTO270
```

Conclusion

I hope that you know more about assembly language programming than you did when you first opened this book. You should now feel confident to alter the programs to suit your own tastes and possibly to improve them. You will only become completely fluent in assembly language if you sit down and actually create something using it − as in so many crafts, practice makes perfect!