This book is a collection of ready-made machine language routines that show how to achieve spectacular effects on the Spectrum.

Most of the routines are aimed at creating visual displays that are at the frontiers of Spectrum's capabilities. The routines include features that have never been published before, such as:

- Full Screen Horizon!

Lets you change the colour of the border and screen at any point, and move this horizon at will!

- Interrupt Driven Sprite Animation!

Animate objects with perfectly flicker-free movement by pixels!

- High Resolution Colour!

Create an area on the Spectrum with eight times the normal Spectrum's Colour Resolution!

- Full Screen Images!

Create images over all of the screen area, including all regions of the border!

This book is for programmers with some experience in machine language programming. All the routines are listed in labelled assembly language and the techniques and principles involved are fully explained.

Other routines included in this book are PRINT, PLOTTING, DRAWING and KEYBOARD SCAN. The routines are all of professional quality and do not involve any calls to the ROM. This means that the speed of your machine language programs can be dramatically increased using these routines.

This book gives a unique insight into the Spectrum and into a professional's machine language programs. All the routines are designed with maximum flexibility for inclusion into your own programs, and in addition, the listings contain exhaustive explanatory comments, so that you can 'learn by example'.

ISBN 0 86161 160 8

ISBN 0-86161-160-8

9 780861 611607

ADVANCED SPECTRUM MACHINE LANGUAGE DAVID WEBB

ADVANCED SPECTRUM MACHINE LANGUAGE

Extend your Spectrum with ready-made machine language routines

ADVANCED SPECTRUM MACHINE LANGUAGE

DAVID WEBB

Melbourne House

Published in the United Kingdom by:
Melbourne House (Publishers) Ltd.,
Church Yard,
Tring, Hertfordshire HP23 5LU.

Published in Australia by:
Melbourne House (Australia) Pty. Ltd.,
Suite 4, 75 Palmerston Crescent,
South Melbourne, Victoria, 3205.

ISBN 0 86161 160 8

Printed in Hong Kong by Colorcraft Ltd.

DCBA9876543210

Contents

Preface

Practically the only branch of Spectrum programming not extensively and comprehensively covered by a plethora of books is that of advanced Spectrum machine language programming. With this book I hope to remedy the situation.

By 'advanced' programming I mean the type of top-level machine language behind many of the most successful Spectrum games. Indeed, some of the techniques produced in this book are completely original and beyond anything seen in Spectrum games at the time of writing. As an example, I cite the high-resolution colour routines, and the suite of routines which allows you to produce full screen pictures over the border. Neither of these special effects has ever been seen in public before.

It is only fair to warn you that this book is not intended for beginners — there are plenty of good publications already available for newcomers to Spectrum machine language — and this book assumes a full understanding of Z-8Ø instruction code right from the start. This makes it possible for me to take you to the very frontiers of state-of-the-art Spectrum programming, extending them as we go. I hope you will enjoy and benefit from the experience.

I would like to acknowledge the contributions of the following people:

- Mum and Dad, for eighteen years of immeasurable patience.
- My publisher, Fred Milgrom, and all those at Melbourne House involved in the production work for this book.
- John, Deb, Brian, Dermot and Nobby for their support and encouragement.

Finally, I dedicate this book to the six of clubs; and that's known as dropping a good clanker.

DAVID M. WEBB
Exeter College,
Oxford.
February, 1984.

Introduction

Assumptions Made for the Use of This Book

The very title of this book indicates that it does not set out to teach elementary machine language. I am assuming that the reader at least has a grasp of the fundamentals of, and preferably a proficiency in, Z-8Ø Programming. It is not, however, essential to have learnt or practised machine language on the Spectrum to any great extent; all the peculiarities specific to the Spectrum will be described in detail, without assuming any previous knowledge of them.

To write anything but the shortest of machine language programs one should be using an assembler, and I am therefore presuming that you either already have one or are prepared to make the very worthwhile investment in one. All of the listings in this book are in assembly language, but I have deliberately restricted the use of 'pseudo-instructions', (i.e. those not in the standard Z-8Ø instruction set) to the ORG, DEFB, DEFW and EQU operations, which any assembler worth its tape should be capable of handling.

Your assembler should be capable of calculating forward and backward relative jumps, and of handling labels, which should preferably be six or more characters in length.

At the head of each listing will be a set of comments informing you of any parameters that the registers should hold on entry to the routine. Also listed will be any significant values held in the registers on exit, and a comment on which registers are preserved in value. Unless stated otherwise, you can assume that the alternate registers AF', BC', DE', HL', the stack pointer SP, the index registers IX and IY, and the interrupt vector register I are all preserved by the routine.

Similarly, unless stated otherwise, you should assume that the registers A, F, B, C, D, E, H and L are all destroyed by calling the routine. The program counter is, of course, preserved on the stack by a CALL.

The multitude of explanatory comments that are integrated into all but the simplest of routines in this book are provided for your own benefit, in the hope that you might gain the enlightenment of learning by example. They are, of course, entirely non-functional to the routines, and may be omitted when you enter the listings into your computer, just as one would omit BASIC REM statements to conserve memory.

Any numeric values printed herein will, by default, be in decimal, unless followed by an 'H' or the abbreviation 'Hex.' for hexadecimal, or by the word 'binary' when using base two.

You are now equipped with the knowledge necessary to use the rest of this book. One word of advice, though; it is intended that the user read in the general direction 'front to back', since many of the latter programs contain references to material printed earlier on in the book.

CHAPTER 1
Screen Addressing

I start this chapter by clarifying what is otherwise a frequent source of confusion. Throughout this book I shall refer to the Spectrum display as having twenty-four LINES, each line having eight ROWS of pixels, rather than twenty-four rows of eight pixel lines. Thus we see that the text area of the screen has 24 × 8 = 192 rows on it.

That technicality out of the way, let me continue with a discussion of how to calculate the text address of any of the 768 (24 × 32) CELLS on the screen.

It cannot have escaped your notice that the display file is laid out in a somewhat unusual way in memory. A quick POKE around with this program will show you what I mean.

```
10 REM TO DEMONSTRATE MEMORY LAY-OUT
   OF TEXT
20 FOR A=0 TO 6143
30 POKE 16384+A,255
40 NEXT A
50 PAUSE 0
```

In fact, the display file resides at addresses 4000H to 57FFH in the following manner. Each row has 32 columns in it and each column is 8 pixels wide. Since there are 8 bits in a byte, each column of each row is represented by one byte. The 32 bytes of each row are, as you would expect, stored consecutively in memory, reading from left to right. First to be stored (from address 4000H) is row 0 of line 0. Next comes row 0 of line 1, and so on down to row 0 of line 7. Then, instead of finding row 0 of line 8, we have row 1 of line 0, down to row 1 of line 7. The pattern

continues down to row 7 of line 6, then row 7 of line 7. At this stage, 2K of memory has been allocated, and we find that the entire top third of the screen is mapped.

The pattern described above is then repeated for the middle and bottom thirds of the screen, each consuming 2K of RAM. A rather happy consequence of having the three thirds of the screen in separate blocks of memory is that we can perform partial SAVE...SCREEN$ commands. The numbers required for this are as follows:

	START ADDRESS
TOP THIRD	16384
MIDDLE	18432
BOTTOM	20480

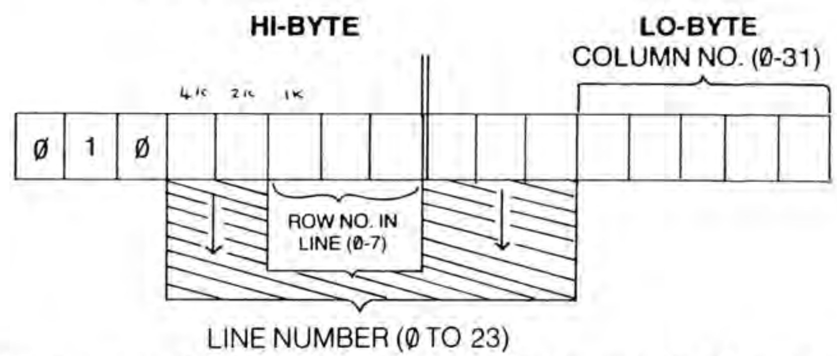
LENGTHS
2k=2048 BYTES
4k=4096 BYTES
6k=6144 BYTES

So to SAVE the bottom two thirds of the screen (of length 4K),

```
SAVE "(NAME)"CODE 18432,4096
```

Since the display file is contained within 8K of RAM from 4000H, the left-most three bits of any address in it are always 010. The complete pattern is composed as seen in this diagram:

Display File Address



The advantage of this layout is that we can step through the addresses of the eight rows of any screen cell simply by incrementing the hi-byte of the original address, rather than adding 32 to the whole address, as would be required in a 'normal' layout. This allows for that extra touch of speed in one-cell printing routines.

Now, I hear you ask, 'What's the easiest way to calculate the address of a cell?' Well, you could do a lot worse than this routine, called DF–LOC for Display File LOCation. It is worth noting that the routine will produce a logical address for any line number entered in the B register, whether or not it is in the range of Ø-23. This can be useful if, for example, you want to work down from the logical start of a 2×2 character block, the top line of which is off the top of the screen. Simply enter B=255 (for −1, the number of the line above the screen) and call DF−LOC as usual.

```
                ;ENTRY :    B=LINE,C=COLUMN
                ;PRESERVED : BC,DE
                ;EXIT:  HL=ADDRESS IN DISPLAY FILE, A=L
78              DF-LOC  LD      A,B
E6F8                    AND     ØF8H
C64Ø                    ADD     A,4ØH
67                      LD      H,A
78                      LD      A,B
E6Ø7                    AND     7
ØF                      RRCA
ØF                      RRCA
ØF                      RRCA
81                      ADD     A,C
6F                      LD      L,A
C9                      RET
```

To turn the routine into a kind of PRINT AT, you could add the line

```
    LD      (5C84H),HL
```

before the RET statement, to load the system variable DF−CC with the address to be next used by some printing routine.

Before moving on to a discussion of the attribute file, I ought to provide a routine to clear the display file, DF−CLS. It works by filling the first byte with a zero, and then using the powerful LDIR instruction to 'copy' the contents of that byte to the one above it, repeating Hex. 17FF times to fill the entire display file with zeroes. This technique should always be used whenever one needs to fill a block of memory with one particular byte.

Notice the use of the instruction

`LD (HL),L` (Since L = Ø)

which is faster and occupies less memory than

```
    LD      (HL),Ø
```

```
                ;PRESERVED : A
                ;EXIT : BC=Ø, DE=58ØØH, HL=57FFH
                ;
21ØØ4Ø          CLS-DF  LD      HL,4ØØØH
Ø1FF17                  LD      BC,17FFH
75                      LD      (HL),L
54                      LD      D,H
1EØ1                    LD      E,1
EDBØ                    LDIR
C9                      RET
```

Obviously and in a similar way to the partial SAVE SCREEN$ command described earlier in this chapter, you could adapt the routine to clear only part of the screen. Some useful numbers are these Hex. start addresses and lengths:

	ADDRESS		**VALUE FOR BC**
TOP THIRD	4ØØØH	ONE THIRD	Ø7FFH
MIDDLE	48ØØH	TWO THIRDS	ØFFFH
BOTTOM	5ØØØH	WHOLE SCREEN	17FFH

So to clear the bottom two thirds of the screen, use the lines

```
    LD      HL,48ØØH
    LD      BC,ØFFFH
```

The attributes of the Spectrum display file are those bytes which are responsible for the colours of the INK and PAPER and the state of BRIGHT and FLASH in each character cell of the screen. Consequently there are 768 bytes in the attribute file, and they are laid out logically as 24 groups of 32 bytes, one for each column, reading from left to right across the screen.

The following routine will find the address of the attributes of any cell on the screen and is called ATTLOC for ATTribute LOCator.

```
                ;ENTRY : B=LINE, C=COLUMN
                ;PRESERVED: BC,DE
                ;EXIT : HL=ADDRESS IN ATTRIBUTE FILE, A=L
78              ATTLOC  LD      A,B
CB2F                    SRA     A
CB2F                    SRA     A
CB2F                    SRA     A
```

```
C658        ADD     A,58H
67          LD      H,A
78          LD      A,B
E6Ø7        AND     7
ØF          RRCA
ØF          RRCA
ØF          RRCA
81          ADD     A,C
6F          LD      L,A
C9          RET
```

Notice the use of SRA A to sign-extend the value in A as it is shifted rightwards. This allows the routine to produce, as in DF–LOC, a logical address given any line number in B (range −128 to +127).

The logical extension of the routine to produce the ATTR (Y, X) function is to add the instruction

```
    LD      A,(HL)
```

before the RET, returning the attribute in the A register.

I have included a couple of routines to convert between display file and attribute file addresses, which may be useful if you have one but not the other. The first routine, DF–ATT will find the address of the attribute covering any byte in the display file, regardless of whether it is on row zero of a line.

```
            ;ENTRY: HL=D.F. ADDRESS
            ;PRESERVED: HL,BC
            ;EXIT: DE=ATTR. ADDRESS, A=D
7C          DF-ATT  LD      A,H
ØF                  RRCA
ØF                  RRCA
ØF                  RRCA
E6Ø3                AND     3
F658                OR      58H
57                  LD      D,A
5D                  LD      E,L
C9                  RET
```

The opposite routine is ATT–DF, which finds the address of the first row of a cell in the display file, given the address of its attributes.

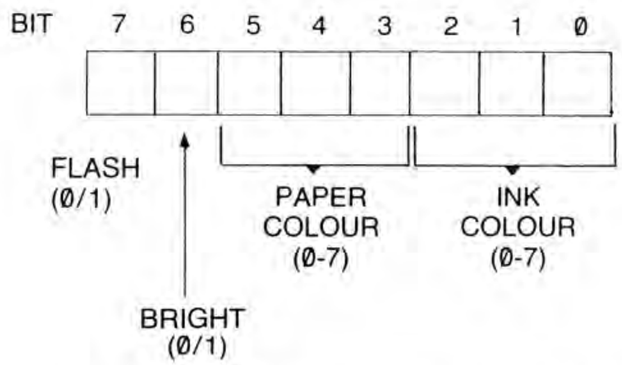

```
            ;ENTRY: HL=ATTR. ADDRESS
            ;PRESERVED: HL, BC
            ;EXIT: DE=D.F. ADDRESS, A=D
7C          ATTDF   LD      A,H
E6Ø3                AND     3
Ø7                  RLCA
Ø7                  RLCA
Ø7                  RLCA
F64Ø                OR      4ØH
57                  LD      D,A
5D                  LD      E,L
C9                  RET
```

To complete the set of 'Locator' routines, I have included a multi-purpose piece that returns the address of a cell in the display file, stores it in a variable labelled DFCC, returns the address of its attributes, and finally the actual attributes, in the accumulator. I have called the routine LOCATE.

```
            ;ENTRY: B=LINE, C=COLUMN
            ;PRESERVED: BC
            ;EXIT: HL=D.F. ADDRESS, DE=ATTR.
             ADDRESS, A=ATTR(B,C)
            ;DF-CC IS ALTERED
            ;
            DFCC    EQU     5C84H
            ;
78          LOCATE  LD      A,B
E618                AND     18H
67                  LD      H,A
CBF4                SET     6,H
ØF                  RRCA
ØF                  RRCA
ØF                  RRCA
F658                OR      58H
57                  LD      D,A
78                  LD      A,B
E6Ø7                AND     7
ØF                  RRCA
ØF                  RRCA
ØF                  RRCA
81                  ADD     A,C
6F                  LD      L,A
```

```
5F                LD      E,A
1A                LD      A,(DE)
22845C            LD      (DFCC),HL
C9                RET
```

As I mentioned, the attributes of each cell tell you its INK and PAPER colours and its state of BRIGHT and FLASH. The bit pattern associated with this is shown in the diagram.



Hence the pattern for FLASH 1, BRIGHT Ø, INK 3, PAPER 6 would be 1Ø11ØØ11, or Hex B3.

We can 'clear' the attribute file by filling it with any bit-pattern using the following routine, CLSATT, which works in a very similar way to CLS–DF.

```
                  ;ENTRY: A=SCREEN ATTRIBUTE
                  ;PRESERVED: A
                  ;EXIT: BC=Ø, DE=5BØØH, HL,5AFFH
                  ;
21ØØ58            CLSATT  LD      HL,58ØØH
Ø1FFØ2                    LD      BC,Ø2FFH
77                        LD      (HL),A
54                        LD      D,H
1EØ1                      LD      E,1
EDBØ                      LDIR
C9                        RET
```

Hence to reset the attributes to their initial condition (FLASH Ø, BRIGHT Ø, PAPER 7, INK Ø),

```
LD      A,38H
CALL    CLSATT
```


I have also included a combination of CLS—DF and CLSATT which clears the display file and sets the attributes to a given value. The routine is called CLS, for obvious reasons.

```
                  ;ENTRY: A=SCREEN ATTRIBUTE
                  ;PRESERVED: A
                  ;EXIT: BC=Ø, DE=5BØØH, HL=5AFFH
                  ;
21ØØ4Ø            CLS     LD      HL,4ØØØH
Ø1ØØ18                    LD      BC,18ØØH
75                        LD      (HL),L
54                        LD      D,H
1EØ1                      LD      E,1
EDBØ                      LDIR
77                        LD      (HL),A
Ø1FFØ2                    LD      BC,Ø2FFH
EDBØ                      LDIR
C9                        RET
```

My final point about screen addressing is the control of the border colour. The current border colour on the Spectrum screen is usually held as bits 3, 4 and 5 of the system variable BORDCR, address 5C48H.

However, altering this address from machine language has no effect on the border colour, it simply changes the value held at 5C48H. To change the border, we must reset bit Ø of the address bus and then output the new colour number on the data bus. In order not to affect any other devices attached to the user port, we SET the other seven bits of the lo-byte of the data bus, giving us the pattern 1111 111Ø in binary, or FE in Hex.

Hence to change the border colour to red (value 2) we use the sequence

```
LD A,2
OUT (ØFEH),A
```

I should point out that the border colour only consumes bits Ø, 1 and 2 of the data bus. In fact, bit 3 controls the MIC and EAR outputs, and bit 4 the loudspeaker. Altering the state of these (COMPLEMENTING them) causes a 'click' to be sent to the MIC and EAR sockets or heard on the speaker.

It is good programming practice to 'mask off' those bits not needed to alter the border colour, so that their state is maintained and no extraneous clicks are heard on the loudspeaker. If we store the last value sent to port ØFEH in the variable BORD, then to change the border colour, this program segment is applicable:

```
LD      A,(BORD)
AND     ØF8H
OR      <NEW BORDER VALUE>
OUT     (ØFEH),A
LD      (BORD),A
```

If you have a spare register pair such as HL then it is slightly faster and more economical to use

```
LD      HL,(BORD)
LD      A,(HL)
AND     ØF8H
OR      (NEW BORDER VALUE)
OUT     (ØFEH),A
LD      (HL),A
```

While on the subject, an interesting quirk of Z-8Ø machine language is that, unlike most other instructions, it is actually quicker to use immediate data rather than a register as the port number when outputting from the accumulator. That is to say,

```
OUT     (ØFEH),A        TAKES 11 T-STATES , while
OUT     (C),A           TAKES 12 T-STATES
```

This can make quite a difference when very high speed output is required, as will be seen later in this book.

CHAPTER 2
Developing a PRINT Routine

The printing routines in the Spectrum ROM are decidedly slow and tedious to use. This is a consequence of the RST 1ØH instruction being used for so many different printing functions. Once called, the routine has to decide, amongst other things, whether you are printing in the INPUT area, the top part of the screen or on the printer, whether you are trying to change the INK or PAPER colour or to execute some other control such as a TAB or AT function; and whether you are printing a 'normal' character, a 'chunky graphic' character or a user-defined graphic character. On top of all this, by the nature of BASIC, the routine also spends time carrying out a series of error-checks that we can do without in a machine-code program.

It is thus imperative that we develop our own, customised printing routine, and that is what I shall be doing in this chapter.

The process of printing a character can be broken down into three stages. First we locate the address of the character data (the eight bytes whose bit patterns define the character), then we copy this data to the required screen cell, and finally we change the attributes of that cell as required.

You are no doubt familiar with the concept of leaving certain attributes of a cell as they are by the use of INK 8, PAPER 8, FLASH 8 and BRIGHT 8 when printing a new character in that cell. These BASIC functions are easily performed in machine language by what is known as MASKING off individual bits of the 'old' attribute byte when printing a 'new' character.

We shall use a one-byte variable, ATT, to hold the new attributes for the character to be printed, and a second byte, MASK, to hold the mask for the old attributes. For each bit of the old attributes that we want preserved, we set the corresponding bit of MASK to 1. Suppose that we just want the BRIGHT bit to be masked (i.e. BRIGHT 8). Then referring to the attribute bit-pattern in chapter one, we see that BRIGHT occupies bit 6. So we set bit 6 of our mask, and have the pattern Ø1ØØ ØØØØ, or Hex. 4Ø so we equate the variable MASK to Hex. 4Ø.

If we examine the 8 values that represent the set of INK and PAPER colours more closely, we find that they are allocated to the colours in an extremely logical way. All colours are combinations of the three primary colours, blue, red and green, and each of these colours has been allocated one of the three bits in each of the INK and PAPER patterns. This makes life easier for the celebrated ULA chip, which crudely speaking has to forward these bits to the blue, red and green electron guns which then fire pixels onto your colour telly screen.

Blue is allocated to the lower of the three bits (value 1), then red (value 2) then green (value 4). Thus the INK and PAPER bit patterns look like this:

HI-BIT LO-BIT

GREEN	RED	BLUE

and the complete attribute (and mask) pattern is as given:

Flash	Bright	←	Paper	→	←	Ink	→
7	6	5	4	3	2	1	Ø
		G	R	B	G	R	B

Whenever a primary colour is required to make up another colour, then its bit is set. Thus cyan, which is a mixture of green and blue, has the binary pattern

G	R	B
1	Ø	1

or decimal 5.

White is the combination of all three primary colours, and so has the pattern

G	R	B
1	1	1

= decimal 7.

while black is a total absence of colour, and is thus represented by:

G	R	B
Ø	Ø	Ø

= decimal Ø.

Incidentally, there is no difference between bright black (with BRIGHT 1) and dark black (BRIGHT Ø). You can check that this is the case by doing

```
BORDER Ø: PAPER Ø: BRIGHT 1: CLS
```

On entering this line the border and text area will become indistinguishable in colour.

An advantage of having the colour values allocated so logically is that you can mask off individual primary colours or combinations of them, rather than being restricted to masking all three bits, which is all that INK 8 and PAPER 8 offer from BASIC.

Having obtained values for ATT and MASK, we are ready to create the new attribute byte for a cell. Loading the old attributes into the accumulator, the quickest way to perform the operation is:

```
XOR     ATT
AND     MASK
XOR     ATT
```

The new attribute byte is now ready to be placed in the attribute file. You will see such a program fragment appearing in the following print routine.

We shall store the base address of the character data to be used in the two-byte variable BASE. This should point at row zero of the first character in your set. I have made provision for up to 256 characters, and since each requires 8 bytes, a full set will need 2K of memory. In the unlikely event that more than 256 characters are required, you will need two bytes to represent each character, and you could use the hi-byte to indicate which value of BASE is required, then call the same print routine.

The Spectrum has the bit-patterns for 96 of its characters in ROM, ranging from SPACE to the copyright symbol. This data occupies the last 768 bytes of the ROM, from address 3DØØH.

The system variable CHARS on page 173 of the Spectrum manual is quoted as holding '256 less than address of character set'. This may seem a little odd, until you realize that the first character, a space, is represented numerically by 2ØH, or 32 decimal. Now 32 × 8 = 256, so by setting CHARS TO 256 less than the address of the character set, the Spectrum can find the address of a character just by multiplying its code by 8 (rows) and adding it to CHARS.

In the PRINT 1 routine you will notice that I have initialized our variable BASE to 3CØØH, so normal CODE values for the Spectrum character set are applicable. I have also placed ATT immediately before MASK, so that the two can be accessed with one LD instruction.

```
                ;ENTRY: A=CHAR. CODE
                ;PRESERVED: C
                ;EXIT: B=Ø, DE=ATTRIBUTE ADDRESS
                ;
ØØ3C            BASE    DEFW    3CØØH
ØØ4Ø            DFCC    DEFW    4ØØØH
38              ATT     DEFB    38H
ØØ              MASK    DEFB    Ø
                ;

                ;CONSTRUCT CHARACTER DATA ADDRESS
6F              PRINT1  LD      L,A
26ØØ                    LD      H,Ø
29                      ADD     HL,HL
29                      ADD     HL,HL
29                      ADD     HL,HL
ED5BØØØØ                LD      DE,(BASE)
19                      ADD     HL,DE
                ;TAKE DISPLAY FILE ADDRESS
ED5BØ2ØØ                LD      DE,(DFCC)
Ø6Ø8                    LD      B,8
                ;PRINT CHARACTER ROW BY ROW
7E.             NXTROW  LD      A,(HL)
12                      LD      (DE),A
23                      INC     HL
14                      INC     D
1ØFA                    DJNZ    NXTROW
                ;CONSTRUCT ATTRIBUTE ADDRESS
7A                      LD      A,D
ØF                      RRCA
```



```
ØF                      RRCA
ØF                      RRCA
3D                      DEC     A
E6Ø3                    AND     3
F658                    OR      58H
57                      LD      D,A
2AØ4ØØ                  LD      HL,(ATT)
                ;H=MASK, L=ATTRIBUTE
                ;TAKE OLD ATTRIBUTE
1A                      LD      A,(DE)
                ;CONSTRUCT NEW ONE
AD                      XOR     L
A4                      AND     H
AD                      XOR     L
                ;REPLACE ATTRIBUTE
12                      LD      (DE),A
                ;FINALLY SET DFCC TO NEXT PRINT POSITION
21Ø2ØØ                  LD      HL,DFCC
34                      INC     (HL)
CØ                      RET     NZ
23                      INC     HL
7E                      LD      A,(HL)
C6Ø8                    ADD     A,8
77                      LD      (HL),A
C9                      RET
```

I made the assumption in the above routine that DF–CC had already been set to the correct address in the display file, by use of the LOCATE routine in Chapter 1 or otherwise. You will notice that it is updated to the next print position every time a character is printed.

As an example to show PRINT 1 in action, here is a routine to print out the character set from the ROM (codes 2ØH to 7FH). You will need the LOCATE routine of Chapter One.

```
                ;PRINT1 DEMO
                ;SET DFCC TO (Ø,Ø)
Ø1ØØØØ                  LD      BC,Ø
CDØØØØ                  CALL    LOCATE
                ;SET BASE TO POINT AT ROM CHARACTER SET
21ØØ3C                  LD      HL,3CØØH
22ØØØØ                  LD      (BASE),HL
                ;NOW PRINT FROM CODE 2ØH TO 7FH
```

```
            ;REMEMBER PRINT1 PRESERVES THE C REGISTER
ØE2Ø                LD      C,2ØH
79          LOOP    LD      A,C
CDØØØØ              CALL    PRINT1
ØC                  INC     C
            ;WHEN C BECOMES NEGATIVE (>7FH) THEN
            ;ALL IS DONE
F2ØEØØ              JP      P,LOOP
C9                  RET
```

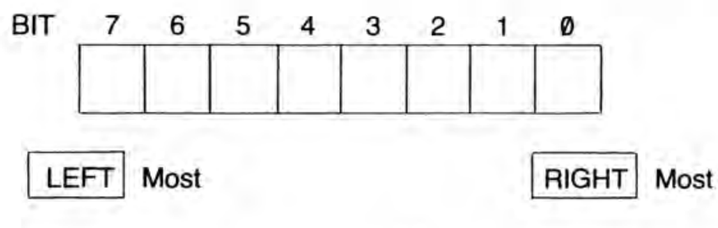

CHAPTER 3

Plotting and Drawing

Many are the times when you may need to plot stars and positions on maps or draw laser rays. Here I shall develop a routine to let you plot anywhere on the screen, and one to draw, using absolute coordinates, a line between any two points on the screen. The routines are slightly faster than those in the BASIC ROM, since I have removed a lot of cumbersome error-checks.

The procedure needed to plot a point on the screen can be broken down into four stages. First, we find the address of the byte in the display file which holds the bit representing our 'target' pixel on the screen. Then we find the address of the attributes of the cell which the target pixel is in. We then change the attributes according to our standard variables ATT and MASK, and finally we make the actual plot, taking careful note of whether INVERSE 1 or OVER 1 is required.

I should remark at this point that given a byte in the display file, which represents one row of a cell, the left-most bit (BIT 7) represents the left-most pixel, while the right-most bit (BIT Ø) represents the right-most pixel. A frequent programming error is to think that bit Ø represents the first (left-most) pixel. The easiest way to remember this is by visualizing this diagram:



You will see that in the PLOT routine I find the attribute address by converting the display file address. This is much easier and quicker than going back to our original coordinates and calculating the address from them.

The routine decides whether to use OVER or INVERSE by referring to two flags in a one-byte variable called PFLAG. This is equivalent to the BASIC system variable of the same name at address 23697 (5C91 Hex.) We signify OVER 1 by setting bit 1 of PFLAG, and INVERSE 1 by setting bit 3.

Notice that I am employing a new coordinate system on the screen, which makes it easier and faster to calculate addresses. The top left-hand corner is (Ø, Ø) while the bottom right-hand corner (including the INPUT lines) is (255, 191). Here, then, is the routine.

```
            ;ENTRY: H=X, L=Y
            ;PRESERVED: HL
            ;EXIT: DE=ADDRESS OF PIXEL IN DISPLAY FILE
            ;A=(DE), C=(PFLAG)
            ;TOP LEFT-HAND CORNER = (Ø,Ø)
            ;
            ;BIT 1 (PFLAG)=OVER
            ;BIT 3 (PFLAG)=INVERSE
            ;
            ;
38          ATT     DEFB    38H
ØØ          MASK    DEFB    Ø
ØØ          PFLAG   DEFB    Ø
            ;
            ;FIND ADDRESS OF REQUIRED BYTE IN D.F.
7D          PLOT    LD      A,L
E6CØ                AND     ØCØH
1F                  RRA
37                  SCF
1F                  RRA
ØF                  RRCA
AD                  XOR     L
E6F8                AND     ØF8H
AD                  XOR     L
57                  LD      D,A
7C                  LD      A,H
Ø7                  RLCA
```



```
Ø7                  RLCA
Ø7                  RLCA
AD                  XOR     L
E6C7                AND     ØC7H
AD                  XOR     L
Ø7                  RLCA
Ø7                  RLCA
5F                  LD      E,A
            ;ADDRESS IS STORED IN DE
D5                  PUSH    DE
            ;FIND ATTRIBUTE ADDRESS
7A                  LD      A,D
ØF                  RRCA
ØF                  RRCA
ØF                  RRCA
E6Ø3                AND     3
F658                OR      58H
57                  LD      D,A
ED4BØØØØ            LD      BC,(ATT)
            ;CHANGE ATTRIBUTE
1A                  LD      A,(DE)
A9                  XOR     C
AØ                  AND     B
A9                  XOR     C
12                  LD      (DE),A
            ;RETRIEVE D.F. ADDRESS
D1                  POP     DE
7C                  LD      A,H
E6Ø7                AND     7
47                  LD      B,A
Ø4                  INC     B
            ;B HOLDS (TARGET BIT NUMBER)+1
3EFE                LD      A,ØFEH
            ;ROTATE A WINDOW TO THE TARGET BIT
ØF          PLOOP   RRCA
1ØFD                DJNZ    PLOOP
47                  LD      B,A
3AØ2ØØ              LD      A,(PFLAG)
4F                  LD      C,A
            ;TAKE BYTE FROM D.F.
1A                  LD      A,(DE)
            ;CHECK FOR OVER 1
```

```
CB49                BIT     1,C
2ØØ1                JR      NZ,OVER1
AØ                  AND     B
        ;CHECK FOR INVERSE 1
CB59    OVER1       BIT     3,C
2ØØ2                JR      NZ,INV1
A8                  XOR     B
2F                  CPL
12      INV1        LD      (DE),A
C9                  RET
```

The final portion of the routine, that does the actual 'plotting', merits a more detailed explanation. Ignoring what happens to the other seven bits in our byte, since they are always unchanged in the end, let us examine the behaviour of the 'target' bit.

```
    AND     B
```

makes the target zero if OVER Ø is selected. OVER 1 causes the instruction to be skipped.

```
    BIT     3,C
    JR      NZ,INV1
```

causes a jump to the end if INVERSE 1 was selected, leaving the bit as it was if OVER 1 was selected, or zero (PAPER) if OVER Ø was selected.

Finally, having 'narrowed down' our selections to INVERSE Ø,

```
    XOR     B
    CPL
```

which may be thought of more clearly as

```
    XOR     B
    XOR     ØFFH
```

leaves the bit complemented in the case of OVER 1, or set in the case of OVER Ø. The byte is now replaced in the display file.

★ ★ ★ ★

Following on from the PLOT routine, I decided it would be a worthwhile exercise to develop our own machine-code DRAW routine. Although it uses the same algorithm as that in the Spectrum ROM, the routine will run somewhat faster due to the use of 'optimized' code and fewer error traps.

I shall be using absolute coordinates as the parameters for the routine, rather than the relative ones used in Spectrum BASIC. This is largely a matter of personal preference, and the routine is easily altered to provide relative drawing. As for the PLOT routine, the coordinates are assigned thus:

In order to discuss the drawing algorithm, let us assume that the line is from (X1, Y1) to (X2, Y2), both inclusive. Before going any further, the routine plots the first point on the line. Now some preparation is needed to decide which direction to draw in.

If (X2−X1) is positive, we will be drawing to the right. If it is negative, then to the left. Similarly, if (Y2−Y1) is positive, we will be drawing downwards, otherwise the direction will be upwards.

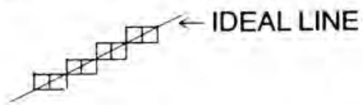
The routine loads the D and E registers with the unit changes in X and Y (respectively) related to the direction along each axis. For example, if we were drawing upwards and to the right, then the change in X would be positive (D = 1) and that in Y would be negative (E = −1). So the routine produces:

```
    DE=Ø1FFH
```

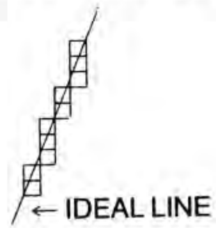
We now need to consider exactly how a line can be formed by making unit steps either horizontally, vertically or diagonally between points on a grid. A moment's thought reveals that, unless the two points at either end of the line are on a diagonal of pixels, we will need to combine a mixture of 'straight' steps and diagonal steps to draw the line. The routine allocates the BC pair as follows:

```
    B=ABS(X2-X1)      C=ABS(Y2-Y1)
```

If B is greater than C, then we will need a mixture of horizontal and diagonal steps, thus:



While if B is less than C, a mixture of VERTICAL and diagonal steps is required, such as this:



The routine decides whether we need vertical or horizontal steps, and stores the required value of DE, as explained earlier, in the variable VHSTP. The direction of the diagonal steps is stored in DIASTP.

The values in B and C are stored so that B is greater than or equal to C. We are now just about ready to begin plotting the line, which will have (B−C) straight steps, and C diagonal steps. In order to ensure that the straight and diagonal steps are evenly distributed, the following procedure is used.

B is copied to H, and then halved and copied to L. Now, entering the loop, C is added to L, and if the result equals or exceeds B, then it is reduced by B and a diagonal step is taken. Otherwise, a straight step is taken. A point is plotted, the counter in H is decremented and the loop is repeated until the line is complete.

What this loop can be thought of as, is continually adding C to itself and taking a diagonal step every time the result passes a new multiple of B. The reason L is initialized to ½ B is simply to ensure that the line is straight at the beginning.

Here at last is the DRAW routine; the most useful exit values are the coordinates of the last point plotted, in HL. Since HL also holds the coordinates of the first point of a line on entry to the routine, we see that it can be used without alteration between calls to the DRAW routine, to draw lines to, and then from, the same point. This facility will be heavily utilized in the demonstration program following the routine.

```
             ;ENTRY: H=X1, L=Y1, B=X2, C=Y2
             ;SO DRAWS FROM (H,L) TO (B,C) INCLUSIVE
             ;PRESERVED: NONE
             ;EXIT: DE IS THE ADDRESS OF THE LAST
             ;       PIXEL PLOTTED WHICH IS AT (H,L).
             ;       B IS THE GREATER, AND C IS THE
             ;       LESSER OF ABS(X2-X1) AND ABS(Y2-Y1).
             ;
             ;
0100         DIASTP  DEFW    1
0100         VHSTP   DEFW    1
             ;
```



```
C5       DRAW    PUSH    BC
         ;PLOT (X1,Y1)
CD0000   DRAW    CALL    PLOT
C1               POP     BC
110101           LD      DE,0101H
         ;DE HOLDS THE DIRECTION OF THE
         ;X AND Y STEPS
78               LD      A,B
         ;GOING LEFT (-1), OR RIGHT (+1)?
94               SUB     H
3004             JR      NC,X2X1
15               DEC     D
15               DEC     D
ED44             NEG
47       X2X1    LD      B,A
         ;B HOLDS NO. OF STEPS IN X
79               LD      A,C
95               SUB     L
         ;GOING UP (-1), OR DOWN (+1)?
3004             JR      NC,Y2Y1
1D               DEC     E
1D               DEC     E
ED44             NEG
4F       Y2Y1    LD      C,A
         ;C HOLDS NO. OF STEPS IN Y
         ;CHECK THAT LINE ISN'T A POINT
B0               OR      B
C8               RET     Z
79               LD      A,C
B8               CP      B
E5               PUSH    HL
         ;STORE THE DIRECTION OF A DIAGONAL STEP
62               LD      H,D
6B               LD      L,E
220000           LD      (DIASTP),HL
         ;DECIDE BETWEEN VERTICAL AND
         ;HORIZONTAL STEPS DEPENDING ON
         ;WHICH IS THE BIGGER OF B AND C
2E00             LD      L,0
3804             JR      C,BBC
65               LD      H,L
```

```
6B                LD      L,E
48                LD      C,B
47                LD      B,A
          ;STORE THE V/H STEP
          ;
22Ø2ØØ    BBC     LD      (VHSTP),HL
          ;
          ;B IS NOW )=C.  THE ROUTINE TAKES B-C
          ;STRAIGHT STEPS AND C DIAGONAL ONES
6Ø                LD      H,B
78                LD      A,B
CB3F              SRL     A
6F                LD      L,A
7D        NXTSTP  LD      A,L
81                ADD     A,C
          ;DECIDE ON A DIAGONAL OR A STRAIGHT STEP
          ;THIS TIME
38Ø3              JR      C,DIAG
B8                CP      B
38Ø8              JR      C,VERHOR
9Ø        DIAG    SUB     B
6F                LD      L,A
ED5BØØØØ          LD      DE,(DIASTP)
18Ø5              JR      STEP
6F        VERHOR  LD      L,A
ED5BØ2ØØ          LD      DE,(VHSTP)
E3        STEP    EX      (SP),HL
          ;MAKE THE STEP ALONG X
7C                LD      A,H
82                ADD     A,D
67                LD      H,A
          ;MAKE THE STEP ALONG Y
7D                LD      A,L
83                ADD     A,E
6F                LD      L,A
          ;THE ACTUAL PLOT!!!!!!
C5                PUSH    BC
CDØØØØ            CALL    PLOT
C1                POP     BC
          ;RETRIEVE COUNTER
E3                EX      (SP),HL
25                DEC     H
```



```
2ØDC              JR      NZ,NXTSTP
E1                POP     HL
C9                RET
```

In the following machine language demonstration program, I have combined the use of CLS (see Chapter one) with PLOT and DRAW to produce a 24-line interference pattern, which I am reliably informed looks like a heavy paperweight on a soft cushion, from above.

If you are calling the routine with USR under direct command from BASIC, then it would be a good idea to follow the USR function with a PAUSE Ø statement, so that the bottom two lines of the pattern are not destroyed on return. The comments in the listing should provide adequate explanation of the program's operation.

```
          ;DEMO ROUTINE FOR CLS, PLOT AND DRAW
          ;
3EØE      MOIRE   LD      A,ØEH
          ;BLUE PAPER, YELLOW INK
CDØØØØ            CALL    CLS
          ;OVER 1
3EØ2              LD      A,2
32ØØØØ            LD      (PFLAG),A
          ;BORDER 6
3EØ6              LD      A,6
D3FE              OUT     (ØFEH),A
          ;SET ATTRIBUTES & MASK
21ØEØØ            LD      HL,ØØØEH
22ØØØØ            LD      (ATT),HL
          ;DRAW TOP BORDER (BC=Ø)
21ØØFF            LD      HL,ØFFØØH
CDØØØØ            CALL    DRAW
          ;DRAW LEFT BORDER
2C                INC     L
Ø1BFØØ            LD      BC,ØØBFH
CDØØØØ            CALL    DRAW
24                INC     H
          ;NOW CREATE PATTERN IN THE REMAINING
          ;(255*191) PIXELS
E5        NXTDR1  PUSH    HL
          ;DRAW EROM LEFT SIDE TO CENTRE
Ø16Ø8Ø            LD      BC,8Ø6ØH
CDØØØØ            CALL    DRAW
```

```
                ;NOW FROM CENTRE TO RIGHT
C1                      POP     BC
C5                      PUSH    BC
Ø6FF                    LD      B,ØFFH
CDØØØØ                  CALL    DRAW
                ;DECREASE COUNTER TO NEXT ROW
                ;UP THE SCREEN
E1                      POP     HL
2D                      DEC     L
2ØEE                    JR      NZ,NXTDR1
2C                      INC     L
E5              NXTDR2  PUSH    HL
                ;DRAW FROM TOP EDGE TO CENTRE
Ø16Ø8Ø                  LD      BC,8Ø6ØH
CDØØØØ                  CALL    DRAW
                ;NOW FROM THE CENTRE TO THE
                ;BOTTOM EDGE
C1                      POP     BC
C5                      PUSH    BC
ØEBF                    LD      C,ØBFH
CDØØØØ                  CALL    DRAW
                ;INCREMENT COUNTER TO THE NEXT PIXEL
                ;COLUMN (RIGHTWARDS)
E1                      POP     HL
24                      INC     H
2ØEE                    JR      NZ,NXTDR2
C9                      RET
```

If speed is of the essence then I should warn you that the DRAW routine should only be used if 'general' lines from non-specific points are required. It is almost always quicker to use a customized routine, perhaps employing a look-up table of plotting coordinates, if specific lines are being drawn.

For example, if you frequently needed to draw a line right across row Ø of the screen, then it is far quicker to load the first 32 bytes of the display file with FFH than it is to DRAW from (Ø, Ø) to (255, Ø).

CHAPTER 4
Producing Animated Loading Screens

It cannot have escaped your notice that almost every self-respecting games program for the Spectrum presents you with a pretty picture to look at, to help you pass the time while the machine code makes its slow and ponderous way along the black lead between your tape player and your computer. The aesthetic appeal of this 'loading screen' is usually proportional to the amount of hard graft and graph paper that went into designing and coding the picture, both of which can be considerable, even with the use of a good 'screen graphics designer' utility program.

In this chapter I will provide you with a pair of short utilty routines to produce a spectacularly eye-catching but easily implemented alternative style of loading screen.

Ask yourself the question 'What does the ULA do while the Z-8Ø is busy loading a program from tape to RAM?' The answer is the same as always; it copes with all input, output and generation of the screen display. The last function includes FLASHing the INK and PAPER colours of any cell whose attribute has bit 7 set. We can use this property to produce a screen which flashes between two images, perhaps showing a figure in two different positions thus 'animating' it whilst loading, or as an alternative, showing two separate words of the title of the game.

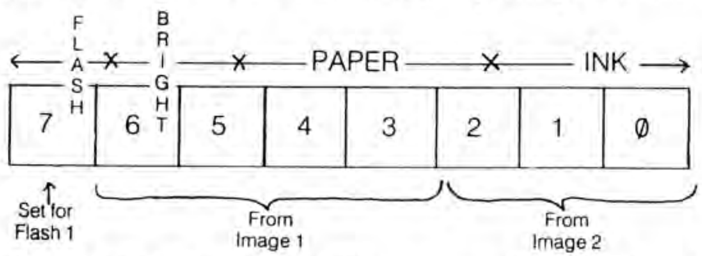
The concepts involved with this technique are very simple. We will take a blank screen, and then print different coloured spaces on it, controlling the colour of these spaces with PAPER commands. The resulting coloured cells will form our first image, and hence we have a 32 column

× 24 line grid to design the image in, each cell being one of eight colours. When the first image is complete, we will use a short machine language routine to copy the attributes into 768 bytes of reserved memory for later use.

We then use another BASIC sequence to print a second image of coloured, blank cells on the screen, and we 'pair off' the attributes of the second image with those of the first one. The PAPER attributes of the second image are shifted three bit to the right (into the INK position) and are then blended with the PAPER and BRIGHT bits of the first image.

The FLASH bit is then set, and the completed byte is replaced in the attribute file, whereupon the cell concerned starts FLASHing between the two colours provided by the images (these may, of course, be the same).

A composition diagram of the new attribute byte may be of assistance:



This technique has the advantage over conventional loading screens that only the attribute file of 768 bytes is required, as against the 6.75K (6912 bytes) of memory occupied by the standard loading screen. Thus the screen may be loaded in one ninth of the time normally required, or about five seconds, before moving swiftly on to load the game itself.

An 'animated' loading screen was first used commercially in *Bug Byte's* highly successful *Manic Miner*, now published by *Software Projects*. The two images used were colourful manifestations of the words 'Manic' and 'Miner'.

Well that's about all the theory, so how about some machine language? First we need a mega-simple block shift routine to copy the attribute file into 'safe' memory, which I have reserved as the 768 bytes from the label IMAGE1.

```
              ;MOVE IMAGE 1 FROM ATTRIBUTE FILE TO
              ;STORAGE AREA
              ;
              ;PRESERVED: A
              ;EXIT:HL=5BØØH,BC=Ø
```



```
              ;
21ØØ58        ATTSTR  LD      HL,58ØØH
11ØCØØ                LD      DE,IMAGE1
Ø1ØØ03                LD      BC,3ØØH
EDBØ                  LDIR
C9                    RET
              IMAGE1  DEFS    768
```

The routine to 'blend' the stored image with that in the attribute file is almost as simple. BLEND places the new image back in the attribute file, and that's abut all the explanation this listing needs!

```
              ;MIX IMAGE 1 FROM STORAGE WITH CURRENT
              ;ATTRIBUTES
              ;
              ;EXIT:  DE=5BØØH, BC=Ø, A=Ø
              ;
11ØØ58        BLEND   LD      DE,58ØØH
21ØØØØ                LD      HL,IMAGE1
Ø1ØØØ3                LD      BC,3ØØH
              ;
              ;TAKE BYTE OF IMAGE 1
              ;
7E            NXTATT  LD      A,(HL)
              ;
              ;MASK OFF ITS PAPER AND BRIGHT VALUES
              ;
E678                  AND     78H
              ;
              ;STORE IT AGAIN
              ;
77                    LD      (HL),A
              ;
              ;TAKE CURRENT ATTRIBUTES
              ;
1A                    LD      A,(DE)
              ;
              ;SHIFT THE PAPER BITS INTO THE INK BITS
              ;AND MASK THEM
              ;
ØF                    RRCA
ØF                    RRCA
ØF                    RRCA
```

```
E6Ø7            AND     7
                ;
                ;BLEND THESE BITS WITH THE PAPER AND BRIGHT BITS
                ;OF IMAGE 1
                ;
B6              OR      (HL)
                ;
                ;SET FLASH 1
                ;
F68Ø            ØR      8ØH
                ;
                ;STORE COMPLETED BYTE IN ATTRIBUTE FILE
                ;
12              LD      (DE),A
                ;
                ;REPEAT FOR THE NEXT CELL
                ;
13              INC     DE
23              INC     HL
ØB              DEC     BC
78              LD      A,B
B1              ØR      C
2ØEB            JR      NZ,NXTATT
C9              RET
```

Once the final image has been created in the attribute file, you can either SAVE the bytes direct to tape, using:

```
SAVE (NAME) CODE 22528,768
```

or, if you have been using the bottom two lines and don't want them to be corrupted by the tape messages, then use ATTSTR again to shift the attributes to 'safe' memory unaffected by the screen, and SAVE them from there.

CHAPTER 5
Scanning the Keyboard

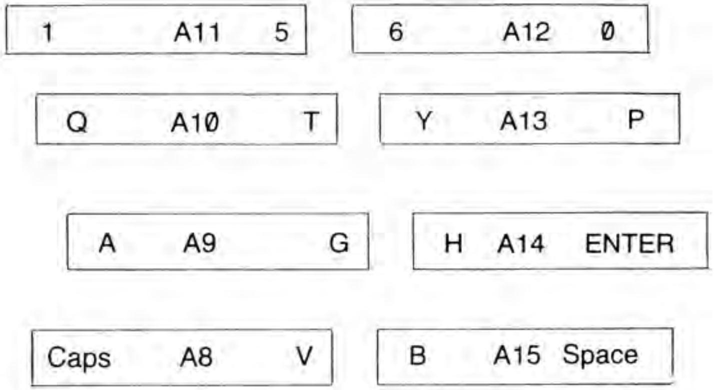
In this chapter I shall explain how the keyboard is mapped and how to read keys or groups of keys in machine language.

The more numerate among you will have noticed that the Spectrum has in fact got 4Ø keys. These appear as four rows of 1Ø, but the computer finds it easier to consider them as eight half-rows of five keys, on account of there being less than 1Ø bits in a byte.

If you have ever dared to remove the lid from your Spectrum (not an option to be recommended, as this technically invalidates your guarantee), you will have found that the keyboard is connected to the printed circuit board with two rather flimsy-looking ribbon cables. Closer examination reveals that one of these has eight tracks, and the other has five. In fact, each of the tracks on the larger cable is connected to one of bits 8 to 15 (the HI-byte) of the address bus, while the smaller cable is connected to the lowest five bits (Ø to 4) of the data bus.

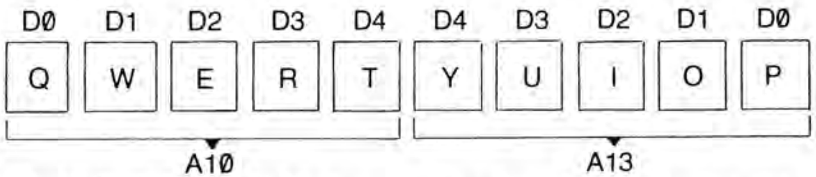
When the Spectrum makes a complete scan of the keyboard (every fiftieth of a second), the procedure it adopts is as follows. To each of the address lines in turn, it applies a 'current'. Now each of the five keys in the corresponding half-row can be considered as a switch, connected between one of the five data lines and the address line, and allowing a current to flow when depressed. The computer reads the five data lines, and if a current comes through on a line then it knows that the corresponding key is depressed, and acts accordingly.

We label the address lines (by convention) A8 to A15, and the data lines DØ to D4. The address lines are allocated to the half-rows in the following way:



Whenever we want to read a particular half-row, we send its address line low (zero). Similarly, whenever a key on that half-row is depressed, its data line goes low (zero). Otherwise it is high (one).

The data lines are attached to any half-row with the lowest bit (DØ) on the outside, counting inwards. Hence the mapping for the second row (for example) is as follows:



Well that's the theory out of the way, so now down to the actual practice. The keyboard itself is selected (rather than some other peripheral such as a microdrive or printer) by sending address line AØ low. Hence the low byte of our input port address is FEH, and we either use the instruction

```
IN       A,(ØFEH)
```

or we load the C register with FEH and use

```
IN       r,(C)
```

where r is a single register. First, however, we must load the hi-byte of the address into A or B (depending on which instruction we are using). For example, suppose we want to read the half-row A to G. This has line A9 (bit 1 of the hi-byte), so we load our register with the binary pattern 1111 11Ø1 =ØFDH

Hence a suitable fragment to read the half-row would be

```
LD       A,ØFDH
IN       A,(ØFEH)
```


For your convenience, here is a table of hi-bytes to read each half-row.

HALF-ROW	LINE	BIT	HI-BYTE	BIT-PATTERN
CAPS SHIFT — V	A8	Ø	FE	= 1 1 1 1 1 1 1 Ø
A — G	A9	1	FD	= 1 1 1 1 1 1 Ø 1
Q — T	A1Ø	2	FB	= 1 1 1 1 1 Ø 1 1
1 — 5	A11	3	F7	= 1 1 1 1 Ø 1 1 1
6 — Ø	A12	4	EF	= 1 1 1 Ø 1 1 1 1
Y — P	A13	5	DF	= 1 1 Ø 1 1 1 1 1
H — ENTER	A14	6	BF	= 1 Ø 1 1 1 1 1 1
B — SPACE	A15	7	7F	= Ø 1 1 1 1 1 1 1

We are now able to produce a program fragment to test the BREAK key (SPACE). To test it on its own, rather than with CAPS SHIFT, we use

```
LD       A,7FH
IN       A,(ØFEH)
RRA                      ;MOVE DØ INTO CARRY
JP       NC,BREAK        ;BREAK IF DØ=Ø
```

Whilst on the subject of BREAK, it may interest you to know that due to a fluke of hardware design on the Spectrum, it is possible to BREAK a Spectrum in BASIC without actually pressing the BREAK key. For some reason, pressing CAPS SHIFT with any of the following pairs of keys causes DØ to go low whenever A15 is sent low, making the Spectrum think that BREAK is being pressed.

Here are the four magic combinations:

CAPS SHIFT with Z and SYMBOL SHIFT
CAPS SHIFT with X and M
CAPS SHIFT with C and N
CAPS SHIFT with V and B

So you know what to do next time your BREAK key breaks! In fact, you will find that if you take any whole row of keys, then press any pair of them attached to the same data line, then press any other key on that row, it will appear to the Spectrum that the other key of that row on the same data line is also being pressed. That may sound a little complicated, so let me give you an example. Press T and Y together (both on data line D4 on the second row). Now press W (on D1). The computer will think that O is also being pressed, since this is also on line D1. This is of little practical use, but fascinating none-the-less.

It is possible to read more than one half-row in one go, simply by resetting more than one bit in the hi-byte of the input address. For example, to read the entire bottom row (lines A8 and A15), the value would be binary

```
Ø111 111Ø = 7EH
```

The value returned is determined as follows. If any of the keys attached to one particular data line are depressed, then the corresponding bit is zero. Otherwise it is set. Hence if we are scanning the bottom two rows of the keyboard, then D1 will be reset if any of the keys Z, S, L and SYMBOL SHIFT are depressed.

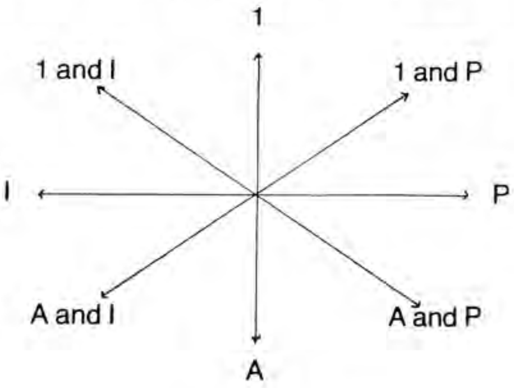
This leads us to an easy way of testing to see if any key on the entire keyboard is pressed, as might be required before the start of a new game.

```
      XOR  A             ;A=Ø,SO SCAN ALL HALF ROWS
WAIT  IN   A,(ØFEH)      ;READ BOARD
      CPL                ;MASK OFF REQUIRED BITS AND
      AND  1FH           ;CHECK FOR ALL 1'S
      JP   NZ,GO         ;JUMP IF KEY PRESSED
      JR   WAIT
```

If a key is being pressed, then the zero flag will be reset, and a jump made to start the game, or whatever you were waiting for. Otherwise the routine will jump back to WAIT with the A register holding zero again.

We now have all the information necessary to design a complete keyboard-scanning routine. As an example, I shall describe the development of a games routine, providing 8-directional control and a 'fire' bar

I shall be using the keys 1 for up, A for down, I for left, P for right, and any key on the entire bottom row for 'fire', thus:

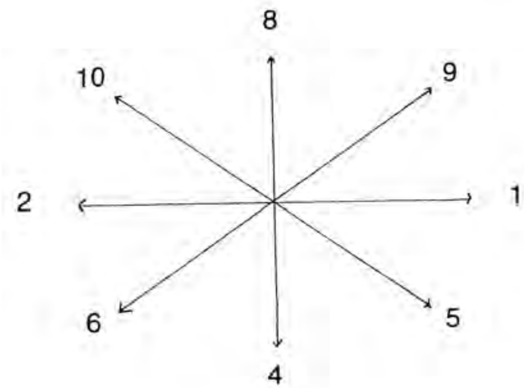


CAPS SHIFT	* FIRE*	SPACE

The routine will return a 'control code' which will depend on which of the control keys are pressed. Allocating one bit of the code to each of the four main directions up, down, left and right, and one bit to our 'fire bar', we can denote all other directions by setting combinations of these five bits. I have allocated the bits as follows:

BIT	CONTROL
Ø	RIGHT
1	LEFT
2	DOWN
3	UP
4	FIRE

Thus the values returned will be as illustrated:



+16 WHEN 'FIRE' IS PRESSED

The three spare bits in the code will, of course, be zeros. Note that since 'North-East' is a combination of upwards and to the right, the corresponding control code is 8+1=9. Similarly for the other three diagonal directions.

Now obviously there are some control values that just wouldn't make sense, for example the value 3 (= 1+2) would indicate a desire to go both right and left. The user has pressed too many keys, and instead of being 'biased', by always choosing one of the two directions in favour of the other, it would be fairer if our routine were to ignore both key presses. This it does, by calling the sub-routine CHECK twice in succession; once for left–right, and once for up–down. On entry to CHECK, the B register holds a 'mask' for the two bits we wish to examine. We are testing for the 'illegal' binary code of 11, and for this we use the fragment

```
CHECK   LD      A,C     ;C HOLDS THE CONTROL CODE
        CPL
        AND     B
        RET     NZ
```

which returns if the code is found to be legal. Otherwise we have found the illegal code 11 in C, and the routine completes its task by resetting the offending bits, with

```
LD      A,B
XOR     C
LD      C,A
RET
```

The rest of the routine listing is self-explanatory and highly demonstrative, so here it is!

```
                ;ENTRY: NONE
                ;PRESERVED: DE,HL
                ;EXIT:C=CONTROL CODE, B=12
                ;

                ;READ HALF-ROW Y-P
                ;
3EDF            SCAN1   LD      A,ØDFH
DBFE                    IN      A,(ØFEH)
                ;
                ;MASK I AND P KEYS
                ;
2F                      CPL
E6Ø5                    AND     5
                ;
                ;PUT I IN BIT 1, AND P IN THE CARRY
                ;
1F                      RRA
                ;
                ;MOVE P FROM THE CARRY TO BIT Ø
                ;
CEØØ                    ADC     A,Ø
                ;
```



```
                ;STORE THE LEFT-RIGHT CONTROL IN C
                ;
4F                      LD      C,A
                ;
                ;READ THE A KEY
                ;
3EFD                    LD      A,ØFDH
DBFE                    IN      A,(ØFEH)
1F                      RRA
                ;
                ;JUMP IF A NOT PRESSED
                ;
38Ø2                    JR      C,NDOWN
CBD1                    SET     2,C
                ;
                ;READ THE 1 KEY
                ;
3EF7            NDOWN   LD      A,ØF7H
DBFE                    IN      A,(ØFEH)
1F                      RRA
                ;
                ;JUMP IF 1 NOT PRESSED
                ;
38Ø2                    JR      C,NUP
CBD9                    SET     3,C
                ;
                ;CHECK FOR LEFT AND RIGHT BOTH
                ;BEING PRESSED
                ;
Ø6Ø3            NUP     LD      B,3
CD32ØØ                  CALL    CHECK
                ;CHECK FOR UP AND DOWN BOTH
                ;BEING PRESSED
Ø6ØC                    LD      B,12
CD32ØØ                  CALL    CHECK
                ;
                ;READ THE BOTTOM ROW
                ;
3E7E                    LD      A,Ø7EH
DBFE                    IN      A,(ØFEH)
2F                      CPL
E61F                    AND     1FH
```

```
                ;
                ;RETURN UNLESS "FIRE"
                ;
C8                      RET     Z
CBE1                    SET     4,C
C9                      RET
                ;
                ;CHECK FOR "IMPOSSIBLE" DIRECTIONS
                ;
79              CHECK   LD      A,C
2F                      CPL
AØ                      AND     B
CØ                      RET     NZ
78                      LD      A,B
A9                      XOR     C
4F                      LD      C,A
C9                      RET
```


CHAPTER 6
Player-Selectable Control Keys

In the last chapter I concluded with an example of how to develop a typical games keyboard control routine, using a predetermined choice of keys. It can often be an advantage in terms of user-friendliness, however, if the user is allowed to select her or his choice of control keys, to suit personal preference and number or shape and size of fingers, thumbs and hands. I shall provide the fundamental routines to allow you to do this in this chapter.

Picture, if you will, our typical player, poised over the keyboard and awaiting our every command, as the game finishes its long and tortuous journey from tape to memory. He is told to press any key (as is invariably the case). He is now asked to select a key to control (say) the upwards movement of his spaceship. Now first we must wait for him to stop pressing 'any key'. The following fragment will do, and is equivalent to the BASIC line.

```
1Ø  IF INKEY$<>" " THEN GO TO 1Ø

WAIT - XOR      A
       IN       A,(ØFEH)  ;READ ENTIRE KEYBOARD
       CPL                ;IF ANY KEY PRESSED
       AND      1FH       ;THEN WAIT
       JR       NZ,WAIT
```

Now we are ready to select the 'upwards' control. What is required is a routine that will return a unique value for each key pressed, and that tells us when no key or more than one key is being pressed. The key value will then be stored away for use during the game, when a separate routine will tell us if the key associated with that value is depressed.

The following routine, KFIND1, returns a key value in the D register with the zero flag set, if exactly one key is depressed. If no key is held, then D will hold FFH, while if more than one key is held, then the zero flag will be reset, indicating an error. The key values, ranging from Ø to 27H, are allocated as follows (all values in hexadecimal).

24	1C	14	C	4	3	B	13	1B	23
25	1D	15	D	5	2	A	12	1A	22
26	1E	16	E	6	1	9	11	19	21
27	1F	17	F	7	Ø	8	1Ø	18	2Ø

To the human eye that layout may seem somewhat crazy, until you realize that it will make life easier for the later game control routine. Looking at the hex values closely, we see that the lowest three bits tell us which half-row the key is in (and so which port to address) while bits 3, 4 and 5 tell us what position in that half-row the key holds. Here is KFIND1.

```
                ;ENTRY: NONE
                ;PRESERVED: L
                ;EXIT: D=KEY CODE, D=FFH IF NO KEY PRESSED.
                ;ZERO FLAG RESET IF MORE THAN ONE KEY PRESSED.
                ;OTHERWISE, ZERO SET AND A=D
                ;
                ;
112FFF  KFIND1  LD      DE,ØFF2FH
Ø1FEFE          LD      BC,ØFEFEH
                ;
                ;D STARTS AT "NO-KEY" E HOLDS INITIAL KEY VALUE
                ;FOR EACH HALF-ROW
                ;BC HOLDS PORT ADDRESS
                ;
                ;READ A HALF-ROW
                ;
ED78    NXHALF  IN      A,(C)
2F              CPL
E61F            AND     1FH
                ;
                ;JUMP IF NO KEY PRESSED
                ;
28ØC            JR      Z,NPRESS
```



```
                ;
                ;TEST FOR MULTI-KEYPRESS
                ;
14                      INC     D
                ;
                ;RETURN WITH Z RESET IF SO
                ;
CØ                      RET     NZ
                ;
                ;CALCULATE KEY VALUE
                ;
67                      LD      H,A
7B                      LD      A,E
D6Ø8            KLOOP   SUB     8
CB3C                    SRL     H
3ØFA                    JR      NC,KLOOP
                ;
                ;TEST FOR MULTI-KEY PRESS
                ;
CØ                      RET     NZ
                ;STORE KEY VALUE IN D
                ;
57                      LD      D,A
                ;
                ;TEST THE OTHER 7 HALF-ROWS
                ;
1D              NPRESS  DEC     E
CBØØ                    RLC     B
38E8                    JR      C,NXHALF
                ;
                ;SET ZERO FLAG
                ;
BF                      CP      A
C8                      RET     Z
```

A typical fragment to wait for a legal keypress in response to our 'please choose a key for upwards' prompt, would be:

```
REPT CALL   KFIND1     ;SCAN KEYBOARD
     JR     NZ,REPT    ;REPEAT IF ILLEGAL ENTRY
     INC    D          ;REPEAT IF NO KEY WAS
     JR     NZ,REPT    ;PRESSED
     DEC    D
```

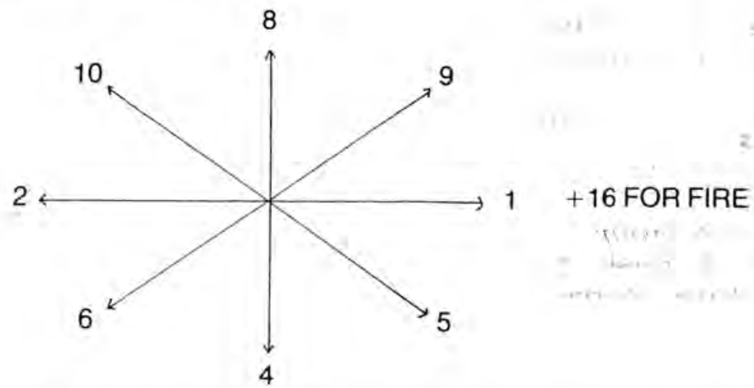
I have named the complementary routine to KFIND1, KTEST1. Every time the Spectrum needs keyboard control during a game, we must call KTEST1 once for each selected control key. The routine will read that key, and return with the carry flag reset if it is depressed, and set otherwise. The only parameter required by KTEST1 is the value of the key we are testing, entered in the accumulator. Here is the listing, followed by a worked example.

```
                ;ENTRY: A=KEY TEST VALUE
                ;PRESERVED: HL, DE
                ;EXIT: CARRY RESET IF KEY DEPRESSED, SET
                ;OTHERWISE BC=Ø
                ;
                ;
4F              KTEST1  LD      C,A
                ;
                ;LET B=16-(ADDRESS LINE NO.)
                ;
E6Ø7                    AND     7
3C                      INC     A
47                      LD      B,A
                ;
                ;LET C=(DATA LINE NO.)+1
                ;I.E. LET C=5-INT(C/8)
                ;
CB39                    SRL     C
CB39                    SRL     C
CB39                    SRL     C
3EØ5                    LD      A,5
91                      SUB     C
4F                      LD      C,A
                ;CALCULATE HI-BYTE OF PORT ADDRESS
                ;
3EFE                    LD      A,ØFEH
ØF              HIFIND  RRCA
1ØFD                    DJNZ    HIFIND
                ;
                ;READ HALF-ROW
                ;
DBFE                    IN      A,(ØFEH)
                ;
                ;PUT REQUIRED KEY BIT INTO CARRY
```



```
                ;
1F              NXKEY   RRA
ØD                      DEC     C
2ØFC                    JR      NZ,NXKEY
C9                      RET
```

As an example, let's suppose our game involves movement in eight compass directions and a fire button. For this our user will have chosen five control keys, for up, down, left, right and fire. Let the associated five key values be stored in a table pointed at by HL, in the order fire, up, down, left, right. We will use the same 'control values' as in the previous chapter, namely:



A suitable routine to build up the control value in the E register is as follows.

```
        LD      E,8         ;E IS ALSO A COUNTER
NXKEY   LD      A,(HL)      ;TAKE KEY VALUE
        INC     HL
        CALL    KTEST1      ;LOOK FOR A KEY-PRESS
        CCF                 ;1=PRESS, Ø=NOT
        RL      E           ;MOVE KEY-BIT INTO E
        JR      NC,NXTKEY   ;REPEAT FOR THE OTHER
        RET                 ;4 KEYS
```

Note that I have made the E register 'double' as a counter for the loop. The initial bit 3 is the highest set bit, and this is shifted to the left once in every pass around the loop, until after five passes it falls into the carry, causing the routine to return with the completed control value in the E register. The usual checks for 'impossible' directions such as left AND right may then be performed.

CHAPTER 7

Everything You Should Know About Interrupts

As you are probably aware, the Z-8Ø microprocessor offers us a choice of three maskable interrupt modes, named by the instructions which select them, IMØ, IM1 and IM2.

The instruction IMØ on the Spectrum is pretty much redundant. In this mode, the Z-8Ø expects an instruction from some peripheral to begin making its way along the data bus during the interrupt acknowledge cycle. In the case of the Spectrum, however, the data bus usually holds FFH during an interrupt, and this is the one-byte Hex. code for RST 38H, which the Z-8Ø duly executes. The reason I said IMØ is redundant is that IM1 performs exactly the same function of RST 38H when an interrupt occurs, whatever the data bus holds at the time.

The Spectrum normally operates in interrupt mode one, and whenever an interrupt occurs the routine at ØØ38H proceeds to increment the television frame counter and scan the keyboard, updating all the various system variables associated with it. The number of interrupts accepted since the computer was turned on is held in the three-byte system variable FRAMES, at 5C78H, 23672 decimal. The use of this counter is well documented both in the Spectrum manual and other books such as 'Super Charge Your Spectrum', also published by Melbourne House. For this reason I shall not comment further on it.

Unless you particularly wish to use the frame counter or keyboard scan while running your machine language program, you should use the instruction DI to disable the interrupts, which would otherwise slow you down. This is especially relevant when you are producing sound or are

involved in some other piece of programming that requires precision timing; otherwise you will hear a 5Ø Hz 'hum' caused by gaps in the sound while interrupts are being processed.

The final maskable interrupt mode, IM2, is the most complex and powerful. When an interrupt occurs, the Z-8Ø takes the byte currently on the data bus as the low order of an address, and the contents of the I register, or 'interrupt vector register' as the hi-byte.

This address points to a second address stored (lo-byte first) in memory, which is then loaded into the program counter. Execution of the subroutine at that address then commences. As an example, suppose the I register held FEH, the data bus had 4ØH, and the address stored at FE4ØH was ØØ38H. Then the Z-8Ø would construct the address FE4ØH from the interrupt vector register and the data bus. It would take the address stored at FE4ØH and jump to ØØ38H, which just happens to be the normal interrupt routine.

In actual fact, since the data bus usually holds FFH during an interrupt, all our 'vector addresses' will end in FFH. A little experimentation will show you that to avoid picture 'break-up' or 'snow' on the Spectrum display, the I register must either form the hi-byte of an address in ROM or in the top 32K of RAM.

This restricts us to the ranges ØØ to 3FH and 8ØH to FFH for I. Now if you have a 48K machine, it should not be too difficult to find an unoccupied vector address amongst the 127 possible options in the top 32K of RAM (note that we cannot practically use I=FFH, since the address stored from FFFFH would have its hi-byte in location Ø, which is in ROM). If, however you only have a 16K machine, or there is no room in the top 32K of RAM, then we must resort to vector addresses in the ROM.

Of the 63 such vector addresses (again 3FH is not really usable, since the hi-byte of the address would be the first byte of screen RAM), only thirteen point at addresses in the bottom 16K of RAM. Four of these thirteen are in the screen memory, leaving a choice of the following nine addresses.

	I=(Hex)	Vector Address (Hex)	Holding (Hex)	Address (Decimal)
(i)	2B	2BFF	5C65	23653
(ii)	29	29FF	5C76	2367Ø
(iii)	2E	2EFF	5CA1	23713
(iv)	19	19FF	5D22	23842
	14	14FF	6469	257Ø5
	1E	1EFF	67CD	26573
	ØF	ØFFF	6D18	27928
	Ø6	Ø6FF	71DD	29149
	28	28FF	7E5C	32348

Notes:

(i) The three bytes necessary for a jump instruction to be stored at 5C65H are normally occupied by the system variables STKEND and BREG, and as such should not be altered if you plan on using the calculator stack or returning to BASIC after your machine language program.

(ii) If your program is a hybrid BASIC/machine language one, then note that the first two bytes from address 5C76H hold the variable SEED for the BASIC pseudo-random number generator, and will be altered by RANDOMIZE or the use of the RND function. The third byte after 5C76H is the low order of FRAMES, so you must ensure that no IM1 interrupts are allowed to occur once the interrupt intercept from 5C76H onwards has been set up. Otherwise the address in any jump instruction inserted at 5C76H would increase by 256 every 2Ø ms!

(iii) The fourth five-byte register in the calculator's memory area starts at 5CA1H, so again, do not use this address if you are planning to use any of the calculator routines in the ROM from your machine language program.

(iv) If you wish to return to and use BASIC, then address 5D22H is out, too. It is simply too low: try the command CLEAR 23841 and you'll see what I mean!

The other type of interrupt not implemented on the ZX Spectrum is the NMI, or Non-Maskable Interrupt. If a Z-8Ø receives an NMI, then it completes the instruction it is dealing with and calls the routine at ØØ66H.

On the Spectrum this routine (in ROM) is the source of rather a sore point among manufacturers of hardware add-ons. The routine is as follows:

```
ØØ66H  PUSH   AF
       PUSH   HL
       LD     HL,(5CBØH)
       LD     A,H
       OR     L
     * JR     NZ,ØØ7ØH
       JP     (HL)
ØØ7ØH  POP    HL
       POP    AF
       RETN
```


The instruction labelled * should have been

```
       JR     Z,ØØ7ØH
```

. . . and would then have caused a jump to the address held in 5CBØH (which, incidentally, is quoted in the Spectrum manual as 'unused') unless the address was zero, in which case a return would have been made. Instead, as it stands, the only possible use of an NMI on the Spectrum is to cause a complete system reset if the address at 5CBØH is zero, which it usually is.

Inside the Z-8Ø are two special bits called interrupt flip-flops, and named IFF1 and IFF2. They are normally handled together under the collective name IFF, except during an NMI, when IFF2 stores the previous value of IFF1, while IFF1 is reset for the duration of the NMI. The function of IFF is to tell the Z-8Ø whether maskable interrupts are currently permitted. If they are set (value 1), then interrupts are authorized. If they are reset (masked) then the maskable interrupts will not be detected. So obviously EI sets them while DI resets them. To be absolutely accurate, the flip-flops are always reset while DI or EI is being processed, and the interrupts are not enabled until the instruction AFTER the EI has been executed. The reason for this is worth mentioning.

Whenever an interrupt is accepted, the IFF are reset automatically. It is, however, the programmer's responsibility to re-enable the interrupts before returning from the interrupt routine with RETI. It could cause untold problems if an interrupt were to occur between enabling them and returning from the last one, and hence the 'delayed action' of EI to allow a safe return to be made, as in

```
       EI
       RETI
```

. . . the standard end to an interrupt routine.

An instruction often overlooked in books on Spectrum machine language is

```
       LD     A,R
```

This may not at first sight appear to serve any useful purpose, but an examination of its effect on the flags will prove otherwise. When the instruction is executed, the parity/overflow (P/V) flag is set to the contents of IFF2. Hence we can use the instruction to tell us whether or not the maskable interrupts have been enabled. When the P/V flag is set it normally indicates even parity (PE), while when it is reset we have odd parity (PO).

Suppose that we want to preserve the contents of the IFF while we disable the interrupts to produce some 'pure' sound, and then restore the IFF. A suitable method would be:

```
        LD      A,R             ;SET P/V TO 1FF
        PUSH    AF              ;STORE P/V
        DI                      ;DISABLE INTERRUPTS
   (PRODUCE SOUND)
        POP     AF              ;RETRIEVE P/V
        JP      PO,NOT-ON       ;IF PE THEN P/V=1, SO
        EI                      ;SET 1FF
NOT-ON
```

This way, if we enter the routine with the interrupts masked, then they will not be enabled at the end of it. The instruction LD A, I affects the P/V flag in the same way as LD A, R.

I stated earlier that the data bus 'usually' holds FFH during an interrupt. For an isolated Spectrum I have never known this not to be the case. There are, however, certain hardware 'add-ons' that do not decode signals on the IOREQ and READ lines of the Z-8Ø correctly, and as a result cause variable numbers to be on the data bus during the interrupt acknowledge cycle. These add-ons do not, incidentally, include the ZX printer or the ZX Interface 1.

Now obviously if the value on the data bus changes then we will have to set up a whole table of interrupt vectors in memory for IM2, so that any of the possible values on the bus will still cause a jump to the correct address.

If we know that the value will be even, then we simply require a table of 128 vector addresses ending in ØØ,Ø2, . . ., FEH, each entry containing the address of our interrupt routine. Similarly, if the data bus will hold an odd value, then we have a table one byte higher in memory, so that the vector addresses end in Ø1, Ø3, . . . FFH.

If, however, the data bus holds any of the 256 possible values, as, for example, is the case when a Kempston Microelectronics joystick is attached to the user port, then we have to use a slightly different technique. Each of the 257 bytes in the vector table must hold the same value, so that whether the vector address is odd or even, the interrupt address will still be the same. Thus the high and low order address bytes of the interrupt routine must be the same. In case this is not clear, let us suppose for contradiction that the interrupt address is 89ABH. If we build up a table by inserting this address 128 times from (say) FEØØH, then an even value on the data bus would cause a correct jump, but an odd value would cause a jump to AB89H; obviously not what we want!

Note that the table is 257 bytes long, not 256, since we must account for the vector address ending in FFH, causing an entry to 'spill over' into the next page of memory.

Probably the most convenient value to set the I register to is FEH, using the highest possible page of RAM for the vector table. If we then fill the table with FDH, an interrupt will cause a jump to FDFDH, which is just three bytes before the start of the table. Now three bytes, as it happens, are just enough to place a jump instruction to our 'real' interrupt routine. This way, we have confined the memory needed for a fool-proof IM2 interrupt to a continuous block of 25Ø bytes, without affecting the versatility of the interrupt in any way (except to add on the 1ØT— states of the JP instruction to the processing time!).

Here is a suitable routine to initialise the IM2 system described above.

```
INT  LD      HL,ØFEØØH      ;LOAD TABLE AT ØFEØØH
     LD      BC,ØØFDH       ;WITH 256 OF FDH
LP1  LD      (HL),C
     INC     HL
     DJNZ    LP1
     LD      (HL),C         ;THE 257TH ENTRY
     LD      A,ØFEH         ;LET I=FEH
     LD      I,A
     IM      2              ;SELECT IM2
     RET

     ORG     ØFDFDH
     JP      ØØ38H          ;INSERT YOUR OWN ADDRESS
```

The above technique is all very well if you have 48K of RAM, but will not work on a 16K machine. As I mentioned before, pointing the interrupt vector register at any page of the lower 16K of RAM will cause 'snow' on the screen. All is not lost, however, for in the case of at least one 'rogue' peripheral, the Kempston joystick, there is a way, although somewhat messy and restrictive, of using interrupt mode 2.

The joystick is normally 'read' in BASIC by a command of the form

```
LET A= IN 31
```

. . . but in fact all that is required for the interface to deposit a value on the data bus is to send address line A5 low (holding zero), so that the command

```
LET A= IN (31+64+128+256+512+1Ø24)
```

for example, would do just as well. When A5 is high, however, the joystick will not affect the contents of the data bus, and the normal value of FFH should result during interrupt acknowledge.

Well that's the theory done. Now how do we ensure that A5 is high whenever an interrupt occurs under IM2? This can be done by ensuring that the program counter holds an address containing bit 5 set before an interrupt, and this is why I called the technique 'somewhat messy and restrictive.'

We principally have two options once the program counter is in a 32-byte block that has bit 5 set for its addresses; we can either come to a HALT instruction while we wait for an interrupt, or we can spend the time doing something useful like generating sound. If the latter option is chosen, there are two main points to remember.

Firstly, we must not allow bit 5 of PC to go low, so the routine must either exist in a 32-byte block, or call other subroutines that are also in locations where A5 is high. Secondly (and assuming that we do not wish to waste time in this routine once an interrupt has been taken), we must continually test some kind of flag that is set by the interrupt routine, so that we know when an interrupt has been taken.

After the interrupt has been dealt with we have up to about 2Øms, which is an awfully long time in machine language, to do as much 'normal' processing as we want before getting back to wait for the next interrupt.

At first sight all the effort required to use IM2 interrupts on the Spectrum may not seem worthwhile, but they do in fact have a wide range of uses. They are the fundamental concept behind many of the commercially available utilities such as real-time clocks, TRACE routines, extensions to BASIC, user-definable function keys and so on.

In addition to this, interrupts have the special property that they are generated at precisely the same frequency as the frames which go to make up your TV display. They always occur when the beam is at the high-point of its 'fly-back' from the bottom to the top line of the display; and consequently we can use interrupts to produce full-screen (BORDER included) horizons, flickeress pixel-by-pixel animation of sprites, and higher resolution colour, to name but a few of the possibilities afforded by TV-Synchronized processing.

CHAPTER 8
A Discussion of Pixel-Animation Techniques

Since the launch of the ZX Spectrum, the quality of games software for it has steadily increased, and with it the technical quality of animation. The emphasis has shifted away from movement by one character at a time, and towards movement by a few pixels at a time. At the same time, the spectrum has been pushed closer and closer to its design limits, with programmers squeezing every last gramme of speed out of the Z-8Ø microprocessor in an effort to achieve more spectacular special effects than the last game.

In the next few chapters I shall be developing a very powerful set of routines that will let you achieve totally flickerless animation and special effects never before seen on the Spectrum.

Before we go any further, let us remind ourselves how the television display is generated. Although when we watch TV, we see a continuous picture, it is in fact only one (in the case of black and white) or three (in the case of most colour sets) electron beams scanning across and down the screen at great speed. If it were not for the human phenomenon of persistence of vision, which 'preserves' on the retina of the eye the image generated by the beam long enough for it to complete one 'frame' of the TV (2Ø milliseconds), then all we would see is a brightly coloured dot moving at high speed, and television displays as we know them would not exist.

In the U.K., televisions are quoted as having a '625-line' display. That is to say that the television pictures are transmitted as signals for 625 scan-lines of the TV. An average TV set only displays about 540 of these scan-lines:— the rest are off the top and bottom of your screen, and some of the resulting 'spare' time is consumed by a period known as 'flyback', when the electron beams are being moved back from the bottom of the screen to the top, ready to produce the next frame.

Some of the spare scan-lines, incidentally, are used to transmit the data for the BBC's 'Ceefax' and ITV's 'Oracle' teletext services. A decoder in your teletext TV then converts the binary data into a full-screen TV picture and displays it. It is the number of scan-lines available for this feature that limits the resolution and choice of colours on teletext graphics:— there just isn't enough room to transmit high-resolution teletext at an acceptable baud-rate.

Well before I digress any further, back to our discussion of picture-generation. The chip that is responsible for TV handling in the Spectrum is called the ULA (Uncommitted Logic Array) and what it does is to use two scan-lines for each row of the Spectrum display. Hence the text area occupies 2×192=384 scan lines; about 70% of the screen height and takes about 61% of each frame-time, or 12.288 ms, to generate.

Now why, I hear you ask, am I going into so much detail about the TV display? Well, in the course of 'normal' animation by one cell at a time, none of this would be necessary. The characters are moved fairly 'rarely', typically about five times a second, or once in every ten frames or so. Consequently no significant interference by the TV display generation is noticed.

However, every time we move a character, we must in some way 'blank out' its old image and 'print in' the new one in the display file. If the television happens to be producing the scan lines on which we are printing and deleting, then it will take the image from memory, whatever state it is in, and display it on the screen. The consequence is that for the current frame an incomplete image will be displayed.

As I said, this interference is not noticeable for low-frequency movement. However, animation by pixels requires up to eight times the frequency of movement to move a character at the same speed as by cells, and this results in unacceptable ghosting and flickering using standard techniques.

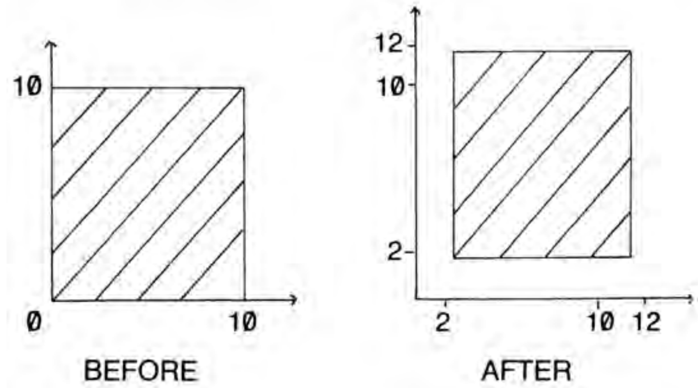
One partial solution to the problem is to deal with the movement in terms of TV scan-lines. You take each row that would be occupied by either the 'new' or the 'old' image in turn. Then blank out any part of the 'old' image

on that row, and print in any part of the new image on that same row. This produces reasonably smooth animation, since we will never have completely blank cells on the screen where there shouldn't be. A similar technique to this has been used by Ashby Computers and Graphics Limited in their highly successful 'Ultimate, Play The Game' series for the Spectrum.

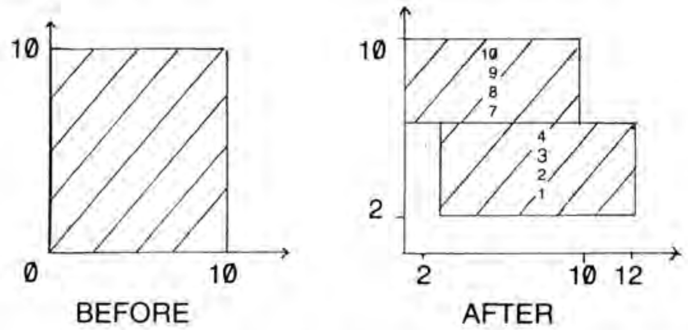
There is, however, a drawback to the technique described above. All interference has not been eliminated, and we are left with two principle effects. Firstly, there is a reduced form of flickering, where a blank row is displayed while we are switching between deleting the old image and printing the new one on that row. The result is that the character is continually 'dissected' in different places as it moves. This effect can be minimised by moving our character from its bottom row upwards, rather than in the traditional top-downwards manner. This way we ensure that the printing routine can only 'clash' with the TV once during each frame, when the two processes 'cross over', working in opposite directions. I still find the remaining interference frequent enough to be annoying, however.

The second effect of interference between the display generation and printing routine is that the character becomes either 'stretched' or 'shrunken' vertically, or disjointed horizontally, depending on the direction of movement.

Imagine that the character is being moved (from the bottom row upwards) in the North-East direction, that is upwards and to the right. For the sake of argument we will assume it to be a 10×10 pixel square moving two pixels at a time along each axis. In the usual case, when there is no interference, the images observed would be as shown.



However, if the TV scan passes over the area in which we are printing whilst we are doing so, then we will end up with a disjointed figure, now shrunken to an overall eight pixels in length. If the 'clash' occurs after the fourth row up has been moved, for example, then we will have the following images:



As you see, we would have lost rows 5 and 6 of our mutilated character for one frame.

The only sure-fire way to obtain totally flickerless animation is to ensure that the TV scan NEVER passes over the area which is currently being printed in. There are a variety of different ways in which to do this, and they all involve keeping track of the interrupts that the Spectrum receives every fiftieth of a second. This is the same frequency as your TV display, and consequently the electron beam is always in the same place, during flyback, when an interrupt signal is sent.

As long as we can confine our printing to times when the TV is not generating the 384 scan-lines of the text area, we can be sure that there will be no flickering during animation, no matter where on the screen our shape is being printed. Hence the 'safe' times are while the bottom and top borders are being generated, and during flyback.

Now unfortunately, unless all our games routines are 'time-constant', that is they always take the same time to execute, we will have no way of knowing when the text area has been just generated, and are thus unable to use the 'bottom border' time for printing. This leaves us with the time between an interrupt and the TV scan reaching the text area, which is about 14,2ØØ T-states, or 4.Ø6 ms. Now incredible as it may seem, this is, in fact, enough time to print forty characters on the screen, using a special 'print processing' interrupt handler, which I shall be developing in the next chapter.

CHAPTER 9
An Interrupt-Driven Print-Processor with Full-Screen Horizon Generator

I shall now begin development of the interrupt-driven 'print processor' routine mentioned at the end of the last chapter. This, together with a comprehensive suite of routines in the following chapters, will enable you to produce the much-sought-after flickerless pixel animation of any 'sprite' (a shape consisting of a block of characters) up to 5×5 or 7×4 cells in area.

In addition to its main function as a print-processor, the interrupt handler, together with a complementary set of routines, will also be capable of generating a full-screen (border included) horizon. At the time of writing, the first and only game to have a full-screen horizon was Quicksilva's 'Aquaplane', by John Hollis. Unlike the stationary horizon of 'Aquaplane', the horizon generated by my routines will be movable at between one and eight 'pixels' per frame within a region bounded by the top of the text area and the very bottom of the screen.

This is only made possible by a special technique that 'fools' the computer into producing three or four colours in the attributes covering the horizon, so that we have something left to print with after two colours have been used up by the background.

I should state now that these routines have been designed to run in the top 32K of RAM on a 48K machine, based on the assumption that if you are a serious machine language programmer, working with an assembler, then you probably have 48K of RAM in order to have room for anything but a small text file once the assembler has loaded.

At the risk of repetition, let me explain that machine code placed in the lowest 16K of RAM will run about 2Ø% slower whenever the TV is generating the text area, as the ULA and Z-8Ø will both be trying to access the same eight memory chips, and the ULA takes priority. As a consequence, the 'print-processor' part of the interrupt handler would not need modification to run in the lower 16K (it only runs while the top border is being generated), but the horizon generator would require extensive modification in order to compensate for the loss of speed and general changes in timing.

In this case it would probably be better for you to settle for a stationary horizon on a boundary between two of the twenty-four lines. That way, no special work is required on the attributes, and the border horizon is generated simply be going through a suitable delay loop and then changing the colour from that above the horizon to that below it, using an OUT instruction.

The print-processor system works in the following manner. Every time some animation routine feels the urge to print something on the screen, instead of doing so directly, it does as much preparation as possible and then deposits the resulting data for each character to be printed in an area of memory which shall be called the 'print buffer'. Then, on every interrupt, the print-processor 'empties' the buffer, one entry at a time, and puts the corresponding character and its attributes in the correct place in the display and attribute files respectively.

We shall label the start of the buffer as BUFFER. Each entry in the buffer is six bytes long; and the data is formatted as follows.

1) ATTRIBUTE BYTE
2) ATTRIBUTE ADDRESS (LO)
3) ATTRIBUTE ADDRESS (HI)
4) DISPLAY FILE ADDRESS (HI)
5) CHARACTER DATA ADDRESS (LO)
6) CHARACTER DATA ADDRESS (HI)

Note that we do not need to store the low order of the display file address, since it is identical to that of the attribute file (byte 2).

I have never known it to be useful or necessary to use FLASH 1 when animating sprites by pixels, and thus decided to sacrifice its attribute

bit, leaving us with room for a flag. The attribute is stored in a form shifted one bit to the left in the buffer, leaving bit Ø as a flag. It is often useful when two sprites overlap to be able to 'merge' one on top of the other using an OR operation, rather than the usual 'blotting out' of the first image by the second. I have named these two types of printing operation 'OR-print' and 'OVER-print' respectively. When bit Ø of the attribute byte is set, it will tell the print-processor to merge this particular character with the current contents of the cell, by OR-printing.

The routine may be easily modified if you wish to use it for your own purposes to print using the XOR (exclusive OR) operation, simply by changing all the relevant OR instructions to XOR ones. Setting the flag would then, of course, indicate 'XOR-printing required.'

In order to generate a stable horizon, it is imperative that any routine executed between the interrupt and the horizon generation is time-constant. Thus I have carefully balanced the print-processor routine so that whatever is in the buffer, it still takes the same time to execute.

Various tricks have been used to make the routine as fast as is practically possible. It emerged that there was time to print exactly 4Ø characters, and hence our buffer needs to be 4Ø×6=24Ø bytes long. If we ensure that the low order of BUFFER is 1Ø Hex, then we can use single-register increment instructions such as

```
                INC     L
```

as opposed to

```
                INC     HL
```

to step through the buffer. This saves two T-states every time we use it, and has the added advantage that we can tell after processing an entry whether the end of the buffer has been reached, simply by using

```
INC     L
```

and then testing the zero flag.

If you are not using the top few lines of the text area for animation, or if you don't mind flickering in that area, then you may at some stage in your life find it desirable to increase the number of characters that the print-processor can handle. Up to a limit of 42 characters (the hitch-hikers answer to everything) this poses no problems, as the buffer would still be contained within one 'page' (256 bytes with the same high order address) of memory. However, beyond this limit, some alteration will be required so that the routine steps across the page-boundary correctly.

Since each buffer entry is six bytes long, there are six INC L instructions in each loop of the routine, as you will see when I eventually produce the listing. The first one is after the attribute byte has been fetched, the second after its low-order address has been fetched, and so on. A quick bit of maths shows us that a buffer ending at FCFFH and 43 entries long, would start at FBFEH. Hence for buffers over 42 entries, change the second INC L to INC HL.

Similarly, for an 86-entry buffer the start address would be FAFCH, so for buffers longer than 85 entries, change the fourth INC L to INC HL. The maximum buffer length by this method is a more-than adequate 128 entries (768 bytes) at which point it would start on a page boundary, FAØØH. As a guide to the extra processing time for a longer buffer, each entry takes about 1.6 rows or 3.2 scan-lines to be printed.

Now obviously there will be times when we do not actually use all forty entries in the buffer. However, we must ensure that the print-processor still takes the same time to execute for a null entry, and probably the easiest way to do this is to make the routine THINK it is printing a character, without actually affecting the screen.

To signify a null entry in the buffer we will set the attribute byte to zero. The following fragment will be executed at the start of each loop, with HL pointing at the start of a buffer entry.

```
NXTCH   LD      A,(HL)
        AND     A
        JR      Z,FAKE
```

At FAKE we will update the buffer pointer in HL and set up the registers necessary to OR-print a space (no net effect) in the bottom right-hand corner of the screen. There will then be a short pause in order to perfectly equalise the timing with the normal printing routine path, followed by a jump into the main section of the OR-printing procedure. The attributes will not be affected, and the fragment at FAKE goes like this:

```
FAKE    LD      A,5       ;ADJUST BUFFER POINTER
        ADD     A,L
        LD      L,A
        LD      DE,5ØFFH  ;D.F. ADDRESS OF (23,31)
        LD      A,(DE)    ;TIMING EQUALIZER
        LD      BC,3DØØH  ;ADDRESS OF 'SPACE' DATA
        EX      DE,HL     ;IN ROM
        NOP               ;WAIT 14 T-STATES
```



```
        JP      $+3       ;NOTE: $=PROGRAM COUNTER
        JR      OR        ;JUMP INTO MAIN ROUTINE
```

The main substance of the instructions from label OR is the fragment

```
        LD      A,(BC)    ;TAKE DATA
        OR      (HL)      ;'OR' WITH DISPLAY ROW
        LD      (HL),A    ;INSERT IN DISPLAY FILE
        INC     BC        ;NEXT BYTE OF DATA
        INC     H         ;NEXT ROW OF DISPLAY FILE
```

. . . which is repeated six times, followed by

```
        LD      A,(BC)    ;PRINT LAST ROW OF CHAR
        OR      (HL)
        LD      (HL),A
        EX      DE,HL
        INC     L         ;TEST FOR END OF BUFFER
        JP      NZ,NXTCH
```

On first sight the listing for this may appear clumsy, but we must remember that time is of the essence, and a conventional loop repeated seven times would take a lot longer to execute. For the same reason, an absolute JP instruction (1Ø T-states) has been employed rather than a relative jump (12 T-states, the extra time being used to add the displacement to the program counter).

While all this is fresh in your mind, and before proceeding to develop the horizon-generating part of the interrupt handler, I shall list the first part of the routine for your contemplation. A word or two of explanation for the first few lines of the routine is required. The first priority in any interrupt handler should be to preserve any registers used by the handler. Having done that, we must output the border colour for the 'sky' above the horizon. This also provides an opportunity to send a 'click' to the loudspeaker, by adding 1Ø hex. to the argument of the XOR instruction at label TOPBRD. We shall always store the last value sent OUT to port FEH in the variable BORD, so preserving the speaker status (bit 4).

The variables CHSTRE and BUFFPT store the number of 'real' entries and the address of the next free entry in the buffer respectively. These variables will be greatly utilised later on. Now for the first part of the interrupt handler; please read at least to the end of this chapter before attempting to use it, as running it on its own would cause an almighty crash. Note also that the $ sign in jump instructions means 'program counter', so

```
JR      $+2  and
JP      $+3
```

simply mean 'advance to the next instruction' and are used as timing delays.

```
          BUFFER  EQU     ØFF1ØH
1ØFF      BUFFPT  DEFW    BUFFER
ØØ        CHSTRE  DEFB    Ø
ØØ        BORD    DEFB    Ø
          ;
          ;
          ;INTERRUPT HANDLER ******
          ;SAVE REGISTERS
          ;
F5        INTERP  PUSH    AF
C5                PUSH    BC
D5                PUSH    DE
E5                PUSH    HL
          ;
          ;SET TOP BORDER
          ;
21Ø3ØØ            LD      HL,BORD
7E                LD      A,(HL)
E61Ø              AND     16
EEØ5      TOPBRD  XOR     5
D3FE              OUT     (ØFEH),A
77                LD      (HL),A
          ;
          ;START WORKING THROUGH BUFFER ENTRIES
          ;
211ØFF            LD      HL,BUFFER
          ;
          ;A ZERO ATTRIBUTE="NO ENTRY" SO PRINT A FAKE
          ;CHARACTER
          ;
7E        NXTCH   LD      A,(HL)
A7                AND     A
2871              JR      Z,FAKE
2C                INC     L
          ;
          ;TAKE ATTRIBUTE ADDRESS
```



```
          ;
5E                LD      E,(HL)
2C                INC     L
56                LD      D,(HL)
2C                INC     L
1F                RRA
          ;
          ;STORE NEW ATTRIBUTE
          ;
12                LD      (DE),A
          ;
          ;FORM D.F. ADDRESS
          ;
56                LD      D,(HL)
2C                INC     L
          ;TAKE CHARACTER DATA ADDRESS
          ;
4E                LD      C,(HL)
2C                INC     L
46                LD      B,(HL)
          ;
          ;DECIDE WHETHER TO MERGE OLD CHARACTER
          ;
3Ø2F              JR      NC,NTOR
EB                EX      DE,HL
          ;
          ;PRINT NEW CHARACTER USING "OR"
          ;
ØA        OR      LD      A,(BC)
B6                OR      (HL)
77                LD      (HL),A
Ø3                INC     BC
24                INC     H
ØA                LD      A,(BC)
B6                OR      (HL)
77                LD      (HL),A
Ø3                INC     BC
24                INC     H
ØA                LD      A,(BC)
B6                OR      (HL)
77                LD      (HL),A
Ø3                INC     BC
```

```
24                      INC     H
ØA                      LD      A,(BC)
B6                      OR      (HL)
77                      LD      (HL),A
Ø3                      INC     BC
24                      INC     H
ØA                      LD      A,(BC)
B6                      OR      (HL)
77                      LD      (HL),A
Ø3                      INC     BC
24                      INC     H
ØA                      LD      A,(BC)
B6                      OR      (HL)
77                      LD      (HL),A
Ø3                      INC     BC
24                      INC     H
ØA                      LD      A,(BC)
B6                      OR      (HL)
77                      LD      (HL),A
Ø3                      INC     BC
24                      INC     H
ØA                      LD      A,(BC)
B6                      OR      (HL)
77                      LD      (HL),A
EB                      EX      DE,HL
                ;
                ;LOOP BACK UNTIL END OF BUFFER
                ;
2C                      INC     L
C216ØØ                  JP      NZ,NXTCH
                ;
                ;JUMP FORWARD
                ;
C3ØØØØ                  JP      ROWS
                ;
                ;PRINT ON TOP OF OLD CHARACTER
                ;
ØA              NTOR    LD      A,(BC)
12                      LD      (DE),A
12                      LD      (DE),A
Ø3                      INC     BC
14                      INC     D
```



```
ØA                      LD      A,(BC)
12                      LD      (DE),A
12                      LD      (DE),A
Ø3                      INC     BC
14                      INC     D
ØA                      LD      A,(BC)
12                      LD      (DE),A
12                      LD      (DE),A
Ø3                      INC     BC
14                      INC     D
ØA                      LD      A,(BC)
12                      LD      (DE),A
12                      LD      (DE),A
Ø3                      INC     BC
14                      INC     D
ØA                      LD      A,(BC)
12                      LD      (DE),A
12                      LD      (DE),A
Ø3                      INC     BC
14                      INC     D
ØA                      LD      A,(BC)
12                      LD      (DE),A
12                      LD      (DE),A
Ø3                      INC     BC
14                      INC     D
ØA                      LD      A,(BC)
12                      LD      (DE),A
12                      LD      (DE),A
Ø3                      INC     BC
14                      INC     D
ØA                      LD      A,(BC)
12                      LD      (DE),A
C384ØØ                  JP      $+3
                ;
                ;LOOP BACK UNTIL END OF BUFFER
2C                      INC     L
C216ØØ                  JP      NZ,NXTCH
                ;
                ;THEN JUMP TO GENERATE HORIZON
                ;
C3ØØØØ                  JP      ROWS
                ;TO BALANCE THE TIMING PRINT
```

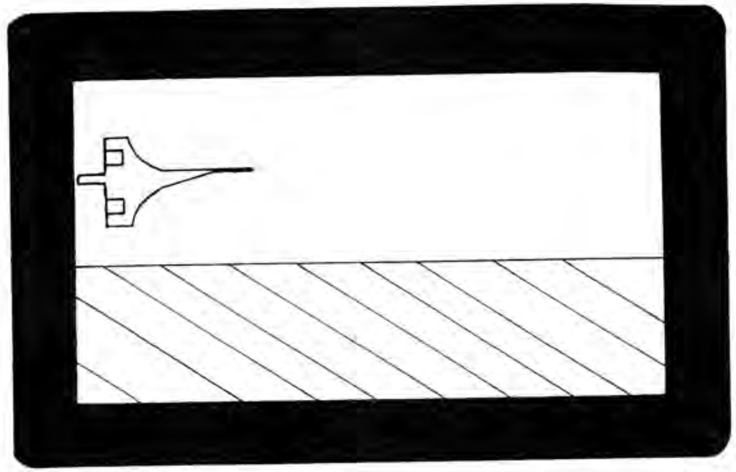
```
                ;A SPACE WITH "OR" IN THE BOTTOM-RIGHT
                ;CORNER
                ;
3EØ5            FAKE    LD      A,5
85                      ADD     A,L
6F                      LD      L,A
11FF5Ø                  LD      DE,5ØFFH
1A                      LD      A,(DE)
Ø1ØØ3D                  LD      BC,3DØØH
EB                      EX      DE,HL
ØØ                      NOP
C39BØØ                  JP      $+3
188C                    JR      OR
```

Incidentally, the label ROWS will be on the first line of the next part of the routine.

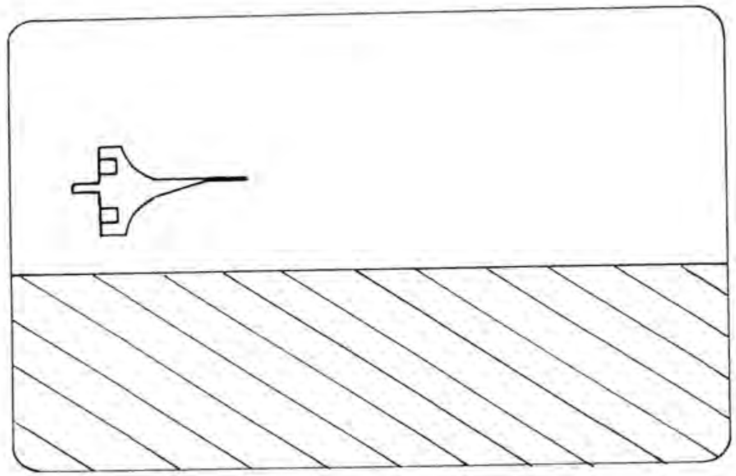
I shall now begin a discussion of the principles involved in generating a moving full-screen horizon. For the sake of convenience, I shall refer to the area of the screen above the horizon as 'sky', and that below it as 'sea'. The routines will produce a cyan sky and blue sea initially, but these colours are very easily changed and I shall indeed be including a routine to do this at a later stage.

You may well be asking yourself 'What use is a full-screen horizon' or 'is it worth all the effort and careful timing involved?'

The answer is that it IS worth the effort, because although extending the horizon doesn't allow you to print in any greater area of the screen, it does increase the effective 'playing area' of a game and is far more aesthetically pleasing. Suppose, for example, that your game involved controlling an aircraft as it flew along above the sea, and that the current position of the 'plane was as far to the left in the text area as is possible, without 'spilling' off the screen. With a conventional text-only horizon, we would see something like this:



As you see, our aircraft looks extremely 'out of place' since it is cramped right up against the left border. Compare this with the 'spacious' look of a full-screen horizon, where the 'plane does not look at all unnatural, even though it is being printed in exactly the same place on the screen:



The principle behind all programming 'tricks' with the border is very simple. The ULA continually reads port 254 and sends the corresponding colour to the TV, which is building up the display line by line. Hence to obtain a steady boundary between two border colours, we just wait an exact time after each interrupt signal before sending out the 'sea' colour to port 254.

Now the Z-8Ø in the Spectrum runs at a clock speed of 3.5 MHz, that is to say there are 3,5ØØ,ØØØ T-states per second. Television frames are generated at 5Ø Hz, and each with 625 scan lines. Hence we have

$$\text{Time taken for one scan-line} = \frac{3\,500\,000}{625 \times 50}$$

$$= 112 \text{ T-states}$$

Not forgetting that the Spectrum uses two scan-lines for each row of the display, we have that each row takes 2×112=224 T-states to generate, and this is how long we have to wait for each row of the display above the horizon, before changing the border colour. A suitable delay loop, where the number of rows is in the accumulator, would be as follows:

```
SCAN1   LD      B,15        ;7 T-STATES
LN      DJNZ    LN          ;14*13+8=19Ø
        AND     ØFFH        ;7 T-STATES
        INC     HL          ;6 T-STATES
        DEC     A           ;4 T-STATES
        JP      NZ,SCAN1    ;1Ø T-STATES, LOOP BACK
                            ;FOR NEXT ROW
```

You will find the above fragment in the second part of the interrupt handler.

Well that's the border control taken care of. Now what about the attributes? If the horizon is on a boundary between two lines of the display, then we have no problem. We simply use cyan paper in the line above the horizon and blue paper in the line below it. If, however, the number of text rows above the horizon is not divisible by eight, then we will need to produce both cyan and blue paper in one line of attributes. It is not sufficient just to use cyan INK and blue PAPER, since this would leave us with no colours to print our sprites in over the horizon.

To produce these 'two-paper' attributes, we need to fill the line containing the horizon with cyan paper (and whatever coloured ink we happen to be using) then wait for the TV-scan to reach the horizon level, then hurriedly refill the line with blue-papered attributes. The ULA tests the attributes every time it generates one row of the text area, so the result should be cyan paper above the horizon and blue paper below it, together with our choice of ink and brightness for each region.

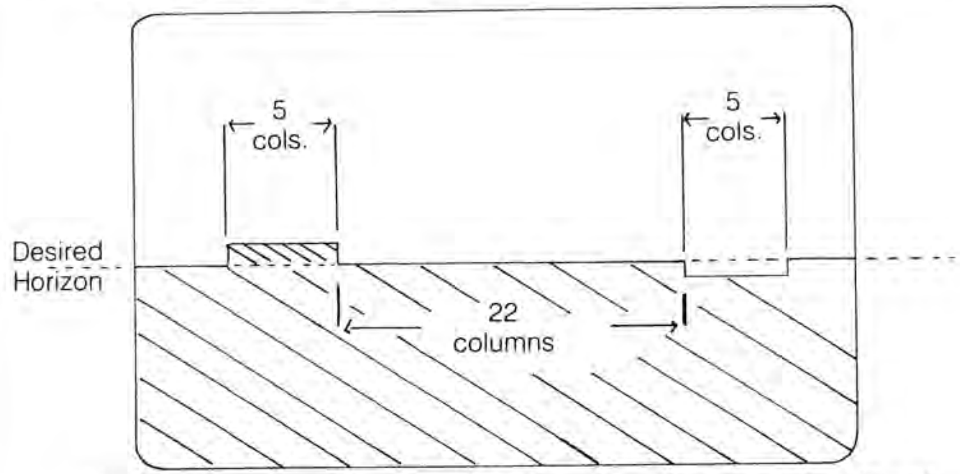
Unfortunately, due to the great speed at which the electron beam hurtles from left to right across the screen, we simply do not have enough time to replace a whole line of attributes before the ULA needs

them again. A quick calculation shows that to change 32 attributes in 224 T-states would require an average rate of one attribute in 7 T-states, which is only enough time to do a basic LD (HL), A, without incrementing any pointers.

There are two other factors to consider. On the plus side, we do in fact have slightly longer than 224 T-states. Suppose, for example, that we had managed to refill half of one TV-row, starting at the left-hand edge of the text area. The time we have had to do this would be that taken for 1.5 rows (336 T-states), since we would have started the moment the beam had left the first attribute, and finished when it 'lapped us' half way across the screen.

On the minus side, we must remember that changing the attributes will require access to the lowest 16K of RAM, and the resulting interruptions by the ULA will mean a slight overall speed decrease.

After experimentation, I found that the best we could hope for is a continuous level 'horizon' of 22 attributes. In most arcade games the action always tends towards the centre of the playing area, so I have positioned these 22 bytes in the centre of the text area, leaving 'steps' of 5 columns in width on either side. As it stands, the resulting malformed horizon would appear as follows:



Now this jagged appearance is, unless you want a rectangular 'hill' and 'valley' on your screen, quite intolerable. The best solution to the problem is to fill the five columns on the left wih one row of ink immediately above the horizon, and the five columns on the right with one row of ink just below the horizon. We will then use cyan ink on the left and blue ink on the right to create a continuous, level horizon.

At the beginning of each interrupt, all the attributes concerned will be cyan paper and (say) white ink. The sequence in which we will change the attributes is as follows (after an exact delay).

1) Fill rightmost 5 attributes with cyan paper, blue ink.
2) Fill leftmost 5 attributes with blue paper, cyan ink.
3) Fill middle 22 attributes with blue paper, white ink.
4) Fill leftmost 5 attributes with blue paper, white ink.
5) Fill rightmost 5 attributes with blue paper, white ink.
6) We must now wait 'till the TV has completely finished generating this line of the display. Part of this time will be spent preparing the print buffer for the next interrupt.
7) Fill all 32 attributes with cyan paper, white ink, ready for the next interrupt.

At a certain critical point during Stage (3), the beam will reach the right-hand side of the screen having generated the cyan row immediately above the horizon. While the beam is in 'flyback' to the left-hand side of the screen, we must take a break from filling attributes and output the new border value. We store the new value at (BOTBRD+1), and add 16 to it if a click is to be sent to the speaker. This combined with the value in (TOPBRD+1) allows us a choice between no sound, a 5Ø Hz or 1ØØ Hz sound. In the last case, you will find that when we move the horizon up and down in the next chapter, the waveform of the sound changes due to the varying time between the 'top' click and the 'bottom' click.

Going back to our procedure for changing the attributes, the only slight side effect of using this technique will be that up to 2 rows of any sprite printed on the horizon in the left or right five columns will be cyan and blue respectively. This is hardly noticeable and a small price to pay for the overall effect of a full-screen display.

The technique will work for anything down to 2 text rows above the horizon, and the number of these rows is stored in (ROWS+1). Zero text rows poses no problem, as the whole screen will be sea, and we just jump straight to label NOWAIT where the border colour is changed. However, in the very unlikely event that you require just one text row above the horizon, then you will have to revert to the old technique of filling it with ink and using cyan ink and blue paper. This is because there is not enough time to manipulate the attributes in the manner required by the new technique. In this case, the interrupt handler jumps to label WT1LN and waits for the TV to reach row one before changing the border.

Scattered through the listing you will find labels HCOL1 to HCOL4; these will be used in the next chapter to change the sky and sea colours. HRZN3 will be used by the horizon movement routines. It holds the address of the 28th horizon attribute, and will be set to point at ØØ1BH (in the ROM) whenever no attribute work is required. This is indeed the case in the routine as it stands, as I have set (ROWS+1) to 96, or half-way down the screen.

After the horizon has been generated, the interrupt handler has two more tasks. Firstly, it must 'tidy up' the print buffer by deleting all the entries just printed, inserting zero attribute bytes to signify forty null entries. It resets BUFFPT to the first free entry, which is now at BUFFER, and sets the number of entries, CHSTRE, to zero. Finally it retrieves all the registers stored at the beginning of the interrupt, and terminates.

Here then is the second part of the interrupt handler, followed by an initialization routine.

```
3E6Ø      ROWS    LD      A,96
          ;
          ;A HOLDS NO. OF ROWS ABOVE HORIZON
          ;IF A=Ø THEN DON'T WAIT TO CHANGE BORDER
          ;
D6Ø1              SUB     1
DAC8ØØ            JP      C,NOWAIT
          ;
          ;IF A=1 THEN WAIT FOR ONE SCAN LINE
          ;
CABEØØ            JP      Z,WT1LN
3D                DEC     A
          ;
          ;IF A=2 THEN SKIP THIS DELAY
CA19ØØ            JP      Z,GO4IT
          ;
          ;THIS LOOP TAKES 224 T-STATES OR ONE
          ;SCAN-LINE PER PASS
Ø6ØF      SCAN1   LD      B,15
1ØFE      LN      DJNZ    LN
E6FF              AND     ØFFH
23                INC     HL
3D                DEC     A
C2ØEØØ            JP      NZ,SCAN1
          ;
          ;TIMING BALANCER
          ;
Ø6ØA      GO4IT   LD      B,1Ø
1ØFE      SELFS   DJNZ    SELFS
```

```
E6FF                AND     ØFFH
            ;
211BØØ      HRZN3   LD      HL,1BH
            ;HL=ADDRESS OF 26TH ATTRIBUTE IN THE LINE
            ;
7D                  LD      A,L
E6EØ                AND     ØEØH
47                  LD      B,A
            ;
            ;FIND BOTTOM BORDER COLOUR
            ;
11ØØØØ              LD      DE,BORD
1A                  LD      A,(DE)
E61Ø                AND     16
EEØ1        BOTBRD  XOR     1
12                  LD      (DE),A
1E29        HCOL1   LD      E,41
ØEØD        HCOL2   LD      C,13
            ;
            ;FILL RIGHT 5 ATTS WITH CYAN PAPER,BLUE INK
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
68                  LD      L,B
1EØF        HCOL3   LD      E,15
            ;
            ;FILL LEFT 5 ATTS WITH BLUE PAPER, CYAN INK
            ;
71                  LD      (HL),C
2C                  INC     L
71                  LD      (HL),C
2C                  INC     L
71                  LD      (HL),C
2C                  INC     L
71                  LD      (HL),C
2C                  INC     L
```



```
71                  LD      (HL),C
2C                  INC     L
            ;
            ;FILL MIDDLE 22 ATTS WITH BLUE PAPER,WHITE INK
            ;
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
            ;MEANWHILE THE TV SCAN HAS REACHED THE RIGHT-HAND
            ;EDGE, SO CHANGE BORDER COLOUR NOW.
            ;
D3FE                OUT     (ØFEH),A
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
2C                  INC     L
73                  LD      (HL),E
```

```
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
7D          LD      A,L
68          LD      L,B
      ;
      ;NOW FILL LEFT 5 ATTS WITH BLUE PAPER, WHITE INK
      ;
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
6F          LD      L,A
2C          INC     L
      ;FINALLY FILL RIGHT 5 ATTS WITH BLUE PAPER,
      ;WHITE INK
      ;
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
2C          INC     L
73          LD      (HL),E
68          LD      L,B
      ;
      ;STORE START OF ATT LINE
E5    INIT3 PUSH    HL
```



```
        ;ENSURE THAT THE BUFFER IS FILLED WITH FAKE
        ;CHARACTERS
        ;
21ØØØØ          LD      HL,CHSTRE
7E              LD      A,(HL)
A7              AND     A
28ØF            JR      Z,END
11Ø6ØØ  INIT2   LD      DE,6
        ;
        ;NOTE   D=Ø
72              LD      (HL),D
21ØØØØ          LD      HL,BUFFER
22ØØØØ          LD      (BUFFPT),HL
47              LD      B,A
72      NXTFL   LD      (HL),D
19              ADD     HL,DE
1ØFC            DJNZ    NXTFL
        ;
        ;RETRIEVE ATT ADDRESS
        ;
E1      END     POP     HL
        ;
        ;H=Ø MEANS NO ATTS TO FILL
        ;
7C              LD      A,H
A7              AND     A
CAB7ØØ          JP      Z,NOPLG
        ;
        ;WAIT TILL TV HAS FINISHED WITH THIS
        ;ATTRIBUTE LINE
        ;N.B. IF FLICKERING OCCURS THEN
        ;INCREASE THIS DELAY
        ;
Ø64D            LD      B,4DH
1ØFE    SELF4   DJNZ    SELF4
        ;
        ;FILL THE LINE WITH CYAN PAPER, WHITE INK
        ;
ØE1F            LD      C,31
54              LD      D,H
5D              LD      E,L
1C              INC     E
```

```
362F     HCOL4   LD      (HL),47
EDBØ             LDIR
                 ;
                 ;RETRIEVE REGISTERS
                 ;
E1       NOPLG   POP     HL
D1               POP     DE
C1               POP     BC
F1               POP     AF
                 ;
                 ;END INTERRUPT
                 ;
FB               EI
ED4D             RETI
                 ;
                 ;
                 ;DELAY FOR HORIZON AT ROW1
                 ;
Ø6ØF     WT1LN   LD      B,15
1ØFE     SELF11  DJNZ    SELF11
E6FF             AND     ØFFH
23               INC     HL
3D               DEC     A
2B               DEC     HL
3C               INC     A
                 ;
                 ;ENTERS HERE FOR ROW ZERO HORIZON
                 ;
                 ;FIND NEW BORDER COLOUR
                 ;
2A2CØØ   NOWAIT  LD      HL,(BOTBRD)
11ØØØØ           LD      DE,BORD
1A               LD      A,(DE)
E61Ø             AND     16
AC               XOR     H
                 ;
                 ;STORE IT AND OUTPUT IT
                 ;
12               LD      (DE),A
D3FE             OUT     (ØFEH),A
                 ;
                 ;SET FLAG FOR NO ATTS TO FILL
```



```
26ØØ             LD      H,Ø
                 ;
                 ;JUMP BACK TO MAIN ROUTINE
                 ;
C38DØØ           JP      INIT3
```

We will, of course, be using interrupt mode 2, and as you will see, I have elected to use a 257 byte vector table pointing to a jump instruction at FDFDH to the interrupt handler. This technique was described more fully in Chapter 7. By putting the table at FEØØH and the buffer at FF1ØH (remember, the low byte must be 1ØH) I have neatly used up the last ½K of RAM, wasting only fifteen bytes. The following initialization routine sets up the vector table, selects IM2 and then jumps into the interrupt handler in order to ensure that the buffer is clear. I have called the routine INT1, and its counterpart for reselecting IM1 (should you wish to return to BASIC) DISINT.

The last three lines of the listing set up the all-important JP at FDFDH.

```
                 ;INITIALIZE INTERRUPT PROCESSOR
                 ;PRESERVE REGISTERS AS WE EXIT VIA THE
                 ;INTERRUPT HANDLER
                 ;
F3       INT1    DI
F5               PUSH    AF
C5               PUSH    BC
D5               PUSH    DE
E5               PUSH    HL
3EFE             LD      A,ØFEH
ED47             LD      I,A
                 ;
                 ;SET UP VECTOR TABLE FOR IM 2 BY FILLING
                 ;257 BYTES FROM ØFEØØH WITH ØFDH
                 ;
21ØØFE           LD      HL,ØFEØØH
45               LD      B,L
3D               DEC     A
77       TBLP    LD      (HL),A
23               INC     HL
1ØFC             DJNZ    TBLP
77               LD      (HL),A
                 ;
ED5E     SELECT IM 2 AND PREPARE FOR....
```

```
3E28              LD      A,40
ED5E              IM      2
2600              LD      H,0
E5                PUSH    HL
210000            LD      HL,CHSTRE
          ;
          ;A JUMP INTO THE INTERRUPT HANDLER TO CLEAR
          ;THE PRINT BUFFER
C30000            JP      INIT2
          ;
          ;
          ;USE THIS ROUTINE TO RESELECT IM 1
3E3E      DISINT  LD      A,3EH
ED56              IM      1
ED47              LD      I,A
C9                RET
          ;
          ;POSITION THE JUMP INSTRUCTIONS TO THE
          ;INTERRUPT HANDLER

          LABEL   ORG     0FDFDH
C30000            JP      INTERP
                  ORG     LABEL
```

In the next chapter I shall produce a suite of routines to deal with every aspect of horizon generation and movement. Following that will be a powerful set of sprite animation routines to make full use of the (as yet unused) print-processor.

If you do not wish to use the full-screen horizon generator in the interrupt handler, then you can 'turn it off' with the following sequence.

```
LD      A,<BORDER COLOUR>
LD      (TOPBRD+1),A
LD      (BOTBRD+1),A
XOR     A
LD      (ROWS+1),A
```

This sets the horizon to its maximum level with the same coloured sea and sky, and causes the routine to skip any work on the attributes. You may then safely ignore the next chapter!

CHAPTER 10
Moving the Full-Screen Horizon by Pixels

Before I supply the horizon control routines, a word or two of caution about the use of the interrupt processor is required.

Although the Z-80 always receives an interrupt signal at exactly the same stage of the TV frame, it will never react to it until it has finished processing the current instruction. This can lead to a variation of as much as 23 T-states (as in the case of one of the longest instructions, EX(SP),IX) in the time at which the interrupt is processed.

In human terms this does not seem very long, but it is long enough for the TV to progress about one tenth of the way through a row, thereby affecting our 'artificial' horizon, shifting the central, level portion momentarily to the left or right. In fact, the horizon generator can tolerate a time displacement of about $\pm$ 4 T-states with no ill effect, so as long as we get back to a HALT instruction (which makes the processor continuously execute NOPs at 4 T-states each) before an interrupt occurs then there will be no problem.

If, however, you wish to execute some other routine while expecting an interrupt, then you will probably find that flickering occurs. In this case the solution is to extend our 'ink rows', which as you recall are normally five columns in width, either side of the central horizon, until they completely cover the flickering area. You should not need to make them wider than seven columns on each side, leaving eighteen untouched columns in the centre of the screen. Naturally you will then need to make slight modifications to all of the horizon routines so that they calculate the correct addresses for, and correctly manipulate, the new attributes and ink rows. The same goes for the horizon generator in the interrupt handler.

I shall now begin development of the first horizon routine, HRZST1. Its function will be to delete the last ink rows inserted on the screen, and calculate the addresses of the new ones, after we have moved the horizon. We will store the address of the left five ink rows in HRZN1, and that of the right five in HRZN2. When no ink rows are required (if the horizon is not in the text or is between two lines) then we will set the high order of HRZN1 and HRZN2 to zero, as flags.

The address of the current attribute line will always be stored in HRZN4, and also inserted in the interrupt handler itself at (HRZN3+1). The high order of the latter will be set to zero when no attribute manipulation is necessary, thereby pointing the horizon generator in the interrupt handler at the ROM. This is the easiest way to ensure that the generator still gets the timing right for the border change.

We will thus need to define the variables at the start of the program:

```
                ORG     (YOUR ADDRESS)
        HRZN1   DEFW    Ø
        HRZN2   DEFW    Ø
        HRZN4   DEFW    Ø
```

Now for the listing of HRZST1, followed by notes on its use.

```
        ;DELETE OLD INK ROWS AND SET UP
        ;NEW VALUES FOR LOCATIONS OF INK ROWS
        ;AND ATTRIBUTES
        ;ENTRY: C=NO. OF TEXT ROWS ABOVE THE
        ;HORIZON
        ;PRESERVED: DE,C
        ;
2AØØØØ  HRZST1  LD      HL,(HRZN1)
        ;
        ;CLEAR THE LEFT FIVE INK ROWS
        ;
AF              XOR     A
77              LD      (HL),A
2C              INC     L
77              LD      (HL),A
2C              INC     L
77              LD      (HL),A
2C              INC     L
```



```
77              LD      (HL),A
2C              INC     L
77              LD      (HL),A
        ;
        ;NOW THE RIGHT FIVE
        ;
2AØØØØ          LD      HL,(HRZN2)
77              LD      (HL),A
2C              INC     L
77              LD      (HL),A
2C              INC     L
77              LD      (HL),A
2C              INC     L
77              LD      (HL),A
2C              INC     L
77              LD      (HL),A
79      HRZST2  LD      A,C
32ØØØØ          LD      (ROWS+1),A
        ;IS THE HORIZON STILL IN THE TEXT AREA?
        ;
FEC1            CP      ØC1H
        ;
        ;IF NOT THEN NO INK ROWS ARE NEEDED
3Ø31            JR      NC,NOWRK
        ;
        ;LOCATE ATTRIBUTE LINE
        ;
Ø7              RLCA
Ø7              RLCA
E6Ø3            AND     3
F658            OR      58H
67              LD      H,A
79              LD      A,C
87              ADD     A,A
87              ADD     A,A
E6EØ            AND     ØEØH
6F              LD      L,A
        ;STORE IT
22ØØØØ          LD      (HRZN4),HL
79              LD      A,C
        ;
        ;IF HORIZON IS BETWEEN TWO LINES THEN NO
```

```
          ;INK ROWS NEEDED
          ;
E6Ø7            AND     7
281C            JR      Z,NOWRK
          ;
          ;LOCATE 28TH ATTRIBUTE
          ;
7D              LD      A,L
F61B            OR      1BH
45              LD      B,L
6F              LD      L,A
          ;
          ;INSERT IT IN INTERRUPT HANDLER
22ØØØØ          LD      (HRZN3+1),HL
          ;LOCATE 28TH BYTE OF D.F. ROW BELOW HORIZON
          ;
79              LD      A,C
1F              RRA
37              SCF
1F              RRA
A7              AND     A
1F              RRA
A9              XOR     C
E6F8            AND     ØF8H
A9              XOR     C
67              LD      H,A
22ØØØØ          LD      (HRZN2),HL
          ;NOW FIRST BYTE OF ROW ABOVE HORIZON
          ;
25              DEC     H
68              LD      L,B
22ØØØØ          LD      (HRZN1),HL
C9              RET
AF        NOWRK XOR     A
          ;WHEN NO INK ROWS ARE NEEDED,POINT
           THE VARIABLES ;AT THE ROM
32ØØØØ          LD      (HRZN1+1),A
32ØØØØ          LD      (HRZN2+1),A
32ØØØØ          LD      (HRZN3+2),A
C9              RET
```


HRZST1 will be called by the next routine, HRZMV1, whenever it moves the horizon (ST for SeT, MV for MoVe). You may also call it directly to move the horizon to any level on the screen in one go, from any 'old' level. Just

```
LD      C,(NO. OF ROWS ABOVE HORIZON)
CALL    HRZST1
```

. . . then fill in the ink rows and attributes as required. If the horizon level is being set for the first time, then you should skip the section that 'blanks out' the old ink rows by entering at HRZST2.

To develop a routine to move the horizon, HRZMV1, we shall first define two variables. HRZSPD will hold the number of rows moved by the horizon every time the routine is called. This will vary between one and eight. The direction of the horizon will be stored in CNTRL, which may be set (for example) by a keyboard scanning routine. Bit 2 of CNTRL will be set (value 4) for downwards movement, and bit 3 (value 8) for upwards. So we start with the lines

```
        ORG     (YOUR ADDRESS)
HRZSPD  DEFB    Ø
CNTRL   DEFB    Ø
```

Due to the problem of a horizon on row one, described in the previous chapter, I have limited movement to levels below this. On my TV there are about 236 rows from the top of the text area to the bottom of the screen, so I have set the lower limit at (ROWS+1)=236. You will probably want to alter this for your own TV, and in any case you must remember that the lower the horizon, the longer it takes to generate it and thus the less time you have to do anything else before the next interrupt (like animating sprites, for example!).

The main function of HRZMV1 is to take care of the attributes as the horizon moves. If, for example, we have just moved the horizon up to row Ø of the current line, or into the line above it, then we will need to change a line of attributes from sky to sea. Similarly, if we have just moved down onto a new line then we must fill its attributes with sky, ready for the next interrupt. HRZMV1 will be called by the forthcoming master horizon routine, HRZNMK. When the routine HRZMV1 is completely satisfied with the attributes, it makes a jump to HRZST1 in order to set up the new values for HRZN1 to HRZN4 and blank out any old ink rows.

You may notice two unused labels in the listing, HCOL5 and HCOL6. These will be used by a later routine which will set up new sky and sea colours. Here comes the listing . . .

```
                ;ROUTINE TO CHANGE HORIZON LEVEL
                ;BY AMOUNT (HRZSPD) IN DIRECTION
                ;(CNTRL) NOTE 4=DOWN, 8=UP
                ;
                ;
3A0000          HRZMV1  LD      A,(HRZSPD)
47                      LD      B,A
3A0000                  LD      A,(CNTRL)
110000                  LD      DE,0
                ;
                ;TEST FOR UPWARDS
                ;
CB5F                    BIT     3,A
C23C00                  JP      NZ,UP2
                ;
                ;TEST FOR DOWNWARDS
                ;
CB57                    BIT     2,A
C8                      RET     Z
                ;
                ;INCREASE ROWS ABOVE HORIZON BY HRZSPD
                ;
3A0000                  LD      A,(ROWS+1)
80                      ADD     A,B
                ;
                ;SAFETY CHECK FOR MINIMUM HORIZON LEVEL 1
                ;
FEEC                    CP      236
D0                      RET     NC
                ;
                ;IF ROWS>192 THEN SKIP TO HRZST1
                ;
FEC1                    CP      0C1H
4F                      LD      C,A
D20000                  JP      NC,HRZST1
                ;
                ;ARE WE ON ROW ZERO OF A LINE?
                ;
E607                    AND     7
2008                    JR      NZ,NROWZ1
                ;
                ;IF SO,THEN IS HRZSPD AT 8 ROWS PER MOVE?
```



```
                ;
CB58                    BIT     3,B
                ;
                ;IF NO, THEN SKIP TO HRZST1
                ;
CA0000                  JP      Z,HRZST1
1808                    JR      ROWZ1
B8              NROWZ1  CP      B
                ;
                ;ARE WE MOVING FROM ROW 0 OF A LINE?
                ;
2805                    JR      Z,ROWZ1
                ;
                ;OTHERWISE, IF WE ARE STILL ON THE SAME
                ;LINE THEN JUMP
                ;
D20000                  JP      NC,HRZST1
1E20                    LD      E,20H
                ;
                ;DELAY TO ENSURE TV HAS FINISHED THE NEW LINE
                ;
064D            ROWZ1   LD      B,77
10FE            SELF42  DJNZ    SELF42
                ;
                ;FILL THE NEW LINE WITH CYAN PAPER, WHITE INK
                ;
062F            HCOL5   LD      B,47
C36000                  JP      UPINIT
                ;
3A0000          UP2     LD      A,(ROWS+1)
                ;
                ;DECREASE ROWS ABOVE HORIZON BY HRZSPD
                ;
90                      SUB     B
                ;
                ;RETURN IF NEGATIVE
                ;
D8                      RET     C
                ;
                ;RETURN IF LESS THAN 2
                ;
FE02                    CP      2
```

```
D8                  RET     C
            ;
            ;IF ROWS>184 THEN SKIP TO HRZST1
            ;
FEB9                CP      ØB9H
4F                  LD      C,A
D2ØØØØ              JP      NC,HRZST1
ED44                NEG
E6Ø7                AND     7
11ØØØØ              LD      DE,Ø
            ;
            ;JUMP IF NOT ON ROW Ø OF A LINE
            ;
2ØØ7                JR      NZ,NROWZ2
            ;
            ;OTHERWISE FILL IN THE CURRENT LINE
            ;WITH BLUE PAPER, WHITE INK
            ;
CB58                BIT     3,B
28Ø7                JR      Z,HCOL6
11EØFF              LD      DE,ØFFEØH
B8          NROWZ2  CP      B
            ;
            ;IF WE ARE STILL ON THE SAME LINE THEN JUMP
            ;
D2ØØØØ              JP      NC,HRZST1
            ;
            ;OTHERWISE FILL THE OLD ONE WITH BLUE PAPER
            ;WHITE INK
Ø6ØF        HCOL6   LD      B,15
2AØØØØ      UPINIT  LD      HL,(HRZN4)
19                  ADD     HL,DE
            ;
            ;THE ALL-PURPOSE FILLER
            ;
54                  LD      D,H
5D                  LD      E,L
1C                  INC     E
7Ø                  LD      (HL),B
79                  LD      A,C
Ø11FØØ              LD      BC,31
EDBØ                LDIR
```



```
4F                  LD      C,A
            ;
            ;FINALLY JUMP TO HRZST1
            ;
C3ØØØØ              JP      HRZST1
```

We now have all the code necessary to set and move the horizon level. HRZMV1 copes with all work on the attributes as the horizon moves, while HRZST1 makes sure that we know where those attributes are, deletes the old ink rows and calculates the addresses of the new ones. All that remains is to actually insert those ink rows into the display file before every interrupt (they may have been overprinted by sprites since the last one). The master routine HRZNMK (MK for MaKe) will do this after having called HRZMV1, which makes any necessary changes to the attributes and variables. Thus HRZNMK is the only routine that we have to call directly after each interrupt, as will be seen in the demonstration routine following its listing.

```
            ;THE MAIN HORIZON ROUTINE
            ;JUST CALL THIS AFTER EACH
            ;INTERRUPT, HAVING SET THE VARIABLES
            ;CNTRL AND HRZSPD
            ;
CDØØØØ      HRZNMK  CALL    HRZMV1
            ;
            ;RETURN IF NO INK ROWS ARE NEEDED
            ;
2AØØØØ              LD      HL,(HRZN1)
24                  INC     H
25                  DEC     H
C8                  RET     Z
            ;
            ;INSERT THE INK ROWS FOR THE HORIZON
            ;
3EFF                LD      A,ØFFH
            ;
            ;FIRST THE LEFT FIVE
            ;
77                  LD      (HL),A
2C                  INC     L
77                  LD      (HL),A
2C                  INC     L
77                  LD      (HL),A
```

```
2C                  INC     L
77                  LD      (HL),A
2C                  INC     L
77                  LD      (HL),A
          ;
          ;NOW THE RIGHT FIVE
          ;
2AØØØØ              LD      HL,(HRZN2)
77                  LD      (HL),A
2C                  INC     L
77                  LD      (HL),A
2C                  INC     L
77                  LD      (HL),A
2C                  INC     L
77                  LD      (HL),A
2C                  INC     L
77                  LD      (HL),A
C9                  RET
```

To illustrate your new-found power over the Spectrum, here is a demonstration routine that employs INT1, HRZST2, HRZNMK, DISINT and indirectly, HRZST1, HRZMV1 and the interrupt handler.

The routine gives you direct control over the horizon. Pressing any of keys 1 to 5 moves it upwards, while keys CAPS SHIFT to V move it downwards. Keys 8,9 and Ø are used to control the speed of the horizon. Think of them as a three bit number, each bit being set when its key is depressed, then add one to obtain HRZSPD. Thus presing keys 8 and Ø (1Ø1 binary=5 decimal) gives a speed of 6 rows per TV frame. Here then is the routine, DEMO.

```
CDØØØØ    DEMO      CALL    INT1
          ;
          ;SET INITIAL HORIZON
          ;
ØE54                LD      C,84
CDØØØØ              CALL    HRZST2
          ;
          ;WAIT FOR INTERRUPT
          ;
76        DMLP      HALT
          ;
          ;C WILL HOLD DIRECTION
```



```
          ;
ØEØØ                LD      C,Ø
          ;
          ;TEST BOTTOM-LEFT HALF-ROW
          ;
3EFE                LD      A,ØFEH
DBFE                IN      A,(ØFEH)
2F                  CPL
E61F                AND     1FH
28Ø2                JR      Z,ND1
          ;
          ;IF PRESSED, THEN SET BIT 2 FOR "DOWN"
          ;
CBD1                SET     2,C
          ;
          ;TEST TOP-LEFT HALF-ROW
          ;
3EF7      ND1       LD      A,ØF7H
DBFE                IN      A,(ØFEH)
2F                  CPL
E61F                AND     1FH
28Ø2                JR      Z,NU1
          ;
          ;IF PRESSED, THEN SET BIT 3 FOR "UP"
          ;
CBD9                SET     3,C
          ;
          ;STORE DIRECTION
          ;
79        NU1       LD      A,C
32ØØØØ              LD      (CNTRL),A
          ;
          ;TEST TOP-RIGHT HALF-ROW
          ;
3EEF                LD      A,ØEFH
DBFE                IN      A,(ØFEH)
          ;
          ;USE RIGHT 3 KEY-BITS FOR HORIZON SPEED
          ;
2F                  CPL
E6Ø7                AND     7
3C                  INC     A
```

```
32ØØØØ              LD      (HRZSPD),A
             ;
             ;CALL THE MASTER HORIZON ROUTINE
             ;
CDØØØØ              CALL    HRZNMK
             ;
             ;TEST BREAK KEY
             ;
3E7F                LD      A,7FH
DBFE                IN      A,(ØFEH)
1F                  RRA
38CE                JR      C,DMLP
             ;
             ;IF PRESSED, THEN BACK TO BASIC
             ;
CDØØØØ              CALL    DISINT
C9                  RET
```

For the final routine in this 'horizon suite' I have produced HRZCOL, which allows you to set up the other routines for any combination of sea and sky colours. It also sets the ink colours for above and below the horizon (if you are moving shapes over the horizon, then you will probably want these two to be the same). You have the option of causing the interrupt handler to generate a background 'motor' sound, either at 5Ø Hz or 1ØØ Hz, by adding 16 to one or both of the paper values respectively. The registers should then be prepared as follows:

H = sea paper value (+16 for sound)
L = sky paper value (+16 for 1ØØ Hz sound)

B = sea ink value
C = sky ink value.

For example, to produce green ground and a white sky, with black ink in both,

```
LD      HL,Ø4Ø7H
LD      BC,ØØØØH
CALL    HRZCOL
```

The routine simply inserts the correct attributes at the labels HCOL1 to HCOL6, which are to be found in the interrupt handler (HCOL1 to HCOL4) and HRZMV1 (HCOL5 and HCOL6).


```
             ;ROUTINE TO SET COLOURS OF SEA AND SKY
             ;ENTRY: H=SEA PAPER, L=SKY PAPER
             ;B=SEA INK, C=SKY INK
             ;ADD 16 TO H OR L OR BOTH FOR SOUND
             ;
             ;
7C           HRZCOL LD      A,H
32ØØØØ              LD      (BOTBRD+1),A
             ;
             ;LEAVE SEA PAPER IN H
             ;
E6Ø7                AND     7
67                  LD      H,A
7D                  LD      A,L
32ØØØØ              LD      (TOPBRD+1),A
             ;
             ;LEAVE SKY PAPER IN L
             ;
E6Ø7                AND     7
6F                  LD      L,A
             ;
             ;HCOL1 NEEDS SKY-PAPER PAPER AND SEA PAPER INK
             ;
Ø7                  RLCA
Ø7                  RLCA
Ø7                  RLCA
5F                  LD      E,A
B4                  OR      H
32ØØØØ              LD      (HCOL1+1),A
             ;
             ;HCOL2 NEEDS SEA-PAPER PAPER AND SKY-PAPER INK
             ;
7C                  LD      A,H
Ø7                  RLCA
Ø7                  RLCA
Ø7                  RLCA
57                  LD      D,A
B5                  OR      L
32ØØØØ              LD      (HCOL2+1),A
             ;HCOL3 NEEDS SEA-PAPER PAPER AND SEA-INK INK
             ;
7A                  LD      A,D
```

```
BØ                OR     B
32ØØØØ            LD     (HCOL3+1),A
           ;
           ;...AS DOES HCOL6
           ;
32ØØØØ            LD     (HCOL6+1),A
           ;
           ;HCOL4 NEEDS SKY-PAPER PAPER AND SKY-INK INK
           ;
7B                LD     A,E
B1                OR     C
32ØØØØ            LD     (HCOL4+1),A
           ;
           ;....AS DOES HCOL5
           ;
32ØØØØ            LD     (HCOL5+1),A
C9                RET
```


CHAPTER 11

A Suite of Routines to Complement the Print-Processor

In this chapter I shall be developing a complete set of printing routines, to take full advantage of the interrupt-driven print-procesor produced in Chapter 9. In the following chapter will come the sprite animation routines.

To start with, it would be useful to have a simple routine with which to send any character to the buffer. Rather than using the address of the current cell in the display file, as does Spectrum BASIC with the system variable DF-CC, I find it more convenient to keep track of the print position using the attributes file. Thus we will define a variable ATCC to hold the attribute address of the current cell, and start with the lines

```
        ORG     (YOUR ADDRESS)
ATCC    DEFW    58ØØH
```

. . . thereby initialising our marker to the top-left corner of the text area. The base of the table holding the character data will be stored in CHARS, and we may as well start by pointing it at the Spectrum BASIC character set, using the line

```
CHARS   DEFW    3CØØH
```

Remember that CHSTRE holds the number of used entries in the print buffer, and BUFFPT points at the next free entry. Both variables are altered accordingly. The rest of the routine is self-explanatory, so without further ado, I hereby present HIPRINT to you.

```
                ;SEND A CHARACTER TO THE BUFFER
                ;ENTRY: A=CHARACTER CODE
                ;EXIT: BC=ADDRESS OF CHARACTER DATA
                ;DE=ATCC (SEE TEXT)
                ;HL=NEXT BUFFER ENTRY
                ;A=HI BYTE OF D.F. ADDRESS
                ;
                ;MULTIPLY CODE BY 8
                ;
6F              HIPRNT  LD      L,A
26Ø0            LD      H,Ø
29              ADD     HL,HL
29              ADD     HL,HL
29              ADD     HL,HL
                ;
                ;ADD BASE ADDRESS OF TABLE
                ;
ED5BØØØØ                LD      DE,(CHARS)
19                      ADD     HL,DE
                ;
                ;LET BC = DATA ADDRESS
                ;
44                      LD      B,H
4D                      LD      C,L
21ØØØØ          PLACE   LD      HL,CHSTRE
7E              PWAIT   LD      A,(HL)
                ;
                ;IF THE BUFFER IS FULL THEN WAIT FOR AN
                ;INTERRUPT
                ;
FE28                    CP      4Ø
DA19ØØ                  JP      C,GO
FB                      EI
18F7                    JR      PWAIT
F3              GO      DI
                ;
                ;UPDATE BUFFER CHARACTER COUNT
                ;
34                      INC     (HL)
ED5BØØØØ                LD      DE,(ATCC)
                ;
                ;DE=ADDRESS OF ATTRIBUTE
```



```
                ;
2AØØØØ                  LD      HL,(ATT)
                ;
                ;H=MASK, L=ATT
                ;CONSTRUCT NEW ATTRIBUTE
                ;
1A                      LD      A,(DE)
AD                      XOR     L
A4                      AND     H
AD                      XOR     L
2AØØØØ                  LD      HL,(BUFFPT)
                ;
                ;HL=FIRST FREE BUFFER LOCATION
                ;ROTATE AND STORE ATTR
                ;
Ø7                      RLCA
77                      LD      (HL),A
                ;
                ;STORE ATTR. ADDRESS
                ;
2C                      INC     L
73                      LD      (HL),E
23                      INC     HL
72                      LD      (HL),D
2C                      INC     L
                ;
                ;FIND AND STORE HI BYTE OF D.F. ADDRESS
                ;
7A                      LD      A,D
E6Ø3                    AND     3
Ø7                      RLCA
Ø7                      RLCA
Ø7                      RLCA
F64Ø                    OR      64
77                      LD      (HL),A
23                      INC     HL
                ;
                ;STORE CHARACTER DATA ADDRESS
                ;
71                      LD      (HL),C
2C                      INC     L
7Ø                      LD      (HL),B
```

```
23                 INC     HL
              ;
              ;SET BUFFPT TO NEXT FREE BUFFER ENTRY
              ;
220000             LD      (BUFFPT),HL
C9                 RET
```

There will be times during some programs when you want the interrupt handler to continually print a particular character or set of characters in the same place on the screen. For example, you may wish to superimpose 'laser target sights' on the centre of the screen, and these would need continual OR-printing on every interrupt, as the enemy spaceship (or whatever) moves behind them. Since time is so short between two interrupts (especially if you are using the horizon generator, at low level), it would be highly preferable not to have to load all the data for the target sights into the print buffer after every interrupt.

In order to enable this, I have devised a system of routines which use a subsection of the print buffer, which shall be called 'RO-buffer' for 'Read-Only buffer'. Unlike normal entries in the print buffer, those in the RO-buffer will not be erased by the interrupt handler when they have been printed. Thus all that we need to do before each interrupt is to make sure the correct attributes for each cell concerned have been inserted in the RO-buffer. If the attribute mask is the zero byte, then we will not even need to do this, since the characters will always be printed with the same attributes, regardless of the 'old' attributes for that cell.

Consecutive entries normally 'grow' upwards from the start of the print buffer: so to keep them separate, we will make the RO-buffer grow downwards from the end of the print buffer. The two regions should never overlap. We will store the number of entries in the RO-buffer in the variable ROLNTH, and the start (lowest address) of the RO-buffer in the variable ROBFPT. Thus we initialise them (for a zero-length RO-buffer) with the lines.

```
        ORG     (YOUR ADDRESS)
ROLNTH  DEFB    0
ROBFPT  DEFW    0
```

Before we go any further we need a routine to set up a RO-buffer. What, in fact, the following routine will do is to ALTER the length of the RO-buffer by the value in C, which may be positive or negative. Having adjusted ROLNTH, the routine fills all the entries between the end of the 'normal' entries (i.e. BUFFPT) and the beginning of the RO-buffer, with 'null' characters, to prevent any garbage being printed. For this reason

the routine, called ALTRBF, should always be called with the interrupts disabled. ALTRBF then sets ROBFPT to the new start address of the RO-buffer, and returns it in HL, which will be used later.

```
        ;ROUTINE TO ALTER LENGTH OF RO-BUFFER
        ;ENTRY: C=ALTERATION TO LENGTH
        ;PRESERVED: C
        ;EXIT: HL=START OF RO-BUFFER, B=0, A=NO. OF
        ;UNUSED ENTRIES IN PRINT BUFFER
        ;
        ;
210000  ALTRBF  LD      HL,ROLNTH
        ;
        ;ALTER ROLNTH BY C
        ;
7E              LD      A,(HL)
81              ADD     A,C
77              LD      (HL),A
        ;
        ;FIND NO. OF UNUSED ENTRIES IN PRINT BUFFER
        ;(>=0)
        ;
3A0000          LD      A,(CHSTRE)
47              LD      B,A
3E28            LD      A,40
90              SUB     B
96              SUB     (HL)
47              LD      B,A
        ;
        ;FILL THEM WITH "NULL" ENTRIES
        ;
2A0000          LD      HL,(BUFFPT)
        ;
        ;BUT JUMP IF THERE ARE NO ENTRIES TO BE FILLED
        ;
2807            JR      Z,HOPFL
110600          LD      DE,6
        ;
        ;NOTE: D=0
        ;
72      BLNK    LD      (HL),D
19              ADD     HL,DE
```

```
10FC          DJNZ   BLNK
        ;
        ;RESET ROBFPT TO START OF RO-BUFFER
        ;
220000 HOPFL  LD     (ROBFPT),HL
C9            RET
```

Obviously any routine written to output characters to the print buffer is easily adapted to use the RO-buffer, which is indeed a subsection of the former. You could modify HIPRNT or any of your own printing routines. I shall be supplying a routine to dump pre-defined shapes such as our target-sight into the RO-buffer, but first let me get the simple routine needed to refresh the attribute bytes of each entry out of the way.

The routine is called SRVR1 (for it is a SeRVice Routine). It takes the attribute address from an entry, finds the attribute from the file, then masks it with our standard variable MASK, which as usual is placed directly after ATT, the attributes for our characters. The completed attribute byte is then rotated left by one bit (to counter the rightwards rotation by the interrupt handler) and inserted in the RO-buffer. Note that if we wish to select OR-printing then we set bit 7 of ATT (bit 7 of MASK should always be zero), which will then be rotated to bit 0 before insertion in the buffer. This also applies to HIPRNT.

Here, then, is the listing for SRVR1. Note the use of the zero flag to detect the end of the buffer, when the lo-byte of its address becomes zero.

```
                ;SERVICE ROUTINE TO UPDATE ATTRIBUTES
                ;IN THE RO-BUFFER
                ;EXIT: B=MASK, C=ATTRIBUTE, A=0
                ;HL=BYTE AFTER PRINT BUFFER
                ;
2A0000          SRVR1   LD      HL,(ROBFPT)
                ;
                ;LET B=MASK, C=ATTRIBUTE
                ;
ED4B0000                LD      BC,(ATT)
                ;
                ;TAKE ATTRIBUTE ADDRESS
                ;
2C              NXSRV1  INC     L
5E                      LD      E,(HL)
2C                      INC     L
56                      LD      D,(HL)
```



```
2D                      DEC     L
2D                      DEC     L
                ;
                ;TAKE CURRENT ATTRIBUTE
                ;
1A                      LD      A,(DE)
                ;
                ;CREATE NEW ONE
                ;
A9                      XOR     C
A0                      AND     B
A9                      XOR     C
                ;
                ;ROTATE LEFT, PRESERVING THE OR-FLAG
                ;
07                      RLCA
                ;
                ;STORE NEW ATTRIBUTE IN BUFFER
                ;
77                      LD      (HL),A
                ;
                ;MOVE ON TO NEXT ENTRY
                ;
7D                      LD      A,L
C606                    ADD     A,6
6F                      LD      L,A
20EE                    JR      NZ,NXSRV1
                ;
                ;UNTIL RO-BUFFER HAS BEEN SERVICED
C9                      RET
```

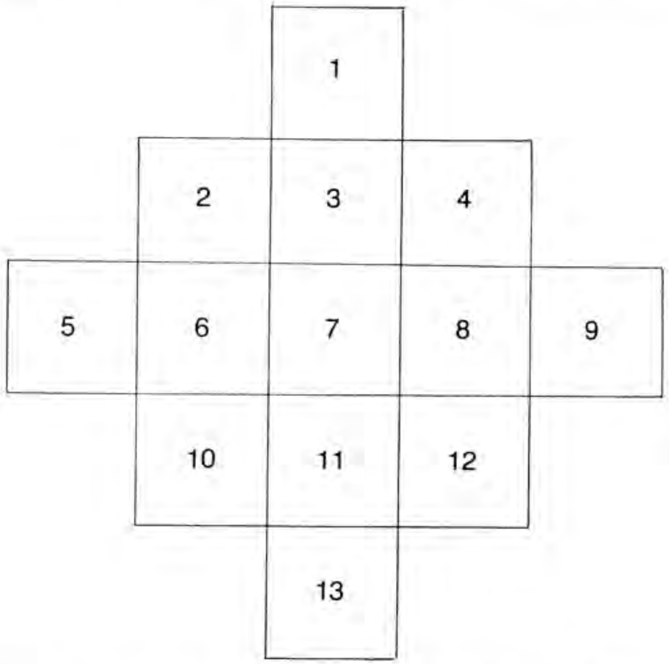
SRVR1 should be called whenever there is a chance that the attributes of the cells used by our RO-buff entries have been changed, for example, by moving a sprite into them, or a horizon down one line. The routine also gives you the opportunity to vary the colour of the 'permanent' characters on your screen, by altering ATT. Since it deals with the entire RO-buffer, every entry will use the same ATT value. If this is not desired, then the easiest modification is to change the fragment

```
XOR     C
AND     B
XOR     C
```

```
to      LD      C,(HL)
        RRC     C
        XOR     C
        AND     B
        XOR     C
```

thereby effectively using the original value of ATT with which the entry was inserted (by a modified HIPRNT, for example).

Now for that rather specialised routine I mentioned earlier, to send specific 'shapes' to the RO-buffer. A shape will consist of a number of separate characters that together form an image to be printed on the screen. As an example, I shall be taking the previously-discussed laser target sight, which will be made up of thirteen characters as shown.

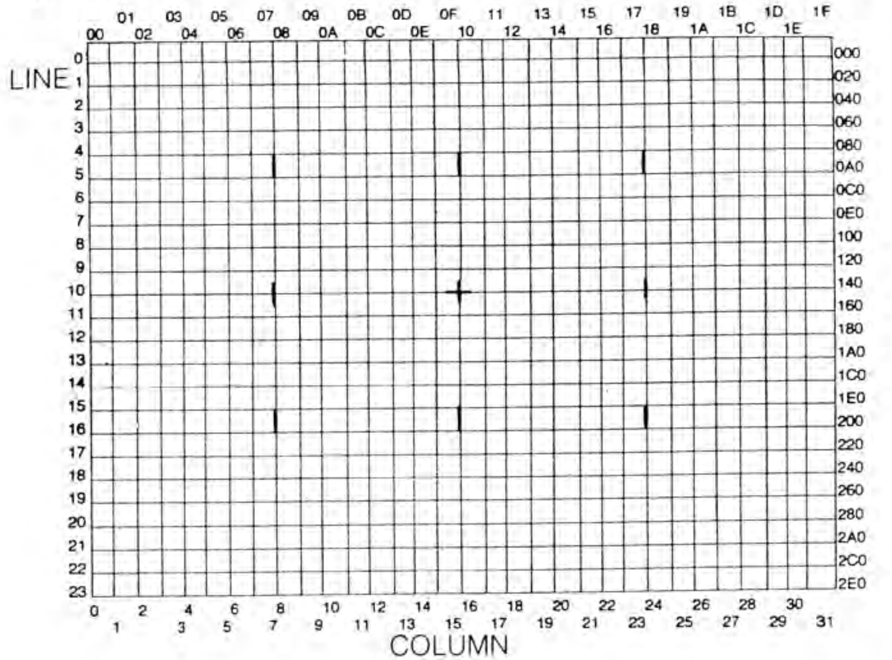


The routine, called SRVR2, will need two data tables. One will hold the desired position of each character on the screen, while the other will be a table of character data, stored, as usual, in eight bytes per character. The position data and character data must, of course, be stored in the same order, with no gaps in the tables!

To represent the position of each character, I have decided to number the cells Ø to Ø2FFH, in the order of their attributes. This has the

advantage that the lo-byte of the position will be identical to that of the attribute and display file addresses for the cell. If you prefer to use a co-ordinate system, then it is a simple task to write a routine to convert from one system to the other. Alternatively, you could modify SRVR2.

As an aid to easy calculation of position data, here is a labelled screen grid — just read the line value and add the column value (both in hex.).



The tasks of SRVR2 are to take the position of a character, calculate the attribute address, display file address, and character data address, and store them all in the correct order in the RO-buffer. I will list the routine before demonstrating its use.

```
                ;ROUTINE TO SEND DATA TO RO-BUFFER
                ;ENTRY: HL=START OF POSITION DATA
                ;DE=START OF RESERVED RO-BUFFER AREA
                ;BC=ADDRESS OF CHARACTER DATA
                ;A=NO. OF CHARACTERS
                ;EXIT: A=Ø, BC=8
                ;N.B. AF IS DESTROYED
                ;
C5              SRVR2   PUSH    BC
                ;
```

```
              ;USE BC AS A CONSTANT
              ;
Ø1Ø8ØØ                LD      BC,8
              ;
              ;A BECOMES A COUNTER
              ;
Ø8            NXCHR9  EX      AF,AF'
1C                    INC     E
              ;
              ;TRANSFER LO-BYTE OF ATTRIBUTE ADDRESS
              ;
7E                    LD      A,(HL)
12                    LD      (DE),A
1C                    INC     E
23                    INC     HL
              ;
              ;FORM HI-BYTE OF ATRIBUTE ADDRESS
              ;
7E                    LD      A,(HL)
F658                  OR      58H
12                    LD      (DE),A
1C                    INC     E
              ;
              ;FORM HI-BYTE OF D.F. ADDRESS
              ;
E6Ø3                  AND     3
Ø7                    RLCA
Ø7                    RLCA
Ø7                    RLCA
F64Ø                  OR      4ØH
              ;
              ;STORE IT
              ;
12                    LD      (DE),A
1C                    INC     E
23                    INC     HL
              ;
              ;RETRIEVE CHAR. DATA ADDRESS, SAVE POSITION DATA
              ;ADDRESS
              ;
E3                    EX      (SP),HL
EB                    EX      DE,HL
```



```
              ;
              ;STORE CHAR. DATA ADDRESS IN BUFFER
              ;
73                    LD      (HL),E
2C                    INC     L
72                    LD      (HL),D
2C                    INC     L
              ;
              ;ADD EIGHT TO IT
              ;
EB                    EX      DE,HL
Ø9                    ADD     HL,BC
              ;
              ;SAVE NEW CHAR. DATA ADDRESS, RETRIEVE
               POSITION DATA ADDRESS
              ;
E3                    EX      (SP),HL
Ø8                    EX      AF,AF'
              ;
              ;NEXT CHARACTER
              ;
3D                    DEC     A
2ØDE                  JR      NZ,NXCHR9
E1                    POP     HL
C9                    RET
```

Referring to the previous block-diagram of the laser target, I shall be printing character 1 at (1Ø, 15), so we have

position = 14Ø + ØF
= 14FH

By inspection we see that character 2 is one line down (+2ØH) and one column to the left (−1) so its position is 16EH, character 3 is at 16FH and so on. I will label the start of the position data TRGPOS and that of the character data TRGDAT (the 'donkey work' of creating the character data has been done for you, and you will find it listed in the routine).

The first task in TARGET is to call INT1 to set up the interrupt handler. This must be done before anything else, since, as you recall, INT1 clears the print buffer. We then disable the interrupts, which are not desirable while we are loading the print buffer. To reserve 13 characters in the RO-buff we use

```
        LD      C,13
        CALL    ALTRBF
```

. . . which returns the start address of the RO-buffer in HL. We need to put this in DE, with

```
EX      DE,HL
```

. . . then prepare the other registers for SRVR2.

```
LD      HL,TRGPOS
LD      BC,TRGDAT
LD      A,13
CALL    SRVR2
```

Now choose a full mask (7FH, since bit 7 should always be reset) and OR—print (set bit 7 of ATT, giving ATT=8ØH). Finally, initialize the 'attribute' entries in the RO-buffer with SRVR1.

```
LD      HL,7F8ØH
LD      (ATT),HL
CALL    SRVR1
```

For demonstration purposes I have put the last instruction in the main loop so that SRVR1 is called after every interrupt, but in this case it is not imperative, since the attribute file is not altered within the loop. It WOULD be necessary, however, if you were to incorporate the horizon demonstration routine of the previous chapter.

When the BREAK key is pressed, TARGET terminates by clearing the RO-buffer (setting ROLNTH to zero) and reselecting IM1.

```
CALL    DISINT
LD      A,(ROLNTH)      ;ADD (-ROLNTH) TO ROLNTH
NEG
LD      C,A
CALL    ALTRBF
RET
```

The rest of the listing is sufficiently explained by the comments, so here is TARGET.

```
                ;THIS DEMO OR-PRINTS A HIGH RESOLUTION
                ;RIFLE SIGHT ON THE CENTRE OF THE SCREEN
                ;
                ;CALL INT FIRST, SINCE IT CLEARS THE BUFFER, AS
                ;WELL AS INITIALIZING THE INTERRUPT HANDLER
                ;
CDØØØØ          TARGET  CALL    INT1
```



```
                ;
                ;NO INTERRUPTS WHILE WE ARE ALTERING THE BUFFER
                ;
F3                      DI
                ;
                ;SET UP ROBUFF FOR 13 ENTRIES
                ;
ØEØD                    LD      C,13
CDØØØØ                  CALL    ALTRBF
                ;
                ;PREPARE TO DUMP DATA FOR 13 CHARACTERS IN THE
                ;BUFFER
                ;
EB                      EX      DE,HL
2134ØØ                  LD      HL,TRGPOS
Ø14EØØ                  LD      BC,TRGDAT
3EØD                    LD      A,13
                ;
                ;FILL RO-BUFFER
                ;
CDØØØØ                  CALL    SRVR2
                ;
                ;SELECT FULL MASK, AND OR-PRINT OPERATION BY
                ;SETTING BIT 7 OF ATT
                ;
21807F                  LD      HL,7F8ØH
220000                  LD      (ATT),HL
FB                      EI
                ;
                ;CALCULATE ATTRIBUTES
                ;
CDØØØØ          TSLP    CALL    SRVR1
                ;
                ;WAIT FOR INTERRUPT
                ;
76                      HALT
                ;
                ;TEST BREAK KEY
                ;
3E7F                    LD      A,7FH
DBFE                    IN      A,(ØFEH)
1F                      RRA
```

```
38F5                JR      C,TSLP
          ;
          ;IF PRESSED THEN SELECT IM 1,
           CLEAR THE RO-BUFFER  AND END
          ;
CDØØØØ              CALL    DISINT
3AØØØØ              LD      A,(ROLNTH)
ED44                NEG
4F                  LD      C,A
CDØØØØ              CALL    ALTRBF
C9                  RET
          ;
          ;THE POSITION DATA
          ;
4FØ1      TRGPOS    DEFW    Ø14FH
6EØ1                DEFW    Ø16EH
6FØ1                DEFW    Ø16FH
7ØØ1                DEFW    Ø17ØH
8DØ1                DEFW    Ø18DH
8EØ1                DEFW    Ø18EH
8FØ1                DEFW    Ø18FH
          ;
9ØØ1                DEFW    Ø19ØH
91Ø1                DEFW    Ø191H
AEØ1                DEFW    Ø1AEH
AFØ1                DEFW    Ø1AFH
BØØ1                DEFW    Ø1BØH
CFØ1                DEFW    Ø1CFH
          ;
          ;THE CHARACTER DATA
          ;
ØØ        TRGDAT    DEFB    Ø
18                  DEFB    24
1Ø                  DEFB    16
1Ø                  DEFB    16
18                  DEFB    24
1Ø                  DEFB    16
1Ø                  DEFB    16
18                  DEFB    24
          ;
ØØ                  DEFB    Ø
ØØ                  DEFB    Ø
```



```
ØØ                  DEFB    Ø
Ø3                  DEFB    3
ØE                  DEFB    14
18                  DEFB    24
1Ø                  DEFB    16
3Ø                  DEFB    48
          ;
1Ø                  DEFB    16
1Ø                  DEFB    16
FE                  DEFB    254
93                  DEFB    147
1Ø                  DEFB    16
18                  DEFB    24
1Ø                  DEFB    16
7C                  DEFB    124
          ;
ØØ                  DEFB    Ø
ØØ                  DEFB    Ø
ØØ                  DEFB    Ø
8Ø                  DEFB    128
EØ                  DEFB    224
3Ø                  DEFB    48
1Ø                  DEFB    16
18                  DEFB    24
          ;
ØØ                  DEFB    Ø
ØØ                  DEFB    Ø
ØØ                  DEFB    Ø
92                  DEFB    146
FF                  DEFB    255
ØØ                  DEFB    Ø
ØØ                  DEFB    Ø
ØØ                  DEFB    Ø
          ;
21                  DEFB    33
61                  DEFB    97
43                  DEFB    67
4A                  DEFB    74
FF                  DEFB    255
42                  DEFB    66
43                  DEFB    67
61                  DEFB    97
          ;
```

```
D7              DEFB    215
11              DEFB    17
11              DEFB    17
ØØ              DEFB    Ø
D7              DEFB    215
ØØ              DEFB    Ø
11              DEFB    17
11              DEFB    17
        ;
Ø8              DEFB    8
ØC              DEFB    12
84              DEFB    132
A4              DEFB    164
FF              DEFB    255
84              DEFB    132
84              DEFB    132
ØC              DEFB    12
        ;
ØØ              DEFB    Ø
ØØ              DEFB    Ø
ØØ              DEFB    Ø
92              DEFB    146
FE              DEFB    254
ØØ              DEFB    Ø
ØØ              DEFB    Ø
ØØ              DEFB    Ø
        ;
21              DEFB    33
3Ø              DEFB    48
1Ø              DEFB    16
18              DEFB    24
ØE              DEFB    14
Ø3              DEFB    3
ØØ              DEFB    Ø
ØØ              DEFB    Ø
        ;
D7              DEFB    215
7C              DEFB    124
1Ø              DEFB    16
18              DEFB    24
1Ø              DEFB    16
93              DEFB    147
```



```
FE              DEFB    254
1Ø              DEFB    16
        ;
Ø8              DEFB    8
18              DEFB    24
1Ø              DEFB    16
3Ø              DEFB    48
```

As you will recall, we have made provision in the print-processor part of the interrupt handler for an 'OR printing' function that merges a new character with the current contents of a screen cell in the display file. I will now provide the support routines for this function.

Whenever we come to print a character on the screen, we will need to know whether the contents of the destination cell are to be preserved by OR-printing, or destroyed by over-printing. For example, when two characters in a game move into the same cell, we will probably want to merge them together, while if we are moving a character from one cell to the next, trailing a blank behind it to delete the old image, then we certainly won't want to OR-print the space with the old image.

To this end we need a map in memory, which I shall call the OR-map, to keep track of which cells are 'occupied'. Only one bit per cell is required, a 1 indicating 'cell occupied' and a zero indicating 'empty cell'.

Thus we have four bytes for each of the twenty-four screen lines, making a 96 byte OR-map. Not surprisingly, I have labelled the start of this as OR MAP. To reserve the required space we need the line:

```
ORMAP   DEFS    96
```

The OR-map will need clearing regularly, so before we go any further, let's have a routine to fill it with zeroes, CLOR.

```
                ;ROUTINE TO CLEAR THE OR-MAP
                ;
21ØØØØ          CLOR    LD      HL,ORMAP
Ø15FØØ                  LD      BC,95
7Ø                      LD      (HL),B
54                      LD      D,H
5D                      LD      E,L
13                      INC     DE
EDBØ                    LDIR
C9                      RET
```

Every time we are preparing to send a character to the print buffer and wish it to be considered for OR-printing with existing and future characters, we should access the bit corresponding to the destination cell in the OR-map.

If that bit is set, then there is already something in that cell, and we select OR-print by setting bit 7 of the attribute byte, ATT. If the bit is zero, then as far as we are concerned the cell is 'empty', and having set the OR-map bit to signify that it is now occupied, we reset bit 7 of ATT to select over-printing. The character is then sent to the print-buffer using HIPRINT (or your own routine) in the usual manner.

The following routine, ORCHK, carries out the process described above, using the pointer ATCC, which holds the location of the current attribute byte, as a means of locating the correct OR-map bit. No entry values are required, and the comments in the listing provide adequate explanation.

```
                ;ROUTINE TO DECIDE WHETHER TO OR-PRINT ON THE
                ;CURRENT CHARACTER CELL
                ;
2A0000          ORCHK   LD      HL,(ATCC)
                ;
                ;TAKE ATTR. ADDRESS DIVIDE ITS LOWEST 10 BITS
                ;BY 8
                ;
7D                      LD      A,L
CB1C                    RR      H
1F                      RRA
CB1C                    RR      H
1F                      RRA
CB3F                    SRL     A
                ;
                ;PUT RESULT IN DE
                ;
5F                      LD      E,A
1600                    LD      D,0
7D                      LD      A,L
                ;
                ;ADD BASE ADDRESS OF TABLE
                ;
210000                  LD      HL,ORMAP
19                      ADD     HL,DE
                ;
```



```
                ;ROTATE A MASK UNTIL THE '1' IS OVER THE
                ;REQUIRED BIT
E607                    AND     7
47                      LD      B,A
04                      INC     B
3E01                    LD      A,1
0F              NXTROT  RRCA
10FD                    DJNZ    NXTROT
                ;
                ;PUT THE MASK IN C
                ;
4F                      LD      C,A
                ;
                ;TEST IS THIS CELL ALREADY OCCUPIED
                ;
A6                      AND     (HL)
110000                  LD      DE,ATT
1A                      LD      A,(DE)
                ;
                ;IF NOT THEN SELECT OVER-PRINT
                ;
CBBF                    RES     7,A
2802                    JR      Z,NOTOR
                ;
                ;OTHERWISE SELECT OR-PRINT
                ;
CBFF                    SET     7,A
12              NOTOR   LD      (DE),A
                ;
                ;NOW SET THE BIT IN THE OR-MAP
                ;TO SIGNIFY "CELL USED"
                ;
79                      LD      A,C
B6                      OR      (HL)
77                      LD      (HL),A
C9                      RET
```

CHAPTER 12
Perfectly Flickerless Sprite Pixel-Animation

We are now ready, at last, to begin development of the sprite generation, printing and control routines for the production of flickerless pixel graphics in conjunction with the interrupt-driven print processor.

As you will recall, we define a sprite to be some image contained in a movable object block (hence their other name, MOBs) of adjacent characters on the screen. This block will always be rectangular, and the image may be anything from 1 × 1 character upwards in size.

The most obvious approach to moving a sprite from one position to another is to 'blank out' the 'old' image, by printing spaces over it, and then to print the new image in the new position. If the two images are mutually exclusive (i.e. they do not overlap), then this is a perfectly acceptable technique.

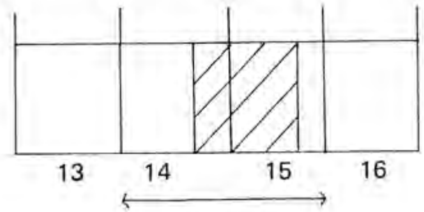
If, however, as is more usually the case, we have only moved the sprite by a few pixels, then there seems little point in printing a space in a cell common to both images, only to replace it with part of the new image almost instantly.

To avoid this time wasting, we will use a more subtle approach to sprite animation. Each shape will be surrounded by a narrow region of blank pixels, so that as we move from one position to the next, these trailing blanks will wipe out any part of the old image not already obliterated by the printing of the new image.

This way, animation will be achieved in just one print operation rather than the two required by the former technique, and the number of characters we need to print will be halved. This is a distinct advantage when you bear in mind that the standard print-processor can only print 40 characters per TV frame.

For the purposes of the rest of this discussion I shall be referring to a one character shape being moved in steps of one pixel.

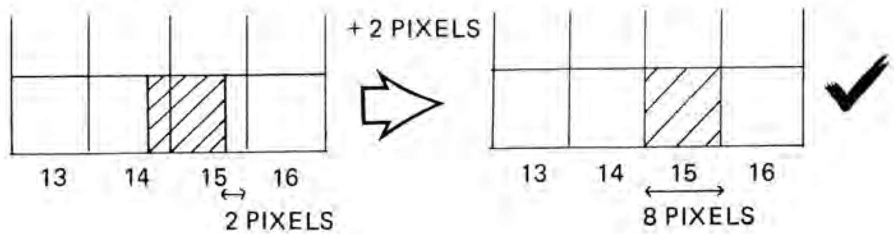
If our shape is containable within (m) columns (i.e. is less than or equal to (m × 8) pixels wide) then we see that it can, at most, occupy (m + 1) separate columns. Hence the sprite must be at least (m + 1) columns in width. For example, our 1 × 1 shape may occupy columns 14 and 15 in the following way:

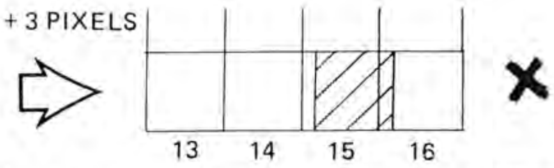


2-column
sprite required

If, in addition, we place the restriction that the shape can never occupy more than (m + 1) different columns in moving from one position to the next, then we see that both the 'old' and 'new' images can be contained in an area (m + 1) columns wide.

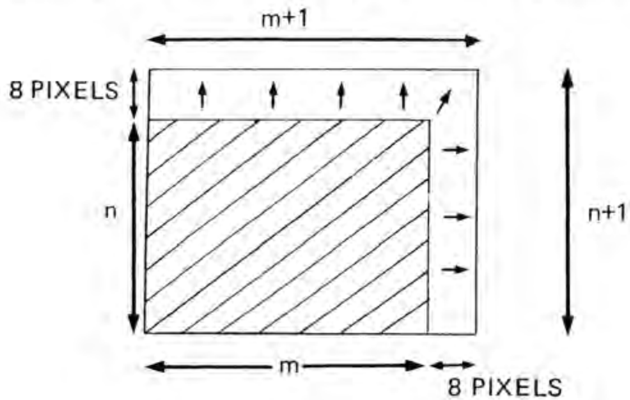
For example, we must restrict the motion of our 1 × 1 shape so that if the old image occupies column 14, the new one doesn't occupy column 16. Hence if our shape in columns 14 and 15 has two blank pixel columns to its right in column 15, then we must not allow it to move more than two pixels to the right:



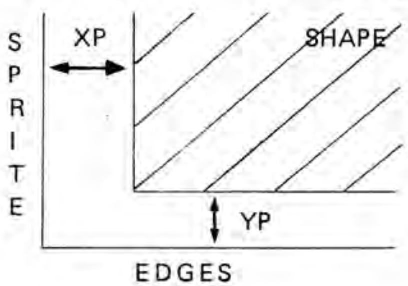


This restriction is, in practice, easy to apply, and leads to the result that we can animate a shape contained in (m × n) characters by continually printing a set of sprite images of fixed dimensions (m + 1) by (n + 1). Thus to move our 1 × 1 shape about the screen we will need a 2 × 2 cell sprite.

Now if you can imagine our m × n shape floating around within its (m + 1) × (n + 1) sprite, you will see that by virtue of the eight different horizontal positions and eight different vertical positions the shape can take, the resulting sprite will be any one of 8 × 8 = 64 possible images.



If the shape is movable at one pixel at a time in each of the X and Y directions, then each of these 64 patterns would at some time need to be printed. Before proceeding any further with this discussion, it would be prudent to define some variables. We will call the horizontal distance (in pixels) of the shape from the left hand edge of its sprite XP, and the vertical distance of the shape from the bottom edge of the sprite YP, thus:



We now have a direct choice over the method of generation of the sprite patterns. We may either store just one of the sprite 'images' in memory, and manipulate the shape within it bit-wise to obtain the required image for printing, or we may store the images in a somewhat larger table in RAM, using an indexing technique to 'pluck out' the required image without further manipulation.

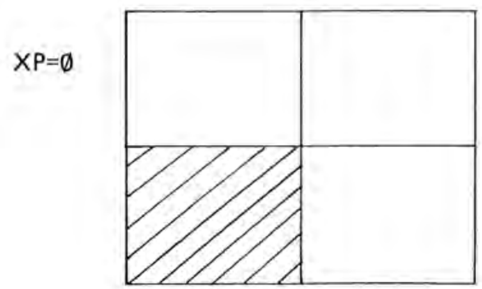
Experience has shown that, while the former technique uses as little as one eighth of the amount of memory to store the image, and is thus useful if RAM is at a premium, it is very time consuming to perform the XP bit-shifting operations required on each of the (M + 1) × N × 8 bytes affected, and it is thus preferable, wherever possible, to employ the latter technique.

Although the sprite may be one of 64 possible patterns, the situation is not as grave as it seems. We do not need to store 64 different images in memory, as it is possible to produce the eight images corresponding to the different values of YP from the one image corresponding to a given XP. Thus we only need to store (at most) eight images, one for each value of XP.

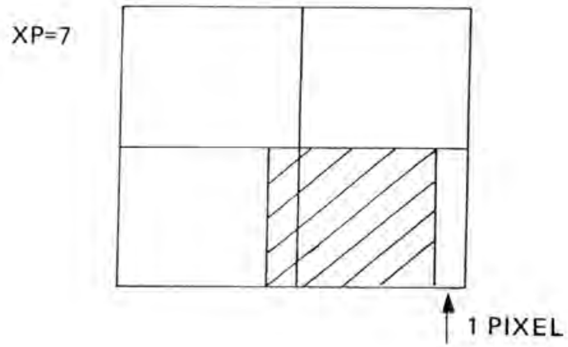
We will store the images one at a time, and each image will be stored one column at a time, working from top to bottom and left to right. Thus the order of storage for an image of our 1 × 1 shape in its 2 × 2 sprite will be:

Ø	2
1	3

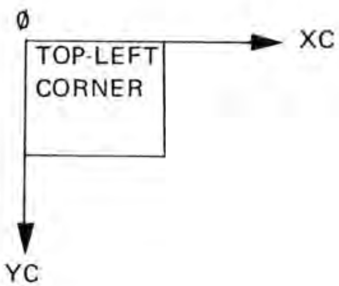
Each image will be stored as though YP = Ø, that is to say the shape will be against the bottom edge of the sprite and the top line of the image will be blank. The images are stored in increasing order of XP, so for our 1 × 1 shape the first image will be:



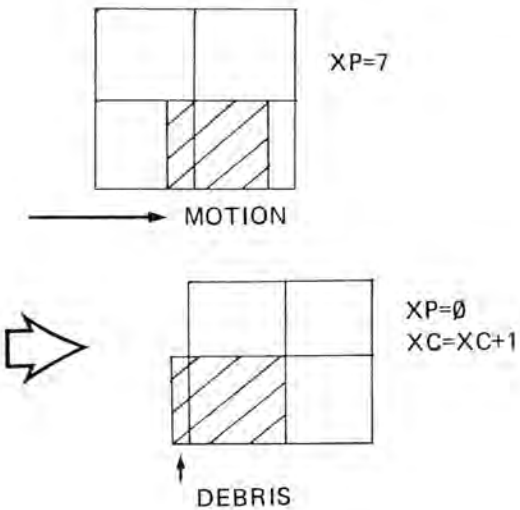
. . . and the last will be:



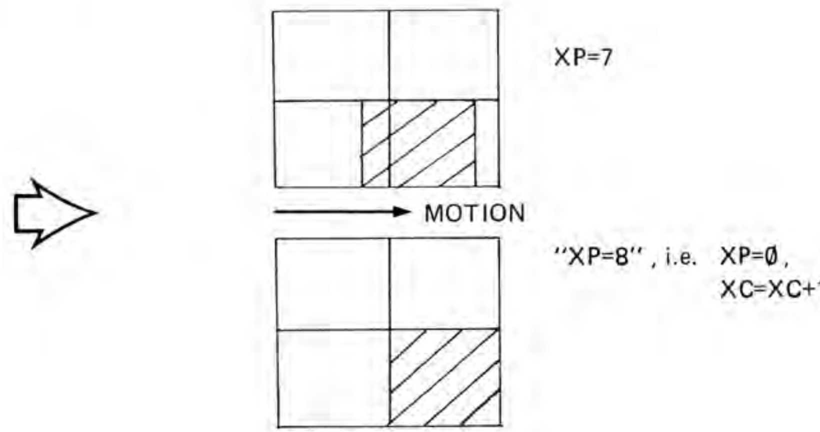
As our shape glides across the screen, the animation routines will simply be cycling through the sequence of images as XP cycles through Ø to 7. I will now define the screen position of the sprite in terms of the position of the top left hand corner of the image, (XC, YC), where XC is measured rightwards from zero, YC is measured downwards from zero, and the C stands for cell co-ordinates rather than the P for pixels. Thus we have:



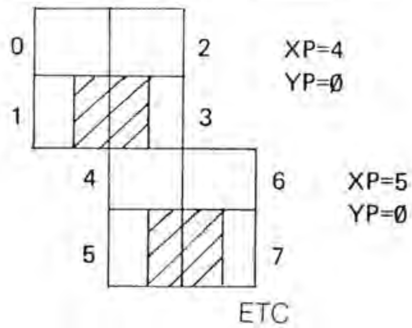
If the shape is moving leftwards, then we will be cycling backwards through the images. When XP reaches Ø, the rightmost column will be completely blank and we will be able to decrement XC and change XP to 7 without leaving any trace of the old image in its rightmost column, which is outside the new sprite area. If however, we are moving rightwards, then it will not be sufficient just to switch from image 7 to image Ø, incrementing XC, as this would leave a pixel-column of the old image in the column immediately to the left of the new one, thus:



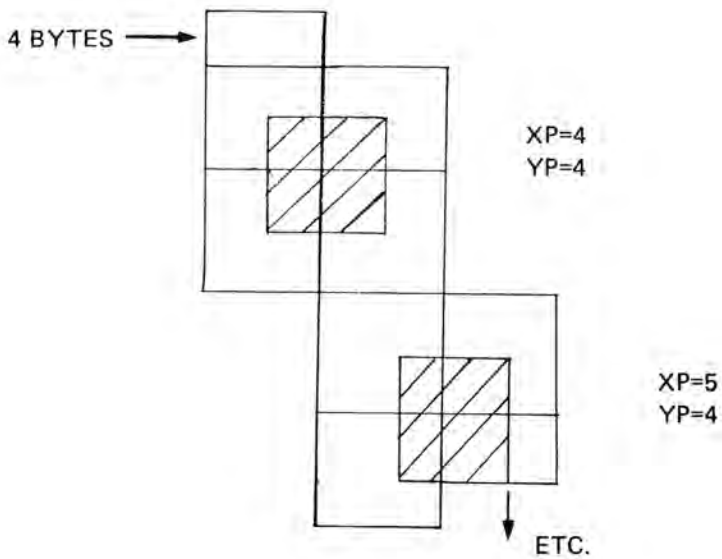
To get around this problem we need to simulate a 'ninth' image, and this will be done by including an extra column of blank cells immediately BEFORE image Ø. We will then move from XP = 7, by pointing the sprite generator at this column of blanks and pretending that XP = 8. As far as the rest of the routines are concerned, XP will be Ø and XC will be incremented. We will then have:



The various images corresponding to each value of YP will be produced by pointing the sprite generator at the YPth row of the XPth image, counting downwards from row zero. The reason for storing the images column by column can now be seen. Let us look at the memory layout of (say) image 4 of our 1 × 1 shape. We find that after the bottom left corner of this image comes the top right corner, then the bottom right corner, and then the top left corner of the next image. Thus:



By pointing at row 4 of cell Ø, we effectively shift up all the bytes in the image by four rows, and the result is a centralised shape with XP = 4, YP = 4:

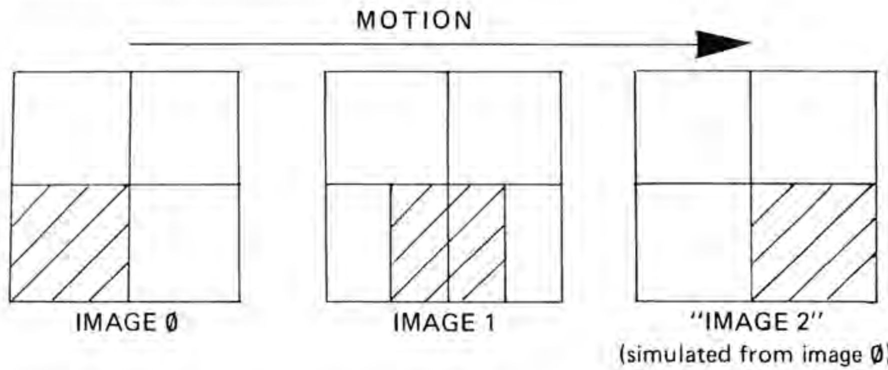


In a similar fashion to XP = 8, we will simulate YP = 8 by pointing the sprite generator at row Ø of cell 1 when moving upwards from YP = 7. Notice that YP and YC are increasing in opposite directions, so when YP reaches 7 we then let YP = Ø and DECREMENT YC.

I should point out at this stage that it is by no means essential to store eight separate images if we do not require movement in the X direction at one pixel per cycle. After all, there is little point in storing eight images if we are only moving in two-pixel steps, since there would only be four attainable values of XP, and hence only four of the images would ever be used. It emerges that we have a choice between storing 8,4,2 and 1 separate images.

With eight images, any horizontal speed up to eight pixels per movement is possible. With four images, we have a step of two pixels between images and thus speeds of Ø, 2, 4, 6 and 8 pixels are allowed, but XP must always be even. With two images, we have a step of four pixels between images, and thus a speed of Ø, 4 or 8 pixels per movement is possible, with XP being a multiple of four.

For example, if we represent our 1 × 1 shape in two separate images (bearing in mind that the shape is always at the bottom left corner of image Ø), then we have the sequence:



Obviously with one image we just have the simple case of movement by one character at a time.

It is now possible to calculate the amount of memory needed to store the images of any one sprite. Take an (m × n) character shape, and enclose it in an (m +1) × (n + 1) character sprite. Producing (a) images of this sprite, and bearing in mind that each cell requires eight bytes and each set of images requires a preceding blank column, we have:

memory needed = 8 × a × (m + 1) × (n + 1) + 8 × (n + 1)
= 8 (a (m + 1) + 1) (n + 1) bytes.

Thus for a shape three columns wide by two lines deep, defined in four images, we have m = 3, n = 2, a = 4 and:

memory needed = 8 (4 (3 + 1) + 1) (2 + 1)
= 4Ø8 bytes.

In addition to this, and assuming that all the images for the sprites currently in use are stored consecutively in memory, we must include eight zero bytes after the last image of the last sprite, to allow for the memory 'take-up' when YP = 8 and the last image is being used. In this case, those eight bytes will represent the bottom right corner of the sprite.

As an example, suppose we have two sprites in use, both of the 3 × 2 shape in the previous memory calculation. If one is a plane, the other a train, then a suitable memory-reserving sequence might be:

```
PLNSPC  DEFS    4Ø8
TRNSPC  DEFS    4Ø8
        DEFW    Ø,Ø,Ø,Ø          ;8 BYTES
```

(Using labels suffixed SPC for SPaCe).

Let us now discuss the method of controlling and keeping track of the position of the sprites. For each sprite we will use a seventeen-byte table of information which we shall call the 'sprite motion data'. This table will tell our animation routine at what speed to move the sprite along X and Y, the whereabouts of the sprite at any time, the location of the image data, the dimensions of the sprite, the colour it is to be printed in and so on. Whenever we want to move a sprite, we will point the IY index register at the start of its motion data and then call the animation routine, which will do all of the rest of the work for us, referring to the table of motion data.

Before proceeding to a complete breakdown of this table, let us first redefine XP to be the number of the image currently being used by the sprite generator. Thus if there are four images, XP will now cycle continuously through the values (Ø, 1, 2, 3) as the sprite moves across the screen. That is to say, that XP is continually incremented and then reduced modulo < number of different images >.

When there are eight images, then XP will have the same value as before, that is the number of pixels from the shape to the left hand edge

of the sprite. Otherwise, you will need to multiply XP by the step between images (2, 4 or 8 pixels) to find this distance. This conversion is worth bearing in mind when you are writing collision-detection routines and such like.

Here then is a list of the seventeen bytes of motion data for each sprite, followed by some elaboratory notes.

Address	Contents
IY	XP = Current image number (< 8)
IY+1	VX = Rate of change of XP (positive or negative)
IY+2	N = Number of images = (max. value of XP)+1
IY+3	XC = position of leftmost sprite column
IY+4	YP (Ø-7)
IY+5	VY = Rate of change of YP (positive or negative)
IY+6	YC = position of uppermost sprite line
IY+7	LO } Address of row Ø, cell Ø of image Ø
IY+8	HI
IY+9	Cycle count (see notes)
IY+1Ø	Cycle period (see notes)
IY+11	Width of expanded sprite
IY+12	Depth of expanded sprite
IY+13	LO } Length of one image = width x depth x 8
IY+14	HI
IY+15	Attribute byte and flag for OR-printing
IY+16	Attribute mask

The 'cycle count' and 'cycle period' of (IY + 9) and (IY + 1Ø) will be used to increase the versatility of our sprite control routine. Whenever the routine is called, the cycle count will be decremented. If the count is not zero then an immediate return will be made. Otherwise, the cycle count will be reloaded with the constant 'cycle period', which controls indirectly the frequency of movement of the sprite, and the sprite will be moved and printed. More about this later.

In view of the amount of memory needed to store the images of a fully operational sprite, it would be wise of us to store the bit-patterns of the various shapes a program requires in as compact a form as possible, and then to expand them to the full blown sprite images as and when required. We will need some utility routines to do this, and I have chosen to provide a two-stage sprite expansion system.

The first routine, called PADOUT, will copy the dormant shape from its storage area to the 'sprite image area', adding a column of blanks to the right, a line of blanks above it, and the statutory preceding blank column to image Ø. The second routine, SPREX, will manipulate a copy of image Ø, row by row, using shifting and rotating operations to generate the other images.

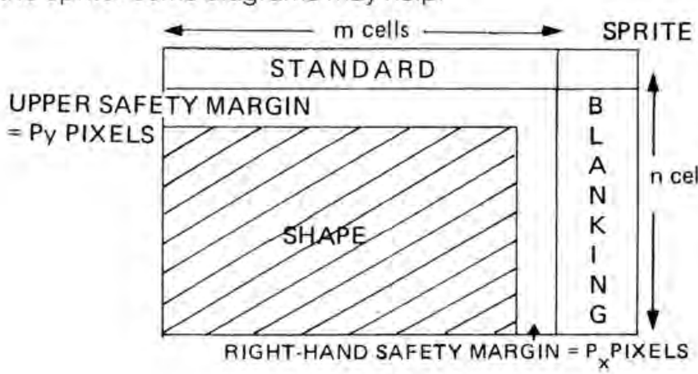
The 'bare' sprite data should be stored one column at a time, one row at a time, working from left to right and top to bottom, in the same manner in which the images are stored.

Referring to the earlier part of this chapter, you will recall that the motion of the (m) column wide shape must be restricted so that it does not occupy more than (m + 1) columns in moving from one position to the next. At the time, I dismissed such a restriction as 'easy to apply'. Now is the time to explain how to do so.

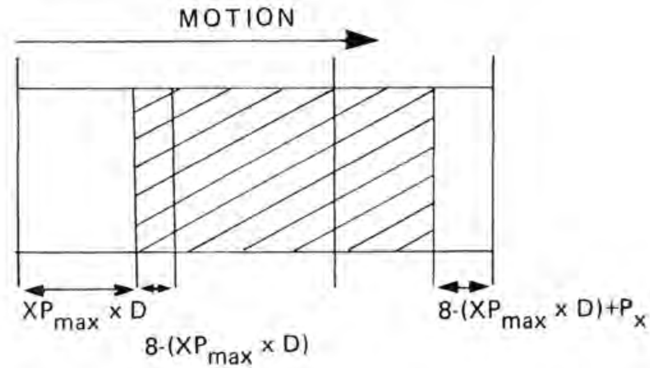
The shape must include as its top and right hand edges an L-shaped region of blanking between Ø and 7 pixels in width. The width of this safety region is determined as follows.

Suppose that the highest valued image used by the sprite is that corresponding to XP_{max}, and that the absolute rate of change of XP is VX per movement. Let the step in pixels between the shape's positions in successive sprite images be D. Then the number of pixels moved each time is D × VX. We see that the distance of the shape from the left hand edge of the sprite is (XP × D), and hence at minimum there are (8 − (XP_{max} × D)) pixels of the shape in the leftmost column.

If the right hand safety margin in sprite is P_x pixels wide, then at minimum there will be (8 − (XP_{max} × D) + P_x) blank pixels in the rightmost column of the sprite. Some diagrams may help:



Above we see how the shape must 'shrink' to allow a blank margin to fit around it. When the XP_{max} image is being used we have



. . . and we see that, if our rule that the m-column shape does not occupy more than m + 1 columns during motion is to be obeyed, then we can only allow (8 − (XP_{max} × D) + P_x) pixels of movement to the right. Now the distance moved to the right is D × VX, hence

$$8 - (XP_{max} \times D) + P_x = D \times VX$$
$$\Rightarrow \quad P_x + 8 = D\,(VX + XP_{max})$$

and finally:

$$\boxed{P_x = D\,(VX + XP_{max}) - 8.}$$

After all that theoretical strain I think a practical example would help to clarify the situation.

Suppose that we are animating a car, which at one time or other will be moving in one pixel steps, but at present is moving two pixels at a time. We will thus need a full set of eight images, which means the 'step between images' is one pixel, i.e.

D=1

to move the car in two pixel steps, we have

$VX = \frac{2}{D} = \frac{2}{1} = 2.$

Now at this constant speed we will be cycling either through the 'odd' images, where

$XP = \{1, 3, 5, 7\}$ giving $XP_{max} = 7$

or through the 'even' images, where

$XP = \{\emptyset, 2, 4, 6\}$ giving $XP_{max} = 6$

Thus we have for the 'odd' cycle,

$P_x = 1\,(2 + 7) - 8 = 1$ pixel

and for the even cycle,

$P_x = 1\,(2 + 6) - 8 = \emptyset$ pixels

This gives us the significant result that if we can restrict XP to multiples of VX then no right hand margin is required, but if we are forced to use the images for odd values of XP then the right most pixel column of the shape must be blank. Hence the car shape must include a blank right hand safety margin of one pixel in width.

A similar analysis is applicable to determine the necessary width of the upper safety margin; I shall not therefor repeat all the gory detail. Taking the absolute vertical speed VY (Ø-8 pixels per movement), and the maximum value of YP, YP_{max} (always less than eight), we find that the thickness of the upper margin, P_y pixels, is given by

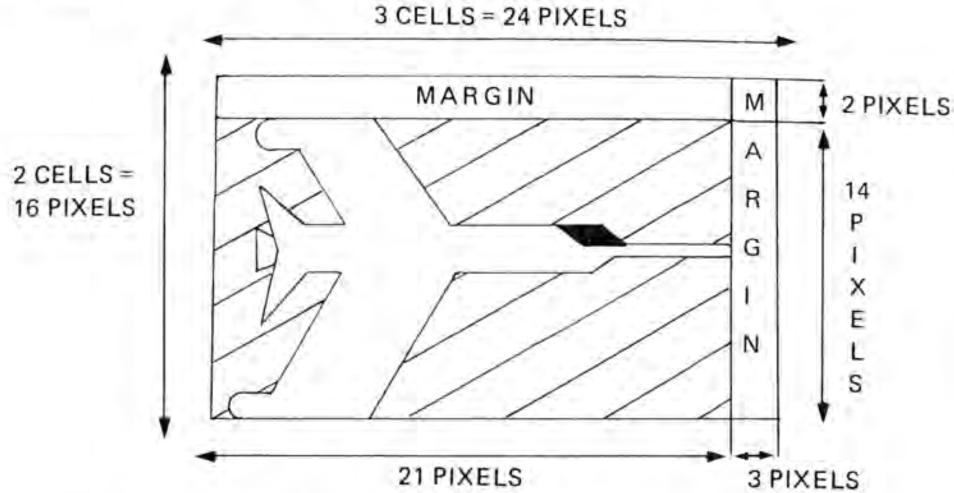
$$P_y = VY + YP_{max} - 8$$

Suppose, as a further example, that we wish to design, within a 4 × 3 sprite and hence a 3 × 2 shape, a fighter plane capable of moving at up to four pixels per movement in the X direction and up to three pixels in the Y direction. How much of the 3 × 2 shape are we free to design in?

Since speeds may be as low as one pixel per frame, we'll need eight images again, so step D = 1 pixel. At maximum, we have VX = 4 and VY = 3. It is quite possible that at some time or other we may reach XP = 7 and YP = 7, the maximum possible values. Hence $XP_{max} = 7$, and $YP_{max} = 7$ (if you are ever in any doubt, then use the maximum available values in calculations, that is YP = 7 and XP = [number of images] – 1). This gives us

$P_x = D\,(VX + XP_{max}) - 8 = 1\,(4 + 7) - 8 = 3$
and
$P_y = VY + YP_{max} - 8 = 3 + 7 - 8 = 2.$

So we need a two-pixel upper margin and a three-pixel right hand margin. Hence the area we have left to design in is (3 × 8) – 3 = 21 pixels wide, and (2 × 8) – 2 = 14 pixels deep, thus:



I will now provide the previously mentioned PADOUT routine which expands the 'standard sprite' or 'shape' data, stored in its compact form, to an 'expanded sprite', which is formed as image Ø in our previously reserved sprite image area.

Taking note of the entry requirements from the assembly listing, we see that for the 3 × 2 shape in the above example, a suitable 'run-up' to calling PADOUT would be as follows:

```
PLNSPC  DEFS    4Ø8             ;IMAGE AREA
        DEFW    Ø,Ø,Ø,Ø
PLNDAT  DEFB    (SPRITE DATA);(3*2)*8=48 BYTES OF DATA
PLNMTN  DEFS    17              ;SEE LATER FOR DETAILS
                                ;ON HOW TO INITIALIZE THE
                                ;MOTION DATA
        LD      BC,Ø3Ø2H        ;WIDTH, DEPTH
        LD      DE,PLNDAT
        LD      HL,PLNSPC
        LD      IY,PLNMTN
        CALL    PADOUT
```

Notice that we call PADOUT with IY pointing at the motion data for our shape. This is because the routine initialises the values(IY + 7), (IY + 8), (IY + 11), (IY + 12), (IY + 13) and (IY + 14) (see previous table for details of these). O.K: get typing!

```
         ;TO "PAD OUT" BARE SPRITE DATA AND PRODUCE ENTRIES
         ;IN THE SPRITE MOTION DATA
         ;
         ;
         ;ENTRY: DE=SPRITE DATA ADDRESS
         ;       HL=IMAGE STORAGE AREA
         ;       B=COLUMN WIDTH OF STANDARD SPRITE
         ;       C=LINE DEPTH OF STANDARD SPRITE
         ;       IY=SPRITE MOTION DATA ADDRESS
         ;EXIT:  HL=ADDRESS OF IMAGE Ø
         ;       B=COLUMN WIDTH OF EXPANDED SPRITE
         ;       C=ROW DEPTH OF EXPANDED SPRITE
         ;NOTE! HL AND BC WILL BE USED BY "SPREX"
         ;NOTE AF' DESTROYED
         ;
         ;CALCULATE NO. OF ROWS IN EXPANDED SPRITE
         ;
ØC       PADOUT  INC     C
FD71ØC           LD      (IY+12),C
CB21             SLA     C
CB21             SLA     C
CB21             SLA     C
         ;
         ;STORE BC FOR SPREX
         ;
78               LD      A,B
Ø4               INC     B
FD7ØØB           LD      (IY+11),B
C5               PUSH    BC
         ;
         ;A COUNTS THE COLUMNS
         ;
Ø8               EX      AF,AF'
         ;
         ;STORE DATA ADDRESS
         ;
D5               PUSH    DE
         ;
         ;START WITH A COLUMN OF BLANKS
         ;
Ø6ØØ             LD      B,Ø
C5               PUSH    BC
```



```
7Ø               LD      (HL),B
ØD               DEC     C
CD58ØØ           CALL    CL
C1               POP     BC
79               LD      A,C
D6Ø8             SUB     8
4F               LD      C,A
         ;
         ;BC=NO. OF ROWS IN STANDARD COLUMN
         ;
E1               POP     HL
         ;
         ;STORE START OF IMAGE Ø
         ;
D5               PUSH    DE
C5               PUSH    BC
Ø8               EX      AF,AF'
         ;
         ;INSERT A SPACE ON THE TOP LINE
         ;
E5       NXSCOL  PUSH    HL
EB               EX      DE,HL
36ØØ             LD      (HL),Ø
ØEØ7             LD      C,7
CD58ØØ           CALL    CL
E1               POP     HL
         ;
         ;FILL THE REST OF THE COLUMN WITH SPRITE DATA
         ;
C1               POP     BC
C5               PUSH    BC
EDBØ             LDIR
         ;
         ;DO NEXT STANDARD COLUMN
         ;
3D               DEC     A
2ØEF             JR      NZ,NXSCOL
         ;
         ;EXPAND WITH A RIGHTMOST BLANK COLUMN
         ;
C1               POP     BC
79               LD      A,C
```

```
C6Ø7              ADD     A,7
4F                LD      C,A
EB                EX      DE,HL
7Ø                LD      (HL),B
CD58ØØ            CALL    CL
           ;
           ;RETRIEVE ADDRESS OF IMAGE Ø
           ;
E1                POP     HL
           ;
           ;AND VALUE IN DE, FOR SPREX
           ;
D1                POP     DE
D5                PUSH    DE
E5                PUSH    HL
           ;
           ;CALCULATE # OF BYTES IN ONE IMAGE AND STORE
           ;IT IN SPRITE MOTION DATA
           ;
6Ø                LD      H,B
68                LD      L,B
42                LD      B,D
54                LD      D,H
19         MUL1   ADD     HL,DE
1ØFD              DJNZ    MUL1
FD75ØD            LD      (IY+13),L
FD74ØE            LD      (IY+14),H
E1                POP     HL
C1                POP     BC
           ;PUT IMAGE Ø LOCATION IN SPRITE MOTION DATA
           ;
FD75Ø7            LD      (IY+7),L
FD74Ø8            LD      (IY+8),H
C9                RET
           ;
           ;CLEARING SUBROUTINE
           ;
54         CL     LD      D,H
5D                LD      E,L
13                INC     DE
EDBØ              LDIR
C9                RET
```


Now that we have taken our bare, unexpanded sprite data from memory and created image Ø from it, we need to generate the other images. Each successive image is formed by shifting the rows of the previous one by one or more bits to the right. The routine SPREX does this by taking each row of image Ø in turn, copying it into an area of 'workspace' and then repeatedly shifting it and copying it into the appropriate position for each of the other images. We will label the start of the workspace as WKSPC and note that since we only need to place one row of a sprite in it at a time, twenty bytes should be ample. Hence you should start your program with a line like:

```
WKSPC   DEFS    2Ø
```

Notice that SPREX needs to be called with IY pointing at the motion data for your sprite, since it sets the number of images in (IY + 2). The entry values of HL and BC are already set up for you by calling PADOUT, so the only parameter you have to set after calling PADOUT is the step (in pixels) between images, stored in D. We therefore append to our previous fragment for setting up the aeroplane sprites, the lines:

```
LD      D,1                 ;FORM 8 IMAGES
CALL    SPREX
```

Beware that nearly all of the alternate register set is used by the routine, so if you are intending to come back to BASIC after using SPREX, be sure to preserve HL' with

```
EXX
PUSH    HL
EXX
```

AND

```
EXX
POP     HL
EXX
```

at the beginning and end of your program respectively.

```
;ROUTINE TO FORM THE "SHIFTED" IMAGES OF EXPANDE
;SPRITE DATA AS PRODUCED BY "PADOUT"
;
;ENTRY: HL=ADDRESS OF IMAGE Ø
;       D=STEP BETWEEN IMAGES
;       B=WIDTH OF EXPANDED SPRITE
;       C=DEPTH OF EXPANDED SPRITE IN ROWS
;PRESERVED: BC
;N.B.! B'C'D'E'H'L' ARE DESTROYED
```

```
          ;EXIT:  DE'=ENTRY VALUE OF DE,BC'=Ø,L'=Ø
          ;
3EØ8      SPREX   LD      A,8
1EFF              LD      E,ØFFH
92        SUBDIV  SUB     D
1C                INC     E
3ØFC              JR      NC,SUBDIV
FD73Ø2            LD      (IY+2),E
1D                DEC     E
D5                PUSH    DE
C5                PUSH    BC
Ø6ØØ              LD      B,Ø
          ;
          ;BC NOWS HOLDS LENGTH OF 1 COLUMN IN BYTES
          ;
11ØØØØ            LD      DE,WKSPC
D9                EXX
E1                POP     HL
D1                POP     DE
          ;
          ;H'=WIDTH,L'=# OF ROWS
          ;D'=IMAGE STEP,E'=# OF IMAGES-1
          ;GENERATE ONE ROW OF EACH IMAGE
          ;
D5        NXROW9  PUSH    DE
44                LD      B,H
D9                EXX
          ;
          ;STORE ADDRESS OF ROW Ø OF IMAGE Ø
          ;
E5                PUSH    HL
11ØØØØ            LD      DE,WKSPC
          ;
          ;BUILD THAT ROW OF SPRITE IN WORK SPACE
          ;
D9                EXX
D9        NXBYT3  EXX
7E                LD      A,(HL)
Ø9                ADD     HL,BC
12                LD      (DE),A
13                INC     DE
D9                EXX
```



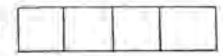
```
1ØF8              DJNZ    NXBYT3
          ;
          ;STORE ADDRESS OF CURRENT ROW OF NEXT IMAGE IN DE
          ;
D9        NXPOS   EXX
EB                EX      DE,HL
D9                EXX
          ;
          ;SHIFT ROW BY D' PIXELS
          ;
D5                PUSH    DE
4A                LD      C,D
          ;
          ;ONE PIXEL AT A TIME
          ;
7C        NXSHF   LD      A,H
D9                EXX
21ØØØØ            LD      HL,WKSPC
A7                AND     A
CB1E      NXBYT   RR      (HL)
2C                INC     L
3D                DEC     A
2ØFA              JR      NZ,NXBYT
          ;
          ;NEXT SHIFT
          ;
D9                EXX
ØD                DEC     C
2ØFØ              JR      NZ,NXSHF
          ;
          ;RETRIEVE ADDRESS OF NEXT IMAGE ROW TO HL
          ;
D9                EXX
EB                EX      DE,HL
11ØØØØ            LD      DE,WKSPC
D9                EXX
          ;
          ;TRANSFER THE ROW OF H' COLUMNS TO IMAGE AREA
          ;
44                LD      B,H
D9        NXBYT2  EXX
1A                LD      A,(DE)
```

```
77              LD      (HL),A
Ø9              ADD     HL,BC
13              INC     DE
D9              EXX
1ØF8            DJNZ    NXBYT2
          ;
          ;LOOP BACK TO GENERATE THE SAME ROW OF
          THE OTHER IMAGES
D1              POP     DE
1D              DEC     E
2ØD8            JR      NZ,NXPOS
D9              EXX
          ;
          ;FIND NEXT ROW OF IMAGE Ø
          ;
E1              POP     HL
23              INC     HL
          ;
          ;RETRIEVE DE' AND REPEAT FOR NEXT ROW
          ;
D9              EXX
D1              POP     DE
2D              DEC     L
2ØCØ            JR      NZ,NXROW9
          ;
          ;RETURN WITH THE CORRECT REGISTER SET
          ;
D9              EXX
C9              RET
```

So far, so good. By now you should have a reasonable understanding of the principles involved in this sprite animation technique, together with a pair of routines that do nearly all the preparation work for such animation. The only real task that you now need to perform whenever you wish to define a sprite, is the unavoidably tedious one of designing the shape and converting it into the original sprite data. Many people find 'character designer' programs useful, and indeed there are a number available, including the somewhat limited one-character version on the introductory 'Horizons' cassette that came with your machine.

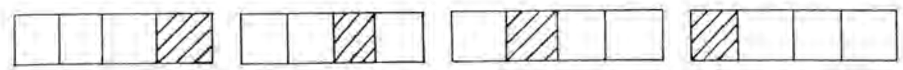
You could invest in one of these programs, or better still, write your own. Personally, I prefer the more traditional method of a pencil, an eraser, a pile of graph paper and a good supply of coffee and patience.

If you take any one row of a character and break it up into two groups of four pixels each, then each group will correspond to one digit in the hexadecimal representation of that row-byte. It is then easily seen that the four pixels will be one of only sixteen patterns, and with a little practice you will find it very easy to attach the correct digit to any given pattern. The most obvious ones are probably:

Ø 

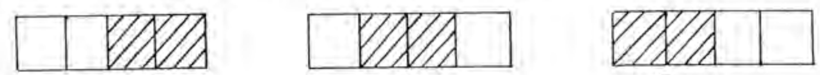
and F 

. . . Followed closely by the one-bit-set patterns.



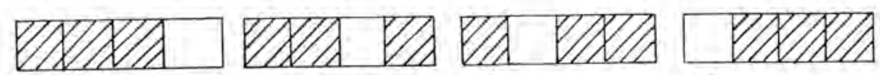
1 2 4 8

Then we have the patterns with two consecutive set bits:



3 6 C

. . . and those with just one reset bit.

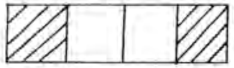


E D B 7

There are two possible patterns with alternate set and reset pixels. Distinguish them by remembering that 5 is odd and thus has the rightmost pixel-bit set:

5 A

and this just leaves the pattern for 9, which has a certain unmistakable symmetry about it:



9

If you are not already familiar with the patterns, then I hope that the above categorisation will provide you with a useful mnemonic.

I shall now provide the actual sprite 'printing' routine, SPRINT, which assimilates the correct information for each sprite and sends it to the print buffer for the print-processor.

SPRINT has been written with speed and versatility as the top priorities. If we are moving sprites once in every TV frame, in addition, perhaps, to producing sound and a low-level full screen horizon, then time is of the essence and should take priority over compactness of code and so on. You will not normally call SPRINT directly, as it will be subsidiary to a more general routine called SPRMV, which will perform various manipulations of the motion data before jumping to SPRINT.

SPRINT allows us to check for OR-printing using the OR-map system described at the end of the last chapter. A slightly modified version of ORCHK has been built into SPRINT for speed, and the option of checking for OR-printing is selected by setting bit 7 of the sprite's attribute byte, stored in (IY + 15).

Resetting this bit causes the OR-map to be ignored, and in this case over-printing will always occur. The state of this flag causes an early branch in the routine between two distinct sections, one incorporating ORCHK, the other not. It was found that this arrangement is far quicker than running a combined routine involving repeated flag tests and jumps would be.

Notice that the routine calls ATTLOC, which was listed in Chapter 1 and is used to provide the attribute address of the top left corner of the sprite.

Since the routine destroys the contents of all alternate registers, you must once again preserve HL'if you wish to return to BASIC. SPRINT makes the assumption that there is actually room for your sprite in the print buffer, and as such should not be called if there is not room, in which case you should wait for an interrupt to clear the buffer. You will notice that the sections involved in sending data to the buffer use single-register increment instructions to step through it. If you have extended the buffer beyond 42 entries in length, as described in chapter 9, then you will need to change the instructions to dual-register increments, i.e. change INC L to INC HL. Recall that the one-byte

variable CHSTRE holds the number of used entries in the buffer, and that BUFFPT points at the next free entry. Both are adjusted accordingly by the routine.

It is not desirable to receive an interrupt when only half a sprite has been sent to the buffer, so unless you are using the interrupt intercept for something other than the print processor, you should disable the interrupts before calling SPRINT, and enable them on return.

SPRINT will cope admirably with sprites that 'spill off' the edges of the screen, or even those that are not on the screen at all. For example, we may have just the right hand column of a 3 × 3 sprite on the screen by sending SPRINT the value XC = −2, or FE Hex.

As the routine stands, any part of the sprite that is in the text area will be printed. However, we can, if we wish, alter the width of this 'sprite window' by changing the operands of the instructions labelled LFTLM1, LFTLM2, RGTLM1 and RGTLM2, where LFTLM stands for 'LeFT LiMit' and RGTLM for 'RiGhT LiMit'. The left limit is the value of the leftmost column in the sprite window, while the right limit is the value of the column immediately to the right of the window (32 in the case of a maximum window).

For example, suppose that we want the sprite window to be over the central 2Ø columns of the screen (we may be using the outer ones for scoring, say). Then the left limit will be column 6, and the right limit will be column 6 + 2Ø = 26. So we use:

```
LD      A,6
LD      (LFTLM1+1),A
LD      (LFTLM2+1),A
LD      A,26
LD      (RGTLM1+1),A
LD      (RGTLM2+1),A
```

That's about all there is to say regarding this formidable listing, so I'll leave you to type it in and peruse the coding.

```
        ;THIS ROUTINE SENDS SPRITE DATA TO THE PRINT BUFFER
        ;ENTRY:  B=XP,C=YP,D=YC,E=XC
        ;        HL=ADDRESS OF IMAGE Ø
        ;ALL AS SET UP BY SPRMV
        ;EXIT:   DE=Ø
        ;DESTROYS: A'F'B'C'D'E'H'L'
        ;
D5      SPRINT  PUSH    DE
```

```
                ;
                ;IF XP=0 THEN LEAVE HL POINTING AT IMAGE 0
                ;
78                      LD      A,B
A7                      AND     A
2809                    JR      Z,POS0
                ;
                ;FIND CORRECT IMAGE
                ;
FD5E0D                  LD      E,(IY+13)
FD560E                  LD      D,(IY+14)
19              NXA     ADD     HL,DE
10FD                    DJNZ    NXA
                ;
                ;FIND CORRECT VERTICAL POSITION
                ;
09              POS0    ADD     HL,BC
                ;
                ;FIND LOCATION OF TOP-LEFT ATTRIBUTE OF SPRITE
                ;
C1                      POP     BC
E5                      PUSH    HL
CD0000                  CALL    ATTLOC
                ;
                ;E COUNTS THE COLUMNS REMAINING
                ;
FD5E0B                  LD      E,(IY+11)
                ;
                ;DECIDE BETWEEN OR-PRINT AND OVER-PRINT MODES
                ;BY TESTING BIT 7 OF THE SPRITE ATTRIBUTES
                ;
D9                      EXX
FD6E0F                  LD      L,(IY+15)
FD6610                  LD      H,(IY+16)
CB7D                    BIT     7,L
CBBD                    RES     7,L
D9                      EXX
CAD800                  JP      Z,SPRTNO
                ;
                ;OR-PRINT IS SELECTED. FIND APPROPRIATE ADDRESS IN
                ;OR-MAP
                ;
```



```
78                      LD      A,B
87                      ADD     A,A
87                      ADD     A,A
47                      LD      B,A
79                      LD      A,C
CB2F                    SRA     A
CB2F                    SRA     A
CB2F                    SRA     A
80                      ADD     A,B
                ;
                ;ADD BASE ADDRESS OF OR-MAP
                ;
D9                      EXX
EB                      EX      DE,HL
4F                      ADD     C,A
0600                    LD      B,0
210000                  LD      HL,ORMAP
09                      LD      HL,BC
3A0000                  LD      A,(CHSTRE)
47                      LD      B,A
D9                      EXX
                ;
                ;HL' HOLDS LOCATION IN OR-MAP ROTATE MASK OVER
                ;CORRECT CELL-BIT IN THE OR-MAP
                ;
79                      LD      A,C
E607                    AND     7
47                      LD      B,A
3E80                    LD      A,80H
2803                    JR      Z,NROT1
0F              NXTRT   RRCA
10FD                    DJNZ    NXTRT
                ;
                ;STORE MASK IN C'
                ;
D9              NROT1   EXX
4F                      LD      C,A
D9                      EXX
79                      LD      A,C
                ;
```

```
        ;POINT BC AT IMAGE DATA
        ;
C1              POP     BC
E5      NXTX1   PUSH    HL
        ;
        ;STORE OR-MAP ADDRESS
        ;
D9              EXX
E5              PUSH    HL
D9              EXX
        ;
        ;LET D=DEPTH IN LINES
        ;
FD560C          LD      D,(IY+12)
        ;
        ;IF PRINT POSN IS OUT OF X-RANGE THEN SKIP COLUMN
        ;
FE20    RGTLM1  CP      32
3071            JR      NC,HOPCL1
FE00    LFTLM1  CP      0
386D            JR      C,HOPCL1
        ;
        ;STORE COLUMN POSN IN A'
        ;
08              EX      AF,AF'
        ;
        ;IF PRINT POSN IS BELOW TEXT AREA THEN END
        ;
7C      NXTY1   LD      A,H
FE5B            CP      91
3068            JR      NC,OUT81
        ;
        ;IF PRINT POSN IS ABOVE TEXT AREA THEN MISS THIS
        ;LINE OF SPRITE
        ;
FE58            CP      88
D5              PUSH    DE
EB              EX      DE,HL
382F            JR      C,NPR1
        ;
        ;DECIDE WHETHER OR-PRINT ON THIS CELL IS NEEDED
        ;
```



```
D9              EXX
79              LD      A,C
A6              AND     (HL)
CBBB            RES     7,E
2802            JR      Z,NOTOR3
        ;
        ;IF CELL IS OCCUPIED THEN SET FLAG FOR OR-PRINT
        ;
CBFB            SET     7,E
        ;
        ;SIGNIFY CELL OCCUPIED
        ;
79      NOTOR3  LD      A,C
B6              OR      (HL)
77              LD      (HL),A
D9              EXX
        ;
        ;SEND CHARACTER TO BUFFER
        ;
1A              LD      A,(DE)
D9              EXX
AB              XOR     E
A2              AND     D
AB              XOR     E
04              INC     B
D9              EXX
2A0000          LD      HL,(BUFFPT)
07              RLCA
77              LD      (HL),A
2C              INC     L
73              LD      (HL),E
2C              INC     L
72              LD      (HL),D
2C              INC     L
7A              LD      A,D
E603            AND     3
07              RLCA
07              RLCA
07              RLCA
F640            OR      64
77              LD      (HL),A
2C              INC     L
```

```
71              LD      (HL),C
2C              INC     L
70              LD      (HL),B
2C              INC     L
220000          LD      (BUFFPT),HL
        ;
        ;INCREASE DATA POINTER TO NEXT CELL OF IMAGE
        ;
210800  NPR1    LD      HL,8
09              ADD     HL,BC
44              LD      B,H
4D              LD      C,L
EB              EX      DE,HL
D1              POP     DE
        ;
        ;IF LINE-COUNT IS ZERO THEN NEXT COLUMN
        ;
15              DEC     D
2810            JR      Z,IN16
        ;
        ;OTHERWISE MOVE OR-MAP POINTER TO NEXT LINE
        ;
D9              EXX
7D              LD      A,L
C604            ADD     A,4
6F              LD      L,A
D9              EXX
        ;
        ;AND MOVE ATTRIBUTE POINTER TO NEXT LINE
        ;
7D              LD      A,L
C620            ADD     A,32
6F              LD      L,A
30AF            JR      NC,NXTY1
24              INC     H
        ;
        ;LOOP BACK FOR NEXT LINE OF SPRITE
        ;
C35E00          JP      NXTY1
        ;
        ;INCREASE COLUMN POSITION
        ;
```



```
08      IN16    EX      AF,AF'
3C              INC     A
        ;
        ;FETCH OR-MAP ADDRESS AND MOVE MASK TO
        ;INCREMENTING THE POINTER IF NECESSARY
        ;
D9              EXX
E1              POP     HL
CB09            RRC     C
3001            JR      NC,NINC2
2C              INC     L
D9      NINC2   EXX
        ;
        ;POINT HL AT FIRST ATTRIBUTE OF NEXT COLUMN
        ;
E1              POP     HL
2C              INC     L
        ;
        ;LOOP BACK FOR NEXT COLUMN
        ;
1D              DEC     E
C24E00          JP      NZ,NXTX1
        ;
        ;SET NEW VALUE OF CHSTRE
        ;
D9              EXX
78              LD      A,B
320000          LD      (CHSTRE),A
D9              EXX
C9              RET
        ;
        ;JUMPS TO HERE TO OMIT ALL OR PART OF A COLUMN
        ;
08      HOPCL1  EX      AF,AF'
        ;
        ;MOVE IMAGE POINTER TO THE NEXT SPRITE COLUMN
        ;
60      OUT81   LD      H,B
69              LD      L,C
010800          LD      BC,8
09      NXT81   ADD     HL,BC
15              DEC     D
```

```
20FC               JR      NZ,NXT81
44                 LD      B,H
4D                 LD      C,L
        ;
        ;JUMP BACK INTO MAIN ROUTINE
        ;
18DB               JR      IN16
        ;
        ;THE MUCH SHORTER AND FASTER OVER-PRINTING SECTION
        ;
79      SPRTNO     LD      A,C
C1                 POP     BC
        ;
        ;HOLD CHSTRE IN E
        ;
D9                 EXX
ED5B0000           LD      DE,(CHSTRE)
D9                 EXX
E5      NXTX2      PUSH    HL
        ;
        ;LET D=DEPTH IN LINES
        ;
FD560C             LD      D,(IY+12)
        ;
        ;IF PRINT POSN IS OUT OF X-RANGE THEN SKIP COLUMN
        ;
FE20    RGTLM2     CP      32
3056               JR      NC,HOPCL2
FE00    LFTLM2     CP      0
3852               JR      C,HOPCL2
        ;
        ;STORE COLUMN POSN IN A
        ;
08                 EX      AF,AF'
        ;
        ;IF PRINT POSN IS BELOW TEXT AREA THEN END
        ;
7C      NXTY2      LD      A,H
FE5B               CP      91
304D               JR      NC,OUT82
        ;
        ;IF PRINT POSN IS ABOVE TEXT AREA THEN MISS THIS
```



```
        ;LINE OF SPRITE
        ;
FE58               CP      88
D5                 PUSH    DE
EB                 EX      DE,HL
3822               JR      C,NPR2
        ;
        ;SEND CHARACTER TO BUFFER
        ;
1A                 LD      A,(DE)
D9                 EXX
AD                 XOR     L
A4                 AND     H
AD                 XOR     L
1C                 INC     E
D9                 EXX
2A0000             LD      HL,(BUFFPT)
07                 RLCA
77                 LD      (HL),A
2C                 INC     L
73                 LD      (HL),E
2C                 INC     L
72                 LD      (HL),D
2C                 INC     L
7A                 LD      A,D
E603               AND     3
07                 RLCA
07                 RLCA
07                 RLCA
F640               OR      64
77                 LD      (HL),A
2C                 INC     L
71                 LD      (HL),C
2C                 INC     L
70                 LD      (HL),B
2C                 INC     L
220000             LD      (BUFFPT),HL
        ;
        ;INCREASE DATA POINTER TO NEXT CELL OF IMAGE
        ;
210800  NPR2       LD      HL,8
09                 ADD     HL,BC
```

```
44                 LD      B,H
4D                 LD      C,L
EB                 EX      DE,HL
D1                 POP     DE
          ;
          ;IF LINE-COUNT IS ZERO THEN NEXT COLUMN
          ;
15                 DEC     D
280A               JR      Z,IN17
          ;
          ;OTHERWISE MOVE ATTRIBUTE POINTER TO NEXT LINE
          ;
7D                 LD      A,L
C620               ADD     A,32
6F                 LD      L,A
30C2               JR      NC,NXTY2
24                 INC     H
          ;
          ;LOOP BACK FOR NEXT LINE OF SPRITE
          ;
C3ED00             JP      NXTY2
          ;
          ;INCREASE COLUMN POSITION
          ;
08        IN17     EX      AF,AF'
3C                 INC     A
          ;
          ;POINT HL AT FIRST ATTRIBUTE OF NEXT COLUMN
          ;
E1                 POP     HL
2C                 INC     L
          ;
          ;LOOP BACK FOR NEXT COLUMN
          ;
1D                 DEC     E
C2E000             JP      NZ,NXTX2
D9                 EXX
7B                 LD      A,E
320000             LD      (CHSTRE),A
D9                 EXX
C9                 RET
          ;
```



```
          ;JUMPS HERE TO OMIT ALL OR PART OF COLUMN
          ;
08        HOPCL2   EX      AF,AF'
          ;
          ;MOVE IMAGE POINTER TO THE NEXT SPRITE COLUMN
          ;
60        OUT82    LD      H,B
69                 LD      L,C
010800             LD      BC,8
09        NXT82    ADD     HL,BC
15                 DEC     D
20FC               JR      NZ,NXT82
44                 LD      B,H
4D                 LD      C,L
          ;
          ;JUMP BACK INTO THE MAIN ROUTINE
          ;
18E3               JR      IN17
```

We now have the three routines necessary to prepare the data for, and actually print, our sprites on the screen. To complete the set of sprite generation routines I shall supply a master sprite controlling routine. The function of SPRMV will be to update the values of XP, XC, YP and YC according to VX and VY (all stored in the motion data, indexed by the IY register) and then to set up the correct parameters in the registers and jump to SPRINT the sprite. The only parameter required by SPRMV is the address of the motion data, in IY. Once SPRMV has been called, no further work is required to move and print your sprite.

Before explaining how to initialise and manipulate the motion data, together with providing a spectacular demonstration routine, allow me to present the listing for SPRMV.

```
          ;
          ;GENERAL PURPOSE SPRITE CONTROLLER
          ;
          ;ENTRY:  IY POINTS AT MOTION DATA SEE TEXT
          ;FOR DETAILS
          ;NOTE:B'C'D'E'H'L'A'F' DESTROYED
          ;NOTE: IY IS PRESERVED
          ;
          ;
          ;DECREMENT CYCLE COUNT
          ;
```

```
FD3509    SPRMV   DEC     (IY+9)
C0                RET     NZ
          ;
          ;IF ZERO THEN REFILL CYCLE COUNT
          ;
FD7E0A            LD      A,(IY+10)
FD7709            LD      (IY+9),A
          ;
          ;LET HL=ADDRESS OF IMAGE 0
          ;
FD6608            LD      H,(IY+8)
FD6E07            LD      L,(IY+7)
          ;
          ;ADD STEP TO XP
          ;
FD7E00            LD      A,(IY+0)
FD8601            ADD     A,(IY+1)
F22500            JP      P,NNEG1
          ;
          ;IF RESULT NEGATIVE THEN LET XP=XP+XMAX
          ;AND LET XC=XC-1
          ;
FD8602            ADD     A,(IY+2)
FD3503            DEC     (IY+3)
FD5E03            LD      E,(IY+3)
          ;
          ;JUMP TO DEAL WITH Y
          ;
C33F00            JP      XDN
FD4602    NNEG1   LD      B,(IY+2)
          ;
          ;IF XP<XMAX THEN GO FOR Y
          ;
B8                CP      B
FD5E03            LD      E,(IY+3)
3811              JR      C,XDN
          ;
          ;ELSE INCREMENT XC, LET XP=XP-XMAX
          ;
FD3403            INC     (IY+3)
90                SUB     B
          ;
```



```
          ;AND DECREASE HL BY ONE COLUMN TO
          ALLOW FOR A BLANK
          ;LEFT COLUMN
          ;
08                EX      AF,AF'
FD7E0C            LD      A,(IY+12)
01F8FF            LD      BC,0FFF8H
09        NXUB    ADD     HL,BC
3D                DEC     A
C23900            JP      NZ,NXUB
08                EX      AF,AF'
          ;
          ;STORE NEW VALUE OF XP
          ;
FD7700    XDN     LD      (IY+0),A
47                LD      B,A
          ;
          ;ADD STEP TO YP
          ;
FD7E04            LD      A,(IY+4)
FD8605            ADD     A,(IY+5)
F25800            JP      P,NNEG2
          ;
          ;IF RESULT NEGATIVE THEN LET YP=YP MOD 8
          ;
E607              AND     7
4F                LD      C,A
          ;
          ;AND INCREMENT YC
          ;
FD3406            INC     (IY+6)
FD5606            LD      D,(IY+6)
C36500            JP      YDN
          ;
          ;IF YP>7 THEN LET YP=YP-8
          ;AND DECREMENT YC
          ;
FE08      NNEG2   CP      8
FD5606            LD      D,(IY+6)
4F                LD      C,A
3805              JR      C,YDN
E607              AND     7
```

```
FD35Ø6              DEC     (IY+6)
FD77Ø4    YDN       LD      (IY+4),A
          ;
          ;JUMP TO THE SPRITE PRINTING ROUTINE
          ;
C3ØØØØ              JP      SPRINT
```

With reference to the table of motion data contents found earlier in this chapter, we recall that seven bytes of the seventeen alotted to each sprite are initialised by the routines PADOUT and SPREX.

Thus we need only reserve space for them with DEFB Ø in the assembly listing. The variables we have to initialise ourselves include the obvious position values XP, XC, YP and YC. Remember that XP is measured to the right and YP upwards from, the bottom left corner of the sprite, while XC is measured to the right of and YC downwards from, the top left corner of the screen. Recall also that (XC, YC) are the co-ordinates of the top left corner of the sprite.

The speeds VX and VY are measured in the same directions as XP and YP, and may be greater than, less than or equal to zero. If VX > Ø then movement is to the right, while if VX < Ø then it is to the left. Similarly VY > Ø signifies upward movement, while VY < Ø sends the sprite downwards. This arrangement allows high versatility in the direction of movement. For example, we could cause a sprite to make a gentle 'dive' with horizontal speed three pixels and vertical speed one pixel per movement, by setting:

VX = Ø3H VY = ØFFH (minus one)

As I explained earlier, the 'cycle count' is provided as a means for regulating the frequency at which the sprites move, and also whether two or more sprites move in phase with each other.

This is best shown by means of an example. Suppose that we have two sprites, with motion data at labels MDAT1 and MDAT2, and that we want one sprite to move every five TV frames, and the second to move once in every three frames. We set the respective 'cycle periods' to values five and three, and, as usual, initialise the 'cycle counts' to one, so that both sprites will move on the first call to SPRMV. All that we then have to do is:

```
LD      IY,MDAT1
CALL    SPRMV
LD      IY,MDAT2
CALL    SPRMV
```

after each interrupt.

If we have (say) two sprites moving at the same frequency, and we wish to keep them 'out of phase', perhaps because there isn't enough room in the print buffer to animate them both in the same TV frame, then we use different initialisations of the cycle count. For example, suppose that two sprites are given cycle periods of two calls.

Then we can make them move on alternate TV frames by setting the first cycle count to one, and the second to value two. Whatever happens, the cycle period and cycle count must always be non-zero. Since otherwise a movement frequency of once in every 256 calls to SPRMV would result.

Using the concepts of cycle count and cycle period, we can animate all the sprites involved in a program in one block. If we place all their motion data consecutively in memory, then a suitable fragment after each interrupt (detected by a HALT instruction) might be as follows:

```
MDAT      EQU     (ADDRESS OF MOTION DATA)
          LD      IY,MDAT
          LD      B,(NUMBER OF SPRITES)
NXTSPRT   PUSH    BC
          CALL    SPRMV
          POP     BC
          LD      DE,17
          ADD     IY,DE
          DJNZ    NXSPRT
```

Referring again to the earlier table of motion data contents, you will notice that the address of the first byte of image Ø (immediately after the preceeding blank column) is stored at (IY + 7). This value is also returned in HL after the call to PADOUT to set up the sprite data.

We can use this entry in the motion data as a means of switching or cycling through different sets of images for any one sprite. For example, you may wish to make your character 'walk' instead of glide, or perhaps make your spacecraft gradually disintergrate in flight, after being hit by a particularly nasty plasma bolt.

To realise this function, set up as many different sets of sprite data as you need, storing the values returned by PADOUT in your own look-up table. IY should be kept pointing at one set of motion data, which will obviously then be set up with the last set of sprite data generated.

Then when you are running your program, use an 'animation count' and 'animation period' analogous to the 'cycle count' and 'cycle period' system to step through the different sets of images, retrieving the appropriate address from your look-up table and inserting it at (IY + 7) every time you want to switch data.

There are various other manipulations of the motion data that you could try; for example, you could make the sprite move in a preprogrammed pattern by running through a table of values for VX and VY, or you could make the sprite do a 'chameleon' act by manipulating the attribute byte at (IY + 15) (remember to preserve bit 7, the OR-printing flag, though!). I'll leave further variations on this theme to your imagination, and begin development of a demonstration routine.

After a great deal of thought I elected to show you how to move two sprites in opposite directions along the horizontal centre line of the screen. One sprite will be of a special playing card, the six of clubs, known traditionally in Oxfordshire as 'Gordon's Card', while the other, to preserve variety, will be of a red telephone.

It would be a pity not to use the smoothest possible animation, so we will move both sprites by one pixel in every TV frame. Now the print buffer can take forty characters, so let's use twenty on each sprite. We know that $5 \times 4 = 2\emptyset$, so we can use a 3×4 cell shape for the card, and a 4×3 cell shape for the telephone.

Recall the formula for the image area for (a) images of an $(m \times n)$ shape, namely:

Memory needed $= 8\,(a\,(m + 1) + 1)\,(n + 1)$ bytes.

For the telephone, $m = 4$, $n = 3$, $a = 8$ and

Memory needed $= 8\,(8\,(4 + 1) + 1)\,(3 + 1)$
$= 1312$ bytes.

For the card, $m = 3$, $n = 4$, $a = 8$ and

Memory needed $= 8\,(8\,(3 + 1) + 1)\,(4 + 1)$
$= 132\emptyset$ bytes.

Not forgetting the eight zero bytes at the end of the combined image area, we reserve space with

```
TELSPC  DEFS    1312
CARSPC  DEFS    132Ø
        DEFW    Ø,Ø,Ø,Ø
```


The horizontal speed will be VX = 1, the maximum value of XP will be $XP_{max} = 7$, and the distance between two successive images will be D = 1 pixel. Hence the width of the safety margin to the right of our shapes will be:

$$P_x = D\,(VX + XP_{max}) - 8$$
$$= 1\,(1 + 7) - 8$$
$$= \emptyset \text{ pixels,}$$

that is to say that we may design our shapes in the full three or four columns! Since the sprites will not be moved vertically, no upper safety margin is necessary anyway. The phone and card have been designed in the full area allowed, and I have encoded the data for you, the results of which will be found at labels TELDAT and CARDAT respectively. Casting your mind back to the procedure for employing PADOUT and SPREX, you will see that we can generate the images of our telephone by the fragment:

```
LD      HL,TELSPC       ;IMAGE AREA
LD      DE,TELDAT       ;SPRITE DATA
LD      BC,Ø4Ø3H        ;B=WIDTH, C=HEIGHT
LD      IY,TELMTN       ;MOTION DATA
CALL    PADOUT
LD      D,1             ;STEP BETWEEN IMAGES OF
CALL    SPREX           ;ONE PIXEL
```

A similar fragment will generate the images of the playing card; where TELMTN and CARMTN are the start addresses of the motion data tables for the phone and card respectively.

We will initialise the position of the telephone to just off the left hand edge of the screen, with the base of the phone in line eleven. Hence the top left corner of the sprite is at (−4, 9) and the shape is at the bottom left corner of the sprite, i.e.

XP = Ø, XC = ØFCH, YP = Ø, YC = 9.

We shall be moving the sprites once in every TV frame, so set the cycle count and cycle period to one, I have elected to use OR-printing in this demonstration, so that the two sprites merge as they cross over each other. The telephone will be red (value 2) and we'll mask the paper from the current attribute (i.e. PAPER 8), hence we have attribute byte 82H and mask byte 38H. Plugging zero into the gaps in our table that will be filled by PADOUT and SPREX, we have the initial motion data:

```
TELMTN  DEFB    Ø,1,Ø,ØFCH,Ø,Ø,9,Ø,Ø,1,1,
                Ø,Ø,Ø,Ø,82H,38H
```

The card will start just off the right hand screen edge with its base in line twelve. Hence we start with

XP = Ø, XC = 32, YP = Ø, YC = 9.

Remembering that VX = −1 = ØFFH since the card is moving leftwards, and using cyan INK, PAPER 8 and OR-printing (bit 7 of the attribute set) we have the initial card motion data

```
CARDAT  DEFB    Ø,ØFFH,Ø,32,Ø,Ø,9,Ø,Ø,1,1,
                Ø,Ø,Ø,Ø,85H,38H
```

To operate the OR-printing function, we must make sure that the OR-map is cleared (using the CLOR routine in the last chapter) before each complete set of sprite movements is started. Otherwise, we would end up OR-printing the new image of a sprite on top of its old one, causing an undesirable trail over the screen as the sprite moves. Thus the main loop of the demonstration will include the lines

```
CALL    CLOR
LD      IY,TELMTN
CALL    SPRMV
LD      IY,CARMTN
CALL    SPRMV
```

before a HALT instruction, to get the sprites actually printed.

The rest of the demonstration listing is self explanatory. Notice that the routines INT1 and DISINT are called from Chapter 9.

Here then is the 'spectacular demonstration' routine. Study the listing carefully, and feel free to try altering the speed of the sprites, the number of images and so on.

```
                ;
                ;
                ;DEMO ROUTINE FOR PADOUT,SPREX,SPRINT
                AND SPRMV
                ;
                ;PRESERVE HL' FOR RETURN TO BASIC
                ;
D9              TEST    EXX
E5                      PUSH    HL
D9                      EXX
                ;
                ;SET ZERO-HORIZON, BLACK SKY AND SEA
```



```
                ;
AF                      XOR     A
320000                  LD      (ROWS+1),A
320000                  LD      (TOPBRD+1),A
320000                  LD      (BOTBRD+1),A
                ;
                ;GENERATE SPRITE DATA FOR TELEPHONE
                ;
214DØ1                  LD      HL,TELSPC
118CØØ                  LD      DE,TELDAT
                ;
                ;PHONE IS 4 COLUMNS BY 3 LINES
                ;
Ø1Ø3Ø4                  LD      BC,Ø4Ø3H
FD216AØØ                LD      IY,TELMTN
CDØØØØ                  CALL    PADOUT
16Ø1                    LD      D,1
CDØØØØ                  CALL    SPREX
                ;
                ;GENERATE SPRITE DATA FOR PLAYING CARD
                ;
216DØ6                  LD      HL,CARSPC
11EDØØ                  LD      DE,CARDAT
                ;
                ;CARD IS 3 COLUMNS BY 4 LINES
                ;
Ø1Ø4Ø3                  LD      BC,Ø3Ø4H
FD217BØØ                LD      IY,CARMTN
CDØØØØ                  CALL    PADOUT
16Ø1                    LD      D,1
CDØØØØ                  CALL    SPREX
                ;
                ;INITIALIZE INTERRUPT-DRIVEN PRINT-
                PROCESSOR
                ;
CDØØØØ                  CALL    INT1
76                      HALT
                ;
                ;MOVE THE SPRITES ACROSS THE SCREEN
                4 TIMES AT ONE
                ;PIXEL PER TV FRAME IN OPPOSITE DIRECTIONS
                ;
```

```
ØEØ2              LD      C,2
Ø6ØØ      NXAM2   LD      B,Ø
C5        NXAM    PUSH    BC
          ;
          ;CLEAR THE OR-MAP BEFORE EACH SET OF
          MOVEMENTS
          ;
CDØØØØ            CALL    CLOR
          ;
          ;MOVE AND PRINT PHONE
          ;
FD216AØØ          LD      IY,TELMTN
CDØØØØ            CALL    SPRMV
          ;
          ;MOVE AND PRINT CARD
          ;
FD217BØØ          LD      IY,CARMTN
CDØØØØ            CALL    SPRMV
C1                POP     BC
76                HALT
          ;
          ;NEXT FRAME
          ;
1ØEA              DJNZ    NXAM
          ;
          ;REVERSE DIRECTIONS ALONG X
          ;
FD7EØ1            LD      A,(IY+1)
FD77FØ            LD      (IY-16),A
ED44              NEG
FD77Ø1            LD      (IY+1),A
          ;
          ;NEXT PASS
          ;
ØD                DEC     C
2ØDA              JR      NZ,NXAM2
          ;
          ;RESELECT IM 1 AND RETRIEVE HL
          ;
CDØØØØ            CALL    DISINT
D9                EXX
E1                POP     HL
```



```
D9                EXX
C9                RET
          ;
          ;SPRITE MOTION DATA
          ;
ØØ        TELMTN  DEFB    Ø
Ø1                DEFB    1
ØØ                DEFB    Ø
FC                DEFB    ØFCH
ØØ                DEFB    Ø
ØØ                DEFB    Ø
Ø9                DEFB    9
ØØ                DEFB    Ø
ØØ                DEFB    Ø
Ø1                DEFB    1
Ø1                DEFB    1
ØØ                DEFB    Ø
ØØ                DEFB    Ø
ØØ                DEFB    Ø
ØØ                DEFB    Ø
82                DEFB    82H
38                DEFB    38H
ØØ        CARMTN  DEFB    Ø
FF                DEFB    ØFFH
ØØ                DEFB    Ø
2Ø                DEFB    32
ØØ                DEFB    Ø
ØØ                DEFB    Ø
Ø9                DEFB    9
ØØ                DEFB    Ø
ØØ                DEFB    Ø
Ø1                DEFB    1
Ø1                DEFB    1
ØØ                DEFB    Ø
ØØ                DEFB    Ø
ØØ                DEFB    Ø
ØØ                DEFB    Ø
85                DEFB    85H
38                DEFB    38H
          ;
          ;UNEXPANDED SPRITE DATA
          ;
```

```
0F      TELDAT  DEFB    15
1F              DEFB    31
3F              DEFB    63
7F              DEFB    127
FF              DEFB    255
FE              DEFB    254
FE              DEFB    254
FC              DEFB    252
78              DEFB    120
30              DEFB    48
01              DEFB    1
03              DEFB    3
03              DEFB    3
07              DEFB    7
07              DEFB    7
        ;
0F              DEFB    15
1F              DEFB    31
3F              DEFB    63
7F              DEFB    127
7F              DEFB    127
7F              DEFB    127
3F              DEFB    63
3F              DEFB    63
0F              DEFB    15
FF              DEFB    255
FF              DEFB    255
FF              DEFB    255
FF              DEFB    255
F0              DEFB    240
60              DEFB    96
        ;
60              DEFB    96
60              DEFB    96
FF              DEFB    255
FF              DEFB    255
F8              DEFB    0F8H
F3              DEFB    0F3H
E7              DEFB    0E7H
CF              DEFB    0CFH
DE              DEFB    0DEH
9C              DEFB    9CH
```



```
        ;
9C              DEFB    9CH
DE              DEFB    0DEH
CF              DEFB    0CFH
E7              DEFB    0E7H
F3              DEFB    243
F8              DEFB    248
FF              DEFB    255
FF              DEFB    255
FF              DEFB    255
FF              DEFB    255
FF              DEFB    255
        ;
FF              DEFB    255
0F              DEFB    15
06              DEFB    6
06              DEFB    6
06              DEFB    6
FF              DEFB    255
FF              DEFB    255
1F              DEFB    31
CF              DEFB    0CFH
E7              DEFB    0E7H
F3              DEFB    243
7B              DEFB    7BH
39              DEFB    39H
39              DEFB    39H
7B              DEFB    7BH
        ;
F3              DEFB    243
E1              DEFB    0E1H
CE              DEFB    0CEH
1F              DEFB    31
FF              DEFB    255
FF              DEFB    255
F0              DEFB    240
F8              DEFB    248
FC              DEFB    252
FE              DEFB    254
FF              DEFB    255
7F              DEFB    127
7F              DEFB    127
```

```
                ;
 3F                     DEFB    63
 1E                     DEFB    30
 ØC                     DEFB    12
 8Ø                     DEFB    128
 CØ                     DEFB    192
 CØ                     DEFB    192
 EØ                     DEFB    ØEØH
 EØ                     DEFB    ØEØH
 FØ                     DEFB    240
 F8                     DEFB    ØF8H
 FC                     DEFB    ØFCH
 FE                     DEFB    ØFEH
 FE                     DEFB    ØFEH
 FE                     DEFB    ØFEH
                ;
 FC                     DEFB    ØFCH
 FC                     DEFB    ØFCH
 FØ                     DEFB    ØFØH


                ;
 3F             CARDAT  DEFB    63
 6Ø                     DEFB    96
 D8                     DEFB    ØD8H
 AØ                     DEFB    ØAØH
 B9                     DEFB    ØB9H
 AB                     DEFB    ØABH
 B9                     DEFB    ØB9H
 85                     DEFB    85H
 8F                     DEFB    8FH
 85                     DEFB    85H
 81                     DEFB    81H
 8Ø                     DEFB    8ØH
                ;
 81                     DEFB    81H
 83                     DEFB    83H
 81                     DEFB    81H
 85                     DEFB    85H
 8F                     DEFB    8FH
 85                     DEFB    85H
 81                     DEFB    81H
 8Ø                     DEFB    8ØH
```



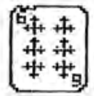
```
 81                     DEFB    81H
 85                     DEFB    85H
 8F                     DEFB    8FH
 85                     DEFB    85H
 81                     DEFB    81H
                ;
 83                     DEFB    83H
 81                     DEFB    81H
 8Ø                     DEFB    8ØH
 8Ø                     DEFB    8ØH
 CØ                     DEFB    ØCØH
 6Ø                     DEFB    96
 3F                     DEFB    63
 FF                     DEFB    255
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
 81                     DEFB    81H
 ØØ                     DEFB    Ø
 42                     DEFB    66
 E7                     DEFB    ØE7H
                ;
 42                     DEFB    66
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
 81                     DEFB    81H
 ØØ                     DEFB    Ø
 42                     DEFB    66
 E7                     DEFB    ØE7H
 42                     DEFB    66
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
 42                     DEFB    66
 E7                     DEFB    ØE7H
 42                     DEFB    66
 ØØ                     DEFB    Ø
 81                     DEFB    81H
 ØØ                     DEFB    Ø
 ØØ                     DEFB    Ø
```

```
00              DEFB    0
        ;
00              DEFB    0
00              DEFB    0
FF              DEFB    0FFH
FC              DEFB    0FCH
06              DEFB    6
03              DEFB    3
01              DEFB    1
81              DEFB    81H
C1              DEFB    0C1H
81              DEFB    81H
A1              DEFB    0A1H
F1              DEFB    0F1H
A1              DEFB    0A1H
81              DEFB    81H
01              DEFB    1
81              DEFB    81H
        ;
C1              DEFB    0C1H
81              DEFB    81H
A1              DEFB    0A1H
F1              DEFB    0F1H
A1              DEFB    0A1H
81              DEFB    081H
01              DEFB    1
81              DEFB    81H
A1              DEFB    0A1H
F1              DEFB    0F1H
A1              DEFB    0A1H
81              DEFB    081H
DD              DEFB    0DDH
        ;
95              DEFB    095H
1D              DEFB    29
05              DEFB    5
1B              DEFB    27
06              DEFB    6
FC              DEFB    0FCH
        ;
        ;IMAGE AREA FOR EXPANDED SPRITE DATA
        ;LENGTH=4*5*64+(4*8)
```



```
        ;
        TELSPC  DEFS    1312
        ;
        ;LENGTH=5*4*64+(5*8)
        ;
        CARSPC  DEFS    1320
0000            DEFW    0
0000            DEFW    0
0000            DEFW    0
0000            DEFW    0
```

If you have followed the last few chapters accurately, the images which should appear on your screen look like this:



CHAPTER 13
High-Resolution Colour

Hands up all those of you who have ever had the need for more than the two colours normally available in each character cell. Well behold, your wishes are about to be granted. With the routines in this chapter you will be able to cover an area of the screen eight columns wide and up to twenty four lines deep with colour attributes at eight times normal resolution; that is, one attribute byte for each row of each cell in the high-resolution area.

The routine works using our tried and trusted technique of interrupts vectored under interrupt mode 2 (IM 2) to our own interrupt handler, as described in Chapter 7.

On receiving an interrupt, the Spectrum will execute a suitable delay routine while it waits for the TV beam to approach the high-resolution area. From then on we have exactly 224 T-states to send as long a row as possible of our 'high-resolution' attributes to the normal attribute file. Experimentation proves that, under the usual restriction that the routine is placed in the top 32K of RAM to avoid delays due to ULA interference, it is possible to replace the attributes of just 8 cells if we are to have time to adjust our pointers and counters ready for the next row of attributes.

The new attributes will be stored in a special high-resolution attribute file, the start of which we shall label with the two-byte variable HIATT. For maximum flexibility, the high-resolution area will be of variable length and variable vertical position. Labelling the top line of the screen as zero

and counting downwards, the first line in the high-resolution area will be STRTLN and the number of lines in the area will be specified by the one-byte variable DEPTH.

The interrupt handler, which I have christened HIRES, includes two stack operations inside its main loop. In order to ensure that these do not run the risk of ULA interference by access to the lowest 16K of RAM, the routine stores away the value of SP in VALSP and then uses its own two-byte machine stack, placed immediately before the routine and thus in the top 32K of RAM.

Central to the routine is a sequence of eight consecutive LDI instructions to load the attributes from the high-resolution file to the normal file. This is the fastest possible method of data transferance, each operation taking a nominal 16 T-states. This compares with a usual 21 T-states per repitition of the LDIR instruction (this only takes 16 T-states on its final execution, when BC = Ø).

High resolution attributes are, of course, mapped out in exactly the same manner as the standard attribute bytes. Bits Ø to 2 are for INK, bits 3 to 5 are for PAPER, bit 6 for BRIGHTness, and setting bit 7 denotes FLASH 1.

Here then is the listing for HIRES, the interrupt handler, followed shortly after by an initialisation routine. Remember the restrictions; the interrupt handler, its preceeding variables, and the high-resolution attribute file must all be in the top 32K of RAM.

```
        ;HIGH RESOLUTION COLOUR
        ;N.B! POSITION ABOVE 32K BOUNDARY
        ;VARIABLES AND ROOM FOR A TWO-BYTE
        ;MACHINE STACK:- USED BY INTERRUPT
        ;HANDLER
        ;
ØØ      STRTLN  DEFB    Ø
18      DEPTH   DEFB    24
ØØØØ    HIATT   DEFW    Ø
ØØØØ    VALSP   DEFW    Ø
ØØØØ            DEFW    Ø
        ;
        ;PRESERVE REGISTERS
        ;
C5      HIRES   PUSH    BC
D5              PUSH    DE
E5              PUSH    HL
```

```
F5                PUSH    AF
             ;
             ;STORE SP AND USE THE TWO BYTES PRECEDING
             ;THIS ROUTINE AS A STACK
             ;
ED73Ø4ØØ          LD      (VALSP),SP
31Ø8ØØ            LD      SP,HIRES
             ;
             ;PRODUCE AN EXACT DELAY
             ;
Ø118Ø2            LD      BC,Ø218H
ØB           DELAY DEC    BC
78                LD      A,B
B1                OR      C
2ØFB              JR      NZ,DELAY
             ;
             ;CALCULATE # OF TEXT ROWS ABOVE HI-RES AREA
             ;
3AØØØØ            LD      A,(STRTLN)
87                ADD     A,A
87                ADD     A,A
87                ADD     A,A
CA2FØØ            JP      Z,GO4IT2
             ;
             ;WAIT UNTIL BEAM REACHES HI-RES AREA
             ;EACH LOOP TAKES 224 T-STATES, OR ONE TV-ROW
             ;
Ø6ØF         SCANL LD     B,15
1ØFE         LN2   DJNZ   LN2
ØØ                NOP
ØØ                NOP
C8                RET     Z
3D                DEC     A
C224ØØ            JP      NZ,SCANL
             ;
             ;CALCULATE ATTRIBUTE ADDRESS FOR (STRTLN,12)
             ;
6F           GO4IT2 LD    L,A
3AØØØØ            LD      A,(STRTLN)
67                LD      H,A
CB3C              SRL     H
CB1D              RR      L
```



```
CB3C              SRL     H
CB1D              RR      L
CB3C              SRL     H
CB1D              RR      L
11ØC58            LD      DE,58ØCH
19                ADD     HL,DE
             ;
             ;PUT ATT. ADDRESS IN DE
             ;
EB                EX      DE,HL
             ;
             ;TAKE START OF HI-RES ATTRIBUTE FILE
             ;
2AØ2ØØ            LD      HL,(HIATT)
             ;
             ;A COUNTS THE NUMBER OF LINES LEFT
             ;
3AØ1ØØ            LD      A,(DEPTH)
             ;
             ;BC COUNTS THE HI-RES COLOUR BYTES
             ;FOR THIS LINE
             ;
Ø14ØØØ       NXLINE LD    BC,64
             ;
             ;SAVE ADDRESS OF LEFT-HAND ATTRIBUTE ON THIS LINE
             ;
D5           NXTROW PUSH  DE
             ;
             ;TRANSFER THE EIGHT ATTRIBUTES FOR THIS ROW
             ;
EDAØ              LDI
EDAØ              LDI
EDAØ              LDI
EDAØ              LDI
EDAØ              LDI
EDAØ              LDI
EDAØ              LDI
EDAØ              LDI
             ;
             ;RETRIEVE ADDRESS OF LEFT-HAND ATTRIBUTE
             ;
D1                POP     DE
```

```
        ;
        ;IF BC=Ø THEN ROW 7 IS COMPLETE
        ;
E26AØØ          JP      PO,LSTROW
        ;
        ;23 T-STATE TIMING EQUALIZER
        ;
18ØØ            JR      $+2
ØØ              NOP
E6FF            AND     ØFFH
18E4            JR      NXTROW
        ;
        ;ADD 32 TO ATTR. ADDRESS AND MOVE
         ON TO NEXT LINE
        ;
EB      LSTROW  EX      DE,HL
ØE2Ø            LD      C,32
Ø9              ADD     HL,BC
EB              EX      DE,HL
3D              DEC     A
C2ØØØØ          JP      NZ,NXLINE
        ;
        ;RETRIEVE SP, THEN UNSTACK THE OTHER REGISTERS
        ;
ED7BØ4ØØ        LD      SP,(VALSP)
F1              POP     AF
E1              POP     HL
D1              POP     DE
C1              POP     BC
        ;
        ;RETURN FROM INTERRUPT
        ;NOTE:  YOU COULD INSERT A JUMP TO THE ROM
        ;INTERRUPT ROUTINE HERE (TO ØØ38H)
FB              EI
ED4D            RETI
```

The interrupts will be intercepted by means of a 257 byte vector table, starting at an arbitrary page boundary which I have chosen to be FEØØH. This technique was detailed in Chapter 7. We need a routine to set up the vector table and select interrupt mode 2, and it shall be called HIRON for HIgh Resolution ON. Following the routine is a short fragment to set up the jump instruction to HIRES at FDFDH, to which all interrupts are vectored.


```
        ;INITIALIZE INTERRUPT INTERCEPTION
        ;WITH A 257 BYTE VECTOR TABLE AT ØFEØØH
        ;
        ;EXIT:  BC=Ø,DE=ØFFØ1,HL=ØFFØ1
        ;
3EFE    HIRON   LD      A,ØFEH
ED47            LD      I,A
Ø1ØØØ1          LD      BC,Ø1ØØH
67              LD      H,A
69              LD      L,C
57              LD      D,A
58              LD      E,B
36FD            LD      (HL),ØFDH
EDBØ            LDIR
ED5E            IM      2
C9              RET
        ;
        ;PRODUCE THE JUMP TO HIRES
        ;AFTER AN INTERRUPT
        ;

        LABEL   ORG     ØFDFDH
C3ØØØØ          JP      HIRES
                ORG     LABEL
```

Now that you have the interrupt handler and initialisation routine, you have all the means to produce high resolution colour, and its nearly time for some examples.

The maximum high-resolution area is 8 × 24 = 192 cells, and hence at most 192 × 8 = 1536 attribute bytes are required, or 1.5K of memory. As it stands, the routine HIRES positions this area in the centre of the screen, starting at column twelve. Some variation in this is possible by altering the base address of the attribute area held in the instruction:

```
LD      DE,58ØCH
```

shortly after label GO4IT2. Some timing adjustment may be necessary, but on my Spectrum I found that the leftmost column of the high-resolution area could quite happily be varied between column Ø and column 13. Thus to cover the area from column 5 to column 12 inclusive, change the instruction to:

```
LD      DE,58Ø5H
```

Unlike the full screen horizon generator in the interrupt handler of Chapter 9, HIRES does not depend on returning to a HALT instruction before each interrupt to maintain stability and prevent flickering attributes. Once we have 'turned on' the high resolution colour, therefore, we are free to do any processing we like without worrying about when the interrupts occur, just as long as we don't disable them.

By substituting the instruction:

```
JP      ØØ38H
```

for the pair

```
EI
RETI
```

at the end of HIRES, we would cause a jump to the standard ROM interrupt handler after every high-resolution frame had been generated. It would then be safe to return to BASIC, which would run normally, apart from the fact that the deeper and lower the high-resolution area, the slower BASIC would get!

For the first example, I have simply pointed HIRES at the start of the ROM and told it to display the first 1.5K in the full-blown high-resolution area, for 256 TV frames (5.12 seconds). The routine is called DEMO1 (points for imagination . . .?).

```
AF        DEMO1  XOR    A
32ØØØØ           LD     (STRTLN),A
                 ;
                 ;USE FIRST 1.5K OF ROM AS
                 ;HI-RES COLOUR FILE
                 ;
6F               LD     L,A
67               LD     H,A
3E18             LD     A,24
32ØØØØ           LD     (DEPTH),A
22ØØØØ           LD     (HIATT),HL
CDØØØØ           CALL   HIRON
                 ;
                 ;NOTE L=Ø FROM HIRON
                 ;
45               LD     B,L
                 ;
```



```
                 ;PRODUCE HI-RES COLOUR FOR
                 ;5.12 SECONDS
                 ;
76               TSLP3  HALT
1ØFD                    DJNZ   TSLP3
                 ;
                 ;RESELECT IM 1 FOR RETURN TO BASIC
                 ;
ED56                    IM     1
3E3F                    LD     A,3FH
ED47                    LD     I,A
C9                      RET
```

The second demonstration is slightly more exotic, and involves the use of a subroutine DATPRP to generate a 25 line attribute file. Obviously not all of these lines can be used at any one time, but by cycling the label pointing at the 'start' of the file, HIATT, backwards or forwards in steps of eight bytes, we can make the high-resolution attributes 'scroll' up or down the screen.

DATPRP, for DATa PRePARer, generates a 25 line attribute file, each line the same, and each row of a line having jut one paper colour and white ink. In an effort to provide some colour separation, I have used the sequence black, magenta, yellow, blue, green, white, red, cyan for the paper colours of successive rows. However, whether it is possible to distinguish these separate colours (or shades, for those of you reading in black and white) will depend on the resolution of your TV sets, over which I regrettably have no control.

Here comes DATPRP, followed closely by DEMO2.

```
                 ;DEMO ROUTINE TO SET UP A 25 LINE
                 ;HI-RES COLOUR FILE
                 ;SPACE NEEDED=25*64=1984
                 ;
                 TSTDAT  DEFS   1984
                 ;
21ØØØØ           DATPRP  LD     HL,TSTDAT
                 ;
                 ;FOR 25 LINES
                 ;
ØE19                     LD     C,25
                 ;
                 ;USE BLACK PAPER ON ROW Ø
```

```
             ;
AF                   XOR      A
             ;
             ;ALWAYS USING WHITE INK
             ;
F6Ø7         NXCOLR  OR       7
             ;
             ;CREATE A ROW OF 8 HI-RES ATTRIBUTES
             ;
Ø6Ø8                 LD       B,8
77           FL9     LD       (HL),A
23                   INC      HL
1ØFC                 DJNZ     FL9
             ;
             ;NEXT PAPER COLOUR
             ;
C618                 ADD      A,24
E638                 AND      38H
2ØF2                 JR       NZ,NXCOLR
             ;
             ;NEXT LINE
             ;
ØD                   DEC      C
2ØEF                 JR       NZ,NXCOLR
C9                   RET
```

DEMO2 will make a multicoloured pile shrink into the 'ground', with the colours scrolling downwards as it goes. The routine is best run with a global black paper and border.

```
        ;SET-UP 25-LINE ATT. FILE AND TURN ON HI-RES
        ;
CDØØØØ  DEMO2   CALL    DATPRP
CDØØØØ          CALL    HIRON
        ;
        ;CYCLE (HIATT) BACKWARDS THROUGH THE FIRST
        ;EIGHT ROWS, MAKING THE COLOUR FLOW DOWN THE
        ;SCREEN. NOTE: THIS IS WHY WE NEED 25-LINES,
        ;NOT 24
        ;
11F8FF          LD      DE,ØFFF8H
        ;
        ;AFTER EVERY CYCLE DECREMENT (DEPTH)
```



```
         AND INCREMENT
        ;(STRTLN), MAKING THE HIRES AREA SHRINK
         DOWNWARDS
        ;
ØE18            LD      C,24
79      TSLP    LD      A,C
32ØØØØ          LD      (DEPTH),A
3E18            LD      A,24
91              SUB     C
32ØØØØ          LD      (STRTLN),A
Ø6Ø8            LD      B,8
21ØØØØ          LD      HL,TSTDAT+64
22ØØØØ  NXRUN   LD      (HIATT),HL
76              HALT
19              ADD     HL,DE
1ØF9            DJNZ    NXRUN
ØD              DEC     C
2ØE7            JR      NZ,TSLP
        ;
        ;RESELECT IM 1 FOR RETURN TO BASIC
        ;
ED56            IM      1
3E3F            LD      A,3FH
ED47            LD      I,A
C9              RET
```

The final demonstration routine for HIRES is a rather spectacular piece called (you guessed it) DEMO3. Again, it uses the ROM to provide a fairly random attribute file, but this time it spend 3Ø.72 seconds or so running HIATT backwards from Ø6ØØH to zero. The result is a very captivating pattern. Try followng its movement from left to right, and then look from right to left across it. Do you notice any difference in its apparent speed?

```
        ;SET UP FULL LENGTH HI-RES AREA
AF      DEMO3   XOR     A
32ØØØØ          LD      (STRTLN),A
3E18            LD      A,24
32ØØØØ          LD      (DEPTH),A
        ;
        ;TURN ON THE HI-RES COLOUR
        ;
CDØØØØ          CALL    HIRON
        ;
```

```
         ;USE ROM AS HI-RES COLOUR FILE, STEPPING HIATT
         ;BACKWARDS FROM Ø6ØØH TO ZERO. NOTE
          THAT THERE ARE Ø6ØØH HI-RES ATTRIBUTES
         ;
2606              LD      H,6
         ;
         ;NOTE L=Ø FROM HIRON
         ;
220000   TSLP2    LD      (HIATT),HL
76                HALT
2D                DEC     L
20F9              JR      NZ,TSLP2
25                DEC     H
20F6              JR      NZ,TSLP2
         ;
         ;RESELECT IM 1 FOR RETURN TO BASIC
         ;
3E3F              LD      A,3FH
ED56              IM      1
ED47              LD      I,A
C9                RET
```

In closing this chapter I ought to point out that the above format is not the only possible layout for high-resolution colour. For a start, if you were willing to have just one attribute byte per scan line, then you wouldn't need to 'dump' a high-resolution colour file on a one-to-one basis, and could dispense with the 16 T-state:

```
LDI
```

in favour of a pair of instructions like:

```
LD      (DE),A
INC     E
```

Where the accumulator would hold the current row attribute, and each pair would take 11 T-states. This way you could probably increase the width of the high-resolution area by three or four columns.

CHAPTER 14
Producing Full-Screen Images with the Border

Spectacular though it was, the full-screen horizon generated in Chapter 9 by switching the border colour one hundred times a second (at 1ØØ Hz) was merely scatching the surface of the potential effects of direct border colour control. In this chapter, I shall realise the full potential of high-speed border switching with a set of routines that will allow you to produce ten distinct columns on the border, with each row of each column taking any one of the eight colours. The switching speeds involved will stretch the Z-8Ø processor to its limits, with a 12 T-state gap between colour changes and an average frequency over one TV row of 15625Ø Hz.

The principles involved in the 'picture generator' are very similar to those of our full-screen horizon. We use interrupts vectored under interrupt mode 2 to our own customised interrupt handler, which after executing appropriate delays for the TV beam to descend to the screen, hurtles through a table of border values like a bat out of hell, always changing the border colour at exactly the same stages in the generation of each TV frame.

To go into more detail, recall that the time taken for the TV to generate one row of the display is exactly 224 T-states. Now the fastest way of transferring data from a table to port 254 is by using a sequence of OUTI instructions, each of which takes 16 T-states. As this instruction is so rarely used, I shall take the trouble to detail its action for you.

The HL pair holds the address of the data byte, the C register holds the lo-byte of the port address, and the B register supplies the hi-byte of that address. In every execution, the B register is decremented, the port address is formed, the data byte at HL is sent out to the port, and HL is incremented. If B reaches zero then the zero flag is set, if not, then it is reset.

Theory would dictate that we can produce INT (224/16) = 14 'border columns' on the screen, but we must remember that the TV beam spends a certain amount of time in 'horizontal flyback' from the right hand edge to the left hand edge of the screen. Experimentation reveals that this traversal occupies the beam for around 64 T-states, or 2/7 or about 29% of its time.

We consequently have just enough time to change the border colour ten times as the beam crosses the screen from left to right, and this results in each 'border column' being four text columns in width.

I name this interrupt handler BORPIC for obvious reasons. The border data for BORPIC will be stored anywhere you like in the top 32K of RAM, and must be pointed to by the two-byte variable PICDAT. We shall format the border data as follows:

FIRST BYTE: NUMBER OF BORDER LINES

then the data for each 'border line':

FIRST BYTE: NUMBER OF TV ROWS IN THIS BORDER LINE

TEN BYTES: The border values for each of the 1Ø border columns.

The concept of border lines is analogous to that of text lines, except that border lines have a variable number of rows in them (up to 256), and the rows continue above and below the text area. It is easily seen that the storage area needed for a picture with n border lines is given by:

Memory needed = (11 * n) + 1

In the listing of BORPIC you will see that I have reserved room for ten lines of border data and labelled it BORSTR. This area will be used later, but for now here is the listing. Please don't try to run it until I've explained how to!

```
ØØØØ      PICDAT  DEFW    Ø
          ;
          ;PICDAT HOLDS ADDRESS OF BORDER DATA
          ;SPACE NEEDED=1+11*(NO. OF BORDER LINES)
          ;
```



```
          BORSTR  DEFS    111
          ;
          ;BORDER PICTURE GENERATOR PRESERVE REGISTERS
          ;
C5        BORPIC  PUSH    BC
D5                PUSH    DE
E5                PUSH    HL
F5                PUSH    AF
Ø8                EX      AF,AF'
F5                PUSH    AF
          ;
          ;WAIT 38 T-STATES
          ;
E3                EX      (SP),HL
E3                EX      (SP),HL
          ;
          ;WAIT FOR (FLYBAK+1) TV ROWS WHILE BEAM REACHES
          ;TOP OF SCREEN
          ;
3E1F      FLYBAK  LD      A,31
Ø6ØF      SCANM   LD      B,15
1ØFE      LN4     DJNZ    LN4
ØØ                NOP
A7                AND     A
C8                RET     Z
3D                DEC     A
C27BØØ            JP      NZ,SCANM
          ;
          ;5 T-STATE TIMING TRIMMER
          ;
CØ                RET     NZ
          ;
          ;POINT HL AT PICTURE DATA
          ;
2AØØØØ            LD      HL,(PICDAT)
          ;
          ;C HOLDS PORT VALUE
          ;
ØEFE              LD      C,ØFEH
          ;
          ;A COUNTS THE LINES OF BORDER DATA
          ;
```

```
7E              LD      A,(HL)
23              INC     HL
Ø8      NXTLN2  EX      AF,AF'
        ;
        ;A COUNTS ROWS FOR THIS BORDER LINE
        ;
7E              LD      A,(HL)
23              INC     HL
        ;
        ;STORE START OF THIS ROW OF DATA IN DE
        ;
54              LD      D,H
5D              LD      E,L
        ;
        ;THE CORE OF 1Ø SUCCESSIVE BORDER CHANGES
        ;
EDA3    NXTRW   OUTI
EDA3            OUTI
EDA3            OUTI
EDA3            OUTI
EDA3            OUTI
EDA3            OUTI
EDA3            OUTI
EDA3            OUTI
EDA3            OUTI
EDA3            OUTI
        ;
        ;GENERATE NEXT ROW OF DISPLAY
        ;
3D              DEC     A
CAB6ØØ          JP      Z,NXTLN
62              LD      H,D
6B              LD      L,E
        ;
        ;FIRST WAIT 3Ø T-STATES
        ;
Ø6ØØ            LD      B,Ø
Ø6ØØ            LD      B,Ø
18ØØ            JR      $+2
ØØ              NOP
18DD            JR      NXTRW
        ;
```



```
        ;NEXT LINE OF BORDER DATA
        ;
Ø8      NXTLN   EX      AF,AF'
3D              DEC     A
        ;
        ;7 T-STATES EQUALIZER
        ;
E6FF            AND     ØFFH
C28EØØ          JP      NZ,NXTLN2
        ;
        ;RETRIEVE REGISTERS AND RETURN FROM INTERRUPT
        ;
F1              POP     AF
Ø8              EX      AF,AF'
F1              POP     AF
E1              POP     HL
D1              POP     DE
C1              POP     BC
FB              EI
ED4D            RETI
```

As I said, BORPIC will be run as an interrupt handler using IM2. We shall use the usual 257-byte table of vectors to a jump instruction to BORPIC, a technique described in detail in Chapter 7. As usual, which page boundary you place the table at and where you put the jump instruction are entirely up to you. If you are undecided, however, then why not put the vector table at FEØØH and the jump instruction at FDFD H. The latter may be achieved by adding the lines:

```
                LABEL   ORG     ØFDFDH
C3ØØØØ                  JP      BORPIC
                        ORG     LABEL
```

The routine HIRON from Chapter 13 may then be used to set up the vector table and select interrupt mode two (it was used previously to set up the same table for the high-resolution colour routine HIRES).

Right then, now we have the routines necessary to make the border picture a reality. The generator is extremly sensitive to timing variations, and so, as in the case of the full-screen horizon generator in Chapter 9, we must always return to a HALT instruction before an interrupt. This way we have a maximum variation in timing of 4 T-states, the time taken for the processor to execute a NOP, which is what it repeatedly does when the HALT instruction is reached.

Referring to the listing of BORPIC, you will see a mysterious label on the line:

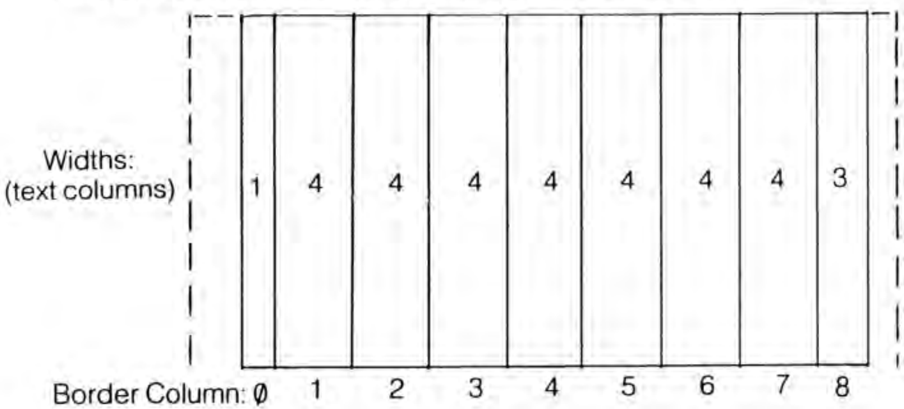
```
FLYBAK  LD      A,31
```

We will use this to make adjustments to the height at which the border picture starts on the screen. The value loaded into A is the number of TV rows the routine should wait before commencing the border data processing. If you want a picture to start right at the top of the screen, then adjust (FLYBAK + 1) until it does. The resulting value will depend on your particular television set as well as your Spectrum.

In the case of my colour portable I found that loading (FLYBAK + 1) with 31 brough the beam down to the top of the screen. There were then 32 rows of top border left before the text area, and indeed it is the general case that the depth of this 'top margin' plus the value in (FLYBAK + 1) should be 63. There are, of course, 192 rows in the text area. Below this is the 'bottom margin', the visible depth of which varies with different TVs and Spectrums, but on my system is about 44 rows deep, giving a total of 32 + 192 + 44 = 268 rows on the screen.

Although we can now produce a stable image on the screen border, the picture will be somewhat incomplete unless we can show the parts of the 'border columns' and 'border lines' which are, if you like, 'behind' the text area. What we need is a routine which examines the data for the 'invisible' part of the border and sets the paper attributes of all text cells appropriately, so that after 'switching on' the picture generator we appear to have a full-screen image, and the boundary between the text area and the border cannot be detected. I shall now develop just such a routine, and call it ATTSET.

Nine of the border columns overlap the text area as shown:



For the purposes of this routine we will assume that the text area has been divided into exactly six border lines, each of 32 rows in depth. If you prefer to have narrower border lines or possibly border lines of variable depth, then ATTSET is easily adjusted. It will not have escaped your notice that 32 rows = 4 text lines in depth, so with this format we end up producing 'border cells' that are four text cells square.

The principles involved in ATTSET are really very simple; we enter the routine with HL pointing at the border data for the first column of the first border line in the text area. ATTSET takes this byte, multiplies it by eight to obtain a PAPER value, ORs it with the INK value of the first cell and then places the lot in the first byte of the attribute file. The next border value is taken and used for the next four text columns of line Ø, and this procedure is repeated for the next six border columns. The value of border column eight is used for the final three text columns, and this line of border data is then reprocessed three times for the remaining text lines of this border line.

The whole of the above procedure is then repeated for each of the five remaining border lines in the text area, and the listing of ATTSET evolves, followed by a demonstration.

```
         ;ROUTINE TO SET THE PAPER ATTRIBUTES
          GIVEN BORDER DATA
          ;ENTRY  :HL=ADDRESS OF FIRST BYTE OF BORDER
          DATA IN  TEXT AREA AS PRODUCED BY "EXPAND"
          ;EXIT:BC=Ø, HL=5BØØH
          ;
          ;POINT HL AT START OF ATTRIBUTES
11ØØ58    ATTSET  LD      DE,58ØØH
EB                EX      DE,HL
          ;
          ;B COUNTS THE BORDER LINES
          ;
Ø6Ø6              LD      B,6
          ;
          ;C COUNTS THE ATTRIBUTE LINES (4 PER BORDER LINE)
          ;
ØEØ4      NXTLN3  LD      C,4
          ;
          ;STORE BORDER DATA ADDRESS
          ;
D5        NXT14   PUSH    DE
C5                PUSH    BC
```

```
        ;C HOLDS MASK FOR PAPER
        ;
ØE38            LD      C,38H
        ;
        ;TAKE BORDER BYTE MULT BY 8 TO GET PAPER BITS
        ;
1A              LD      A,(DE)
Ø7              RLCA
Ø7              RLCA
Ø7              RLCA
        ;
        ;USE THE INK OF THE CELL WITH OUR PAPER TO FORM
        ;NEW ATTRIBUTE BYTE
        ;
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
77              LD      (HL),A
        ;
        ;DO THE SAME FOR THE NEXT SEVEN BORDER COLUMNS
        ;EACH OF WHICH IS FOUR TEXT COLUMNS WIDE
        ;
13              INC     DE
2C              INC     L
Ø6Ø7            LD      B,7
1A      NXT12   LD      A,(DE)
Ø7              RLCA
Ø7              RLCA
Ø7              RLCA
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
77              LD      (HL),A
2C              INC     L
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
77              LD      (HL),A
2C              INC     L
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
```



```
77              LD      (HL),A
2C              INC     L
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
77              LD      (HL),A
2C              INC     L
13              INC     DE
        ;
        ;NEXT BORDER COLUMN
        ;
10E5            DJNZ    NXT12
        ;
        ;NOW DO THE THREE RIGHT-MOST ATTRIBUTE COLUMNS
        ;
1A              LD      A,(DE)
Ø7              RLCA
Ø7              RLCA
Ø7              RLCA
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
77              LD      (HL),A
2C              INC     L
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
77              LD      (HL),A
2C              INC     L
AE              XOR     (HL)
A1              AND     C
AE              XOR     (HL)
77              LD      (HL),A
23              INC     HL
        ;
        ;HL NOW POINTS AT THE NXT LINE OF ATTRIBUTES
        ;
C1              POP     BC
        ;
        ;REPEAT FOR THE NEXT THREE ATTRIBUTE LINES
        ;
ØD              DEC     C
```

```
2803            JR      Z,OUT1
D1              POP     DE
18BB            JR      NXT14
        ;
        ;DISCARD LAST STACK ENTRY
        ;
F1      OUT1    POP     AF
        ;INCREASE POINTER TO NEXT LINE OF BORDER DATA
        ;
13              INC     DE
13              INC     DE
13              INC     DE
        ;
        ;REPEAT FOR FIVE BORDER LINES
        ;
10B3            DJNZ    NXTLN3
C9              RET
```

As a demonstration for BORPIC and ATTSET, we shall produce a multi-coloured 'quilt' pattern of eight border lines by eight border columns, each line being 32 rows deep, as indeed is required by ATTSET. The first 32 rows above the text area are needed for the first line, so we must set (FLYBAK + 1) to 63 − 32 = 31 in order to start generating the image in the right place. The border data will be built up at BORSTR, and the space needed will be 1 + (8 × 11) = 89 bytes, which is within the 111 bytes we reserved in BORPIC.

Since we have one byte for the number of lines, eleven bytes for the first border line and one byte for the depth of the second, the first border value of the second border line will be at (BORSTR + 1 + 11 + 1) = (BORSTR + 13), hence we set the PAPER attributes with:

```
LD      HL,BORSTR+13
CALL    ATTSET
```

The comments in the assembly listing should provide adequate explanation of the rest of the routine, named BPDEMO.

```
        ;DEMONSTRATION FOR BORPIC AND ATTSET
        ;
3E1F    BPDEMO  LD      A,31
320000          LD      (FLYBAK+1),A
        ;
        ;BUILD BORDER DATA AT BORSTR
```



```
        ;
210000          LD      HL,BORSTR
220000          LD      (PICDAT),HL
        ;
        ;START WITH BLACK BORDER
        ;
AF              XOR     A
        ;
        ;DENOTE "8 BORDER LINES"
        ;
3608            LD      (HL),8
23              INC     HL
        ;
        ;LOOP TO GENERATE DATA FOR EACH BORDER LINE
        ;DENOTE "32 ROWS IN THIS LINE"
3620    NXBLIN  LD      (HL),32
23              INC     HL
        ;MAKE FIRST BORDER COLUMN BLACK
        ;
3600            LD      (HL),0
23              INC     HL
        ;
        ;RUN THROUGH THE 8 COLOURS FOR THE
        MIDDLE 8 COLUMNS
0608            LD      B,8
77      NXBCLM  LD      (HL),A
C603            ADD     A,3
E607            AND     7
23              INC     HL
10F8            DJNZ    NXBCLM
        ;
        ;MAKE LAST COLUMN BLACK
        ;
70              LD      (HL),B
23              INC     HL
        ;
        ;CHANGE COLOUR OF SECOND COLUMN TO
        NEXT IN SERIES
C603            ADD     A,3
E607            AND     7
20E8            JR      NZ,NXBLIN
        ;
```

```
              ;SET PAPER ATTRIBUTES TO MATCH BORDER
              ;DATA
              ;
21ØDØØ                LD      HL,BORSTR+13
CDØØØØ                CALL    ATTSET
              ;
              ;TURN ON BORDER PICTURE
              ;
CDØØØØ                CALL    HIRON
              ;
              ;GENERATE IT FOR 5.12 SECONDS
              ;NOTE B=Ø FROM ATTSET
76            TSLP9   HALT
1ØFD                  DJNZ    TSLP9
              ;
              ;RESELECT IM 1 FOR BASIC
              ;
ED56                  IM      1
3E3F                  LD      A,3FH
ED47                  LD      I,A
C9                    RET
```

As a final utility routine for BORPIC I thought it would be rather useful to have one which generates the border data for us, given a set of bit-patterns and colour values, which I shall call collectively 'compact border data'.

The routine EXPAND will allow us to specify any number of border lines, each of any depth (up to 256 in each case), and use two colours for each border line, which will then be defined by the leftmost ten bits of two 'compact data bytes'. Each of these ten bits corresponds to one border column on a border line. Using a system analogous to the Spectrum BASIC's INK and PAPER values, let the two colours available on each border line be BINK and BAPER, denoting a BINK border cell by a 1, and a BAPER cell by a Ø.

To minimise the amount of data required for an image, we will only specify the BINK and BAPER values at the start of the data and whenever we wish to change their values as we work down the screen. We need some way of telling EXPAND to start using new colours, and probably the easiest way to do that is by using the spare six rightmost bits of the data for a line. We will set them to 3FH for a change of colours, then follow that byte with two bytes containing the BINK and BAPER values respectively. Setting the same bits to 3EH will denote 'end of data'. This procedure will become clearer with the examples following the listing of EXPAND.


```
              ;ROUTINE TO EXPAND BORDER DATA FOR BORPIC
              ;
              ;ENTRY: HL=START OF COMPACT BORDER DATA
              ;EXIT:  HL=ADDRESS OF FIRST BORDER VALUE OF
              ;FIRST BORDER LINE IN TEXT AREA
              ;A=Ø, B=BAPER COLOUR, C=BINK COLOUR
              ;DE=NEXT BYTE AFTER COMPACTED DATA
              ;BUILD UP THE DATA IN THE SPACE AT BORSTR
              ;
11ØØØØ        EXPAND  LD      DE,BORSTR
              ;
              ;TRANSFER "# OF LINES"
              ;
EDAØ                  LDI
              ;
              ;A REG. WILL COUNT ROWS OF T.V. DISPLAY ALLOTED
              ;TO DATE
              ;
AF                    XOR     A
Ø8                    EX      AF,AF'
              ;
              ;C=BINK COLOUR
              ;
4E            NEWCOL  LD      C,(HL)
23                    INC     HL
              ;
              ;B=BAPER COLOUR
              ;
46                    LD      B,(HL)
23                    INC     HL
              ;
              ;IF WE'RE AT THE TEXT AREA THEN STORE ADDRESS OF
              ;EXPANDED DATA
              ;
Ø8            NXTWD   EX      AF,AF'
              ;
              ;THE NEXT VALUE MAY BE ALTERED TO CHANGE
              THE DEPTH  OF THE TOP MARGIN
              ;
FE21          TPMRGN  CP      33
C212ØØ                JP      NZ,NYET
D5                    PUSH    DE
```

```
        ;
        ;INCREASE ROW COUNT
        ;
86      NYET    ADD     A,(HL)
Ø8              EX      AF,AF'
        ;
        ;TRANSFER DEPTH OF THIS LINE (IN ROWS)
        ;
EDAØ            LDI
Ø3              INC     BC
        ;
        ;TAKE FIRST COMPACT DATA BYTE
        ;
7E              LD      A,(HL)
EB              EX      DE,HL
D5              PUSH    DE
        ;
        ;FOR EACH OF EIGHT BITS...
        ;
1EØ8            LD      E,8
        ;
        ;PLACE A BAPER BYTE IN BORDER DATA IF BIT IS SET
        ;
17      ABC     RLA
7Ø              LD      (HL),B
D222ØØ          JP      NC,PAPER
        ;
        ;OTHERWISE INSERT A BINK BYTE
        ;
71              LD      (HL),C
        ;
        ;MOVE ON TO NEXT BIT
        ;
23      PAPER   INC     HL
1D              DEC     E
C21CØØ          JP      NZ,ABC
        ;
        ;TAKE SECOND COMPACT BYTE
        ;
D1              POP     DE
13              INC     DE
1A              LD      A,(DE)
```



```
        ;
        ;CHOOSE BAPER OR BINK FOR EACH OF
        LEFT-MOST 2 BITS
17              RLA
7Ø              LD      (HL),B
D23ØØØ          JP      NC,PAPER2
71              LD      (HL),C
17      PAPER2  RLA
23              INC     HL
7Ø              LD      (HL),B
D237ØØ          JP      NC,PAPER3
71              LD      (HL),C
23      PAPER3  INC     HL
        ;
        ;TEST BITS Ø-5 OF SECOND COMPACT DATA BYTE
        ;
1A              LD      A,(DE)
13              INC     DE
EB              EX      DE,HL
        ;
        ;3FH INDICATES NEW COLOURS REQUIRED
        ;
F6CØ            ØR      ØCØH
3C              INC     A
28C7            JR      Z,NEWCOL
        ;
        ;3EH INDICATES END OF DATA
        ;
3C              INC     A
C2ØBØØ          JP      NZ,NXTWD
        ;
        ;IN WHICH CASE, RETRIEVE ADDRESS FOR ATTSET
        ;
E1              POP     HL
23              INC     HL
C9              RET
```

Notice the line

```
        TPMRGN  CP      32
```

The value in this instruction is the number of rows of the border picture which are above the text area, and should always be equal to 63 − (FLYBAK + 1), that is to say the sum of the two values at labels FLYBAK in BORPIC and TPMRGN in expand should be 63:

(FLYBAK + 1) + (TPMRGN + 1) = 63

TPMRGN in case you hadn't guessed, stands for 'top margin'. EXPAND uses this value to find the correct address in the border data for use as an entry value to ATTSET, where such a use is applicable. It then stores this value away and returns it in HL ready for immediate use, if so desired, with ATTSET.

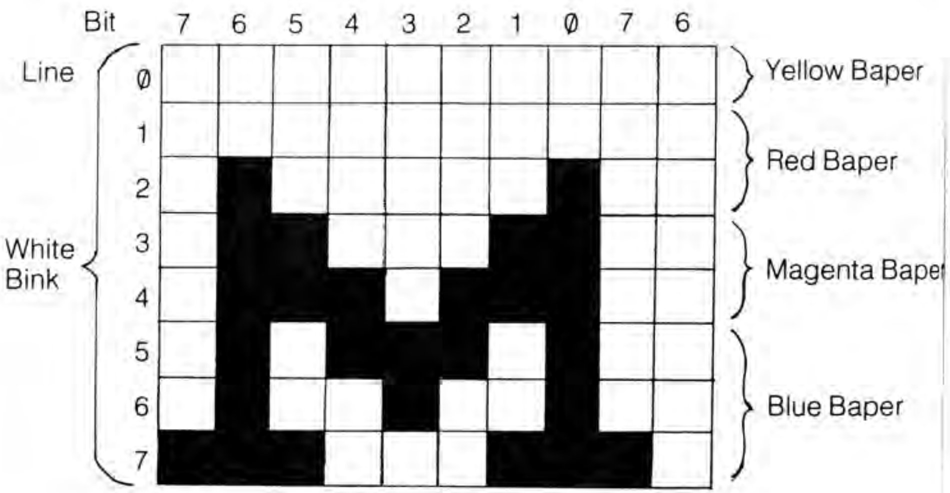
As a simple demonstration for EXPAND, I have written a routine to display a crude but large manifestation of the Melbourne House logo, as seen on the spine of this book. We will be using a grid of 1Ø × 8 square border cells, so we need the top margin to be 32 rows deep. This is set by:

```
LD      A,32
LD      (TPMRGN+1),A
LD      A,31
LD      (FLYBAK+1),A
```

EXPAND builds the data in the previously reserved space at BORSTR, so we must point PICDAT at it:

```
LD      HL,BORSTR
LD      (PICDAT),HL.
```

The desired image is as shown:



On first sight, the bit pattern for this is given by the hex values:

```
ØØ    ØØ
ØØ    ØØ
41    ØØ
63    ØØ
77    ØØ
5D    ØØ
49    ØØ
E3    8Ø
```

We must now incorporate the other information. The first byte must be the number of border lines (8) followed by the first BINK and BAPER values (7 and 6 respectively). We require a new BAPER after line Ø, so we denote this by changing the data

from ØØ ØØ to ØØ 3F

and then include the new value of BAPER after the BINK value, which remains the same. The first seven bytes are now:

Ø8 — BORDER LINES
Ø7 Ø6 — BINK, BAPER
ØØ 3F — DATA FOR LINE Ø
Ø7 Ø2 — BINK, BAPER.

The rest of the data is treated in the same fashion, adding 3EH to the last value to signify 'end of data', thus the last two bytes change

from E3 8Ø to E3 BE.

The final list of compact border data is found at label MELDAT in the assembly listing, so we set up the image with the simple, rapid-fire sequence:

```
LD      HL,MELDAT
CALL    EXPAND
CALL    ATTSET
CALL    HIRON
```

Here is the complete listing, called EXDEMO.

```
        ;DEMO ROUTINE FOR EXPAND ATTSET AND BORPIC
        ;GENERATES THE MELBOURNE HOUSE LOGO
        ;
        ;32 LINES OF THE PICTURE WILL BE ABOVE TEXT
```

```
                ;
3E20            EXDEMO  LD      A,32
320000                  LD      (TPMRGN+1),A
                ;
                ;NOTE 63-32=31 FOR FLYBAK
                ;
3E1F                    LD      A,31
320000                  LD      (FLYBAK+1),A
                ;
                ;BUILD UP BORDER DATA AT BORSTR...
                ;
210000                  LD      HL,BORSTR
220000                  LD      (PICDAT),HL
                ;
                ;BY USING "EXPAND" ON COMPACT BORDER DATA
                ;
212600                  LD      HL,MELDAT
CD0000                  CALL    EXPAND
                ;
                ;NOW SET PAPER ATTRIBUTES APPROPRIATELY
                ;
CD0000                  CALL    ATTSET
                ;
                ;TURN ON BORDER PICTURE GENERATOR
                ;
CD0000                  CALL    HIRON
                ;
                ;GENERATE PICTURE FOR 5.12 SECONDS
                ;NOTE B=0 FROM HIRON
                ;
76              XLP     HALT
10FD                    DJNZ    XLP
                ;
                ;RESELECT IM 1 FOR BASIC
                ;
ED56                    IM      1
3E3F                    LD      A,3FH
ED47                    LD      I,A
C9                      RET
                ;
                ;COMPACT BORDER DATA 8 BORDER LINES
                ;WITH WHITE BINK;
```



```
08              MELDAT  DEFB    8
                ;
                ;ONE WITH YELLOW BAPER
                ;
07                      DEFB    7
06                      DEFB    6
20                      DEFB    32
00                      DEFB    0
3F                      DEFB    03FH
                ;
                ;TWO WITH RED BAPER
                ;
07                      DEFB    7
02                      DEFB    2
20                      DEFB    32
00                      DEFB    0
00                      DEFB    0
20                      DEFB    32
41                      DEFB    41H
3F                      DEFB    3FH
                ;
                ;TWO WITH MAGENTA BAPER
                ;
07                      DEFB    7
03                      DEFB    3
20                      DEFB    32
63                      DEFB    63H
00                      DEFB    0
20                      DEFB    32
77                      DEFB    77H
3F                      DEFB    3FH
                ;
                ;AND THREE WITH BLUE PAPER
                ;
07                      DEFB    7
01                      DEFB    1
20                      DEFB    32
5D                      DEFB    5DH
00                      DEFB    0
20                      DEFB    32
49                      DEFB    49H
00                      DEFB    0
```

```
2Ø              DEFB    32
E3              DEFB    ØE3H
BE              DEFB    ØBEH
```

The routines provided in this chapter have a great many possible uses. You could use BORPIC on its own to provide some very fancy graphics in the top and/or bottom margin (remember you may use border lines as little as one row deep), using a border line 192 rows deep of one colour in between. Alternatively, and remembering that the routines do not affect the display file or INK attributes, you could use BORPIC, ATTSET and EXPAND to provide a spectacular background when, say, an arcade game has been frozen or you have just been exterminated.

I will conclude this chapter with one such example, producing a sequence of four images of the numbers 3, 2, 1 and Ø, in that order. This countdown may be used, for example, as the background to a text image of a submarine about to launch one of its Polaris missiles, in spectacular 3-D, of course.

```
                ;3-2-1-Ø COUNTDOWN
                ;
                ;COMPACT BORDER DATA
                ;
                ;IMAGE "3"
                ;
Ø8              DAT3    DEFB    8
Ø5                      DEFB    5
Ø2                      DEFB    2
2Ø                      DEFB    32
FF                      DEFB    ØFFH
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
Ø1                      DEFB    1
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
Ø1                      DEFB    1
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
FF                      DEFB    ØFFH
8Ø                      DEFB    8ØH
                ;
2Ø                      DEFB    32
FF                      DEFB    ØFFH
```



```
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
Ø1                      DEFB    1
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
Ø1                      DEFB    1
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
FF                      DEFB    ØFFH
BE                      DEFB    ØBEH
                ;
                ;IMAGE "2"
                ;
Ø8              DAT 2   DEFB    8
Ø2                      DEFB    2
Ø4                      DEFB    4
2Ø                      DEFB    32
FF                      DEFB    ØFFH
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
Ø1                      DEFB    1
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
Ø1                      DEFB    1
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
FF                      DEFB    ØFFH
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
FF                      DEFB    ØFFH
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
CØ                      DEFB    ØCØH
ØØ                      DEFB    Ø
2Ø                      DEFB    32
CØ                      DEFB    ØCØH
ØØ                      DEFB    Ø
2Ø                      DEFB    32
FF                      DEFB    ØFFH
BE                      DEFB    ØBEH
                ;
```

```
                ;IMAGE "1"
Ø8              DAT1    DEFB    8
Ø7                      DEFB    7
Ø2                      DEFB    2
2Ø                      DEFB    32
1C                      DEFB    1CH
ØØ                      DEFB    Ø
2Ø                      DEFB    32
7C                      DEFB    7CH
ØØ                      DEFB    Ø
2Ø                      DEFB    32
ØC                      DEFB    12
ØØ                      DEFB    Ø
2Ø                      DEFB    32
ØC                      DEFB    12
ØØ                      DEFB    Ø
2Ø                      DEFB    32
ØC                      DEFB    12
                ;
ØØ                      DEFB    Ø
2Ø                      DEFB    32
ØC                      DEFB    12
ØØ                      DEFB    Ø
2Ø                      DEFB    32
7F                      DEFB    7FH
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
7F                      DEFB    7FH
BE                      DEFB    ØBEH
                ;
                ;IMAGE "O"
                ;
Ø8              DATØ    DEFB    8
Ø1                      DEFB    1
Ø5                      DEFB    5
2Ø                      DEFB    32
7F                      DEFB    7FH
ØØ                      DEFB    Ø
2Ø                      DEFB    32
FF                      DEFB    ØFFH
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
```



```
C7                      DEFB    ØC7H
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
CD                      DEFB    ØCDH
                ;
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
D9                      DEFB    ØD9H
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
F1                      DEFB    ØF1H
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
FF                      DEFB    ØFFH
8Ø                      DEFB    8ØH
2Ø                      DEFB    32
7F                      DEFB    7FH
3E                      DEFB    3EH
                ;
                ;TABLE OF COMPACT BORDER DATA ADDRESSES
                ;
ØØØØ            BORTAB  DEFW    DAT3
1BØØ                    DEFW    DAT2
36ØØ                    DEFW    DAT1
51ØØ                    DEFW    DATØ
                ;
                ;THE COUNTDOWN ROUTINE******
                ;USE 32 TV-ROWS ABOVE TEXT AREA
                ;
3E2Ø            CNTDWN  LD      A,32
32ØØØØ                  LD      (TPMRGN+1),A
3E1F                    LD      A,31
32ØØØØ                  LD      (FLYBAK+1),A
21ØØØØ                  LD      HL,BORSTR
22ØØØØ                  LD      (PICDAT),HL
                ;
                ;SET UP VECTOR TABLE, BUT DON'T TURN ON
                ;PICTURE GENERATOR UNTIL BORDER DATA IS SET UP
                ;
F3                      DI
CDØØØØ                  CALL    HIRON
216CØØ                  LD      HL,BORTAB
```

```
Ø6Ø4              LD      B,4
           ;
           ;TAKE ADDRESS OF COMPACT BORDER DATA
           ;
5E         NXTNUM LD      E,(HL)
23                INC     HL
56                LD      D,(HL)
23                INC     HL
C5                PUSH    BC
E5                PUSH    HL
           ;
           ;SET UP BORDER DATA AND PAPER ATTRIBUTES
           ;
EB                EX      DE,HL
CDØØØØ            CALL    EXPAND
CDØØØØ            CALL    ATTSET
           ;
           ;PRODUCE THE PICTURE FOR 1 SECOND
           ;
FB                EI
Ø632              LD      B,5Ø
76         PSE    HALT
1ØFD              DJNZ    PSE
F3                DI
           ;
           ;NEXT PICTURE
           ;
E1                POP     HL
C1                POP     BC
1ØE8              DJNZ    NXTNUM
           ;
           ;RESELECT IM 1 FOR BASIC
           ;
3E3F              LD      A,3FH
ED47              LD      I,A
ED56              IM      1
FB                EI
C9                RET
```


APPENDIX A:
A List of all Principal Routines

APPENDIX B:
Recommended Reading

Throughout this book you will have seen references to 'T-states' and the various times taken for different instructions to be executed. A complete breakdown of the timing for each Z-80 instruction, its op-code and its affect on the flags can be found in what is considered by many to be the authoritative guide to Z-80 programming, 'Programming The Z-80 by Rodnay Zaks, published by Sybex.

This book is undoubtedly worth having as a reference guide, although it is somewhat pricey.

The other book no good Spectrum machine language programmer should be without is, 'The Complete Spectrum ROM Disassembly' by Dr. Ian Logan and Dr. Frank O'Hara, published by Melbourne House.

A complete, fully-commented ROM assembly listing almost entirely fills the 236 pages of the book, which should be on hand whenever you need to study how a particular ROM routine has been programmed, or what entry values are necessary to its utilisation.

APPENDIX C:
Recommended Assemblers and Monitor/Disassemblers

Hisoft 'DEVPAC 3'

. . . is comprised of 'GENS3' assembler and 'MON3' monitor/ disassembler. The routines in this book were developed on 'GENS3'. A microdrive-compatible version is available. 'GENS3' is 7K long and hence only practical to use on a 48K Spectrum.

Hisoft
13 Gooseacre
Cheddington
Leighton Buzzard
Beds.
LU7 ØSR

Oxford Computer Publishing (OCP)

. . . supply two separate programs for the 16K/48K Spectrum; 'Full Screen Editor/Assembler' and 'Machine Code Test Tool' — a tutor and debugging monitor.

Oxford Computer Publishing Ltd.
4 High Street
Chalfont St. Peter
Bucks
SL9 9QB

Picturesque

. . . do a very powerful pair of utilities called 'Editor Assembler' and 'Spectrum Monitor', the latter also being a disassembler, and both being for the 16K or 48K Spectrum.

Picturesque
6 Corkscrew Hill
West Wickham
Kent
BR4 9BB

Sinclair Research

. . . a company with which you should now be familiar, publishes 'ZEUS Assembler' and 'Monitor/Disassembler', both for the 48K Spectrum and originally written by Crystal Computing Limited.

Sinclair Research
Stanhope Road
Camberley
Surrey
GU15 3BR.

ADVANCED SPECTRUM MACHINE LANGUAGE
MELBOURNE HOUSE REGISTRATION CARD

Please fill out this page and return it promptly in order that we may keep you informed of new software and special offers that arise. Simply fill in and send to the correct address on the reverse side.

Name ....

Address ....

.... Code ....

What product did you purchase? ....

Which computer do you own? ....

Where did you learn of this product?

☐ Magazine. If so, which one? ....

☐ Through a friend

☐ Saw it in a Retail Shop

☐ Other. Please specify ....

Which magazines do you purchase?

Regularly: ....

Occasionally: ....

What Age are you?

☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ Over 25

We are continually writing new material and would appreciate receiving your comments on our product.

How would you rate this book?

☐ Excellent ☐ Value for money
☐ Good ☐ Priced right
☐ Poor ☐ Overpriced

Please tell us what books you would like to see produced for your SPECTRUM.

____

____

____

PUT THIS IN A STAMPED ENVELOPE AND SEND TO:

In the United Kingdom return page to:
Melbourne House (Publishers) Ltd., Melbourne House, Church Yard, Tring, Hertfordshire, HP23 5LU

In Australia & New Zealand return page to:
Melbourne House (Australia) Pty. Ltd., Suite 4, 75 Palmerston Crescent, South Melbourne, Victoria, 3205.