machine code sprites and graphics
for the zx spectrum

a complete guide to sprite coding

john durst

CONTENTS

Contents in detail

CHAPTER 7
Printing Six-bit Characters
The difficulties of printing in six-bit — organising a counter to deal with the number of bits shifted — shifting the letters into their new positions — saving bytes — using six-bit characters.

CHAPTER 8
Sprites — Animation
Drawing sprites — animating within a single character — storing sprites in upper RAM — 'running man' demonstration.

CHAPTER 9
The Moving Sprite
Moving a sprite smoothly about the screen — the display file layout — printing a single character to the screen — printing a character which is laterally offset — rearranging the bytes of a sprite — printing a sprite to the screen.

CHAPTER 10
Sprite Backgrounds
Restoring the background wiped out by the sprite — make your sprite float free, without a 'white' surround — explaining the matte process — printing sprites with background.

CHAPTER 11
The Attribute File
The arrangement of the attributes file — 'hiding' items — drawing pictures to the attributes file — storing the file — colour.

CHAPTER 12
The Display File
The make-up of a column of print positions — finding user-defined graphic characters — scrolling the file — rearranging the file — shrinking the screen.

CHAPTER 13
Inputs and Outputs
The interrupts — output to the television screen, making an 'icecream' border, 'exploding' the screen — the Spectrum BEEP.

CHAPTER 14
Following a Machine Code Program — Hex/Dec
Following through a machine code program, to convert hex to dec, and dec to hex.
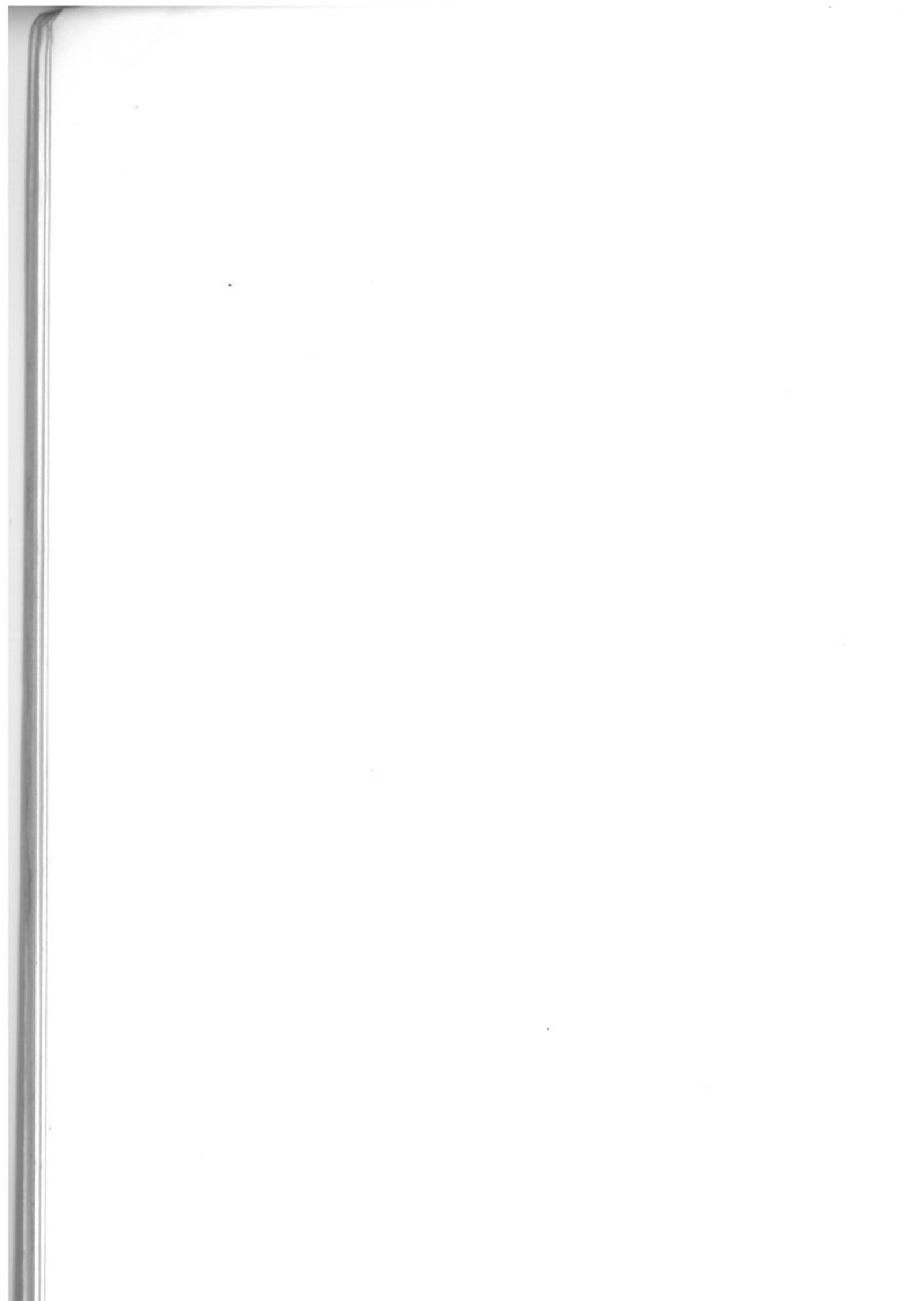
Introduction

This is a book about the Spectrum display. It explains how to print patterns, pictures and letters in a range far more colourful and intricate than you would ever be able to achieve through normal BASIC programming. And you only need the standard Spectrum devices.

You will discover how to draw your own sprites, how to animate them and how to move them about the screen — quite independently of the background. You will learn how to make letters outside the normal Spectrum character range — large letters (from two to eight times normal size), small letters (fitting 40 or even 64 characters to the line) and sideways letters. All of these can be used in your own programs to make them more interesting — more fun.

To do all this, you have to get down to the nitty-gritty of how the character set, the display file and the attributes file are organised. By the end of the book — I hope! — you will be able to manipulate these as neatly and spectacularly as a circus juggler manages his plates and bottles. And once you understand all they do, and everything you can do with them, you will be able to use your Spectrum in ways Sinclair Research never dreamed of!

The techniques described here use machine code most of the time, to get around the restrictions of BASIC. Although I have assumed that you will know something of machine code programming, I have tried to make the *thinking* behind the routines as simple and as logically developed as possible: I hope that everyone will be able to follow the development of each program, step by step.

Nearly all of the routines will stand alone: that is, they can be programmed and run individually to produce a certain result. But few of them are meant to be left at that. They are intended to be used by you — to be adapted and incorporated in your programs, to give you new ideas for better programs. The routines will do what *you* want them to do.

I hope that this book turns out to be useful but, above all, I hope that you enjoy it.

ZX Machine Code User's Club

To keep up to date with machine code techniques, most people reading this book would enjoy the ZX Machine Code User's Club. The club holds meetings, from time to time, and publishes a magazine, 'MicroArts' (with pages numbered in hex!) full of good things. The club is non-profit-making — actually, more loss-making, at the moment. Particulars from the Secretary, Miss Toni Baker, 37 Stratford Road, Wolverton, Milton Keynes MK12 5LW.

Program Notes

Please be careful when entering programs that you key in the correct characters. In particular, the number 1 and lower case l, and commas and full stops, can look very similar.

Page references to the Spectrum manual are to the second edition (1983).

CHAPTER 1
Into Machine Code

Any writer has to start by making some assumptions, and my main one is that the reader of this book will not be a complete novice at machine code programming. You don't think that the 'BC register' is a list of 2000-year-old families, or that 'shift right logical' has something to do with politics.

There are many good introductory books on machine coding, which will take you through the rules of the system and explain how to achieve interesting results. If you are hooked (and you probably must be hooked, to buy this book) you will already have a couple of these works, which explain Z80 coding in terms of the Sinclair Spectrum. But, in addition, I would say that there are at least two 'musts' (or near-musts) which you will never regret having on your shelves.

The first is *The Complete Spectrum ROM Disassembly* (Melbourne House) by Dr Ian Logan and Dr Frank O'Hara. The Spectrum ROM is a treasure-house of routines which can be picked out for use in your own programs (some will be mentioned in the course of this book). The great beauty of them is that, since they are there, already debugged, you don't really have to understand how they work — you just need to know their start addresses and what they will do. But quite often you *might* want to change the rules, perhaps by entering a routine at a later point than normal. To follow what happens in any of these routines, there is nothing like being able to go back to basics and consulting the original listing, which is what you can do with the *Spectrum ROM Disassembly*.

The other book I could not get along without is Rodnay Zaks' *Programming the Z80* (Sybex). It's a big book, I'm afraid, and an expensive one, but it contains *everything* concerned with the Spectrum's microprocessor: if you want to find out what happens to the parity flag after some obscure operation (or, indeed, what the parity flag *is*), then Zaks is your man.

I have done my best in this book to explain the more difficult points, but machine code is an intricate subject and you will no doubt find that other, more detailed, explanations will sometimes make things clearer.

Most of the routines given here have starting addresses at F000h — or sometimes at F100h, or F200h. These are quite arbitrary: in fact, if, as I suggest, the routines form part of a bigger program, you would want to

put them at different addresses to form a block with the rest of your program.

The programs were worked out on the 48K Spectrum. However, nearly all of them, except those using very large blocks of memory, will work equally well on the 16K model but, to avoid cluttering already crowded pages, I have not listed the alternative addresses.

If you *are* using a 16K Spectrum, as a general rule, where the 48K Spectrum has addresses beginning at F000h at the very top of its memory, the 16K Spectrum should have addresses beginning at 7000h. So, to fit the routines into a 16K context, you could substitute '7' wherever you find 'F' at the beginning of a hex address (but you would need to check through the programs carefully, to make sure that you had not set any traps for yourself, by doing so).

Hexadecimal v. decimal

The previous paragraph introduces another subject, which we shall need to tackle — the great 'hex v. decimal' controversy. I have used hex addresses because it seems a sensible rule to follow in machine coding, although, when programming in BASIC, decimal notation is really the only option available on the Spectrum. (The limited binary input is chiefly used for creating graphics.)

Why not be consistent and use decimal throughout? The reason is that, in machine code, decimal gives a very poor impression of the underlying binary, which is all the processor understands.

For example, the decimal numbers 19, 27, 35 and 43 are all Z80 instructions. On the face of it, they don't bear much of a family resemblance, although they stand for 'increment DE', 'decrement DE', 'increment HL' and 'decrement HL'. But expressed as hex numbers they become 13, 1B, 23, 2B. You can quickly see that references to the HL register seem to start with '2', while those to the DE register start with '1'; increments of double registers end with '3' and decrements end with 'B'. You could even carry your Sherlocking further and deduce that, if the instruction 'Load BC...' is '01...', then 'increment BC' should be '03'. And you would be right.

However, the Z80 is not entirely based on this idea of 'matching', and other factors intervene. But hex notation brings out the fact that the Z80 instructions are not just a set of arbitrary codes — they are a logically constructed family in which the binary on/off signals carry out a planned structure of tasks.

All this makes it worth our while to use hex notation in setting out machine code routines. I confess that I have never succeeded in learning my 'hex times tables' properly: I always look them up, even if I think I know them. The single-byte codes are easily found from the character set

in the Sinclair manual, on pp.183–188, and most machine code primers have tables from which to work out the longer numbers. If you feel that this involves too much effort, or takes too long, the complete program discussed in Chapter 13 is a machine code routine to convert hex into decimal, and vice versa, which works almost instantaneously.

I should point out here that all references to the Spectrum manual will be to the second edition of the manual, which was produced in 1983.

Hex Entry

Understanding the principles of machine coding is not the same thing as using it: you need more than this if you are going to construct practical programs. Apart from notebook, pen, pocket calculator and reference books, you need some method of entering the machine code on the Spectrum.

All introductory books on machine coding list some method of setting up code in the RAM. They vary in complexity, but all do an efficient job of transforming your jabs at the keyboard into bytes stored in the Spectrum memory. However, I've included another system here.

```
  1 REM ***POKE HEX $STRING***
  2 INPUT "Start address? ";ad
  3 LET ad1=INT (ad/256): LET ad2=ad-25
6*ad1
  4 POKE USR "A",ad2: POKE USR "A"+1,ad
1: CLEAR ad-1
 10 DEF FN a(j)=CODE a$(j)-48-7*(CODE a
$(j)>=65)
 20 DEF FN b$(x)=CHR$ (INT (x/16)+48+7*
(INT (x/16)>9))
 30 DEF FN c$(x)=CHR$ (x-16*INT (x/16)+
48+7*((x-16*INT (x/16))>9))
 40 LET a$="21FF00C9"
 50 IF LEN a$<>2*INT (LEN a$/2) THEN PR
INT "Hex digit missing": STOP
 60 FOR j=1 TO LEN a$: IF NOT ((a$(j)>=
"0" AND a$(j)<="9") OR (a$(j)>="A" AND a
$(j)<="F")) THEN PRINT "Fault at ";j;" =
 ";a$(j): STOP
 70 NEXT j
 80 LET ad=256*PEEK (USR "A"+1)+PEEK US
R "A"
 90 FOR j=1 TO LEN a$ STEP 2: POKE ad-1
+j/2,16*FN a(j)+FN a(j+1): NEXT j
 95 PRINT "Code entered": STOP
```

```
100 REM ***PRINT HEX PAIRS***
110 INPUT "Start address? ";ad
120 FOR j=ad TO ad+100
130 LET ad1=INT (j/256): LET ad2=j-256*
ad1
140 LET byt=PEEK j
150 PRINT j;TAB 10;FN b$(ad1);FN c$(ad1
);FN b$(ad2);FN c$(ad2);TAB 18;FN b$(byt
);FN c$(byt)
160 NEXT j
```

This program asks you to set up a string in line 40(a$) which contains the standard hex listing for your routine (using the digits 0 to 9, and the letters A to F). Lines 50–70 check that the string holds valid notation, after which line 110 asks for a start address in decimal. When that is entered, the program POKEs the required values to the chosen addresses.

The second part of the program, from line 100, allows you to display the hex values of a series of bytes, starting from a chosen address. It also displays the address of each byte in decimal and hex.

While this is not a proper disassembly, it allows a quick check of an entered program — or of any other region of the memory. It can, of course, be made to LPRINT, if required.

The main advantage of this particular Hex Entry program lies in the fact that the listing is always preserved in 'a$'. This makes it possible to edit the coding — to change it, debug it, etc. (Most hex loaders involve typing in code which is immediately POKEd into memory, leaving you with nothing you can see.)

Also, the first thing you learn about machine coding is that the tiniest error seems to end in total disaster, general paralysis of the computer, etc., etc. If you are wise, you *always* SAVE your program, before running it. By having the routine in a string, it can be SAVEd along with the BASIC program, without having to do an extra SAVE and then LOAD bytes, as is the case with programs which load the code straight into the memory.

Storing machine code

It's a nice point to decide whereabouts in the RAM to store machine code. The upper RAM is the location favoured by Sinclair Research (see the manual, p.168) and the 'CLEAR xxxxx' command has been included in the Spectrum layout partly to provide safe space in which to hold the code while a program is being RUN.

As I've said, most of the routines in the present book have been placed at, or about, F000h — sometimes E000h — and, although the decimal

equivalents are not very memorable, '61440' and '57344' soon get etched on to your brain. To provide safe areas for these two starting points, you have to CLEAR one digit less — that is, 'CLEAR 61399' and 'CLEAR 57343'.

Another useful location for the machine code is in a REM statement in line 1 of a BASIC program. A BASIC program on the Spectrum does not have a fixed location, but, in practice, the unexpanded Spectrum always has the first character of a line 1 REM statement at 23762. However, if you have an Interface 1 connected, beware! You will have to find the address you need indirectly, using the system variable, PROG, as outlined in Chapter 6.

The REM statement provides a useful place for a shortish routine, closely linked to a BASIC program, as in the case of the Titivator program in Chapter 6, which uses a machine code routine to produce oversize letters. A program at this location loads automatically and is out of the way of any other pieces of machine code which you may have in action (as might be the case when using Titivator).

The technique used is to prepare a REM line with the required number of 'spaces' and then POKE the code bytes into it (**Figure 1.1**).

```
  1 REM 11111111111111111111111
11111111222222222222222222222222
222222223333333333333333333333333
333333334444444444444444444444444
```

Figure 1.1:   Line 1 REM Statement

It's handy to use numbers for the spaces, because then you can calculate the total easily, since each line contains 32 'spaces'.

Quite often, the fully POKEd REM won't LIST properly, but this doesn't make any difference to the way in which the program works.

If you use the BASIC Hex Entry program to prepare your line 1 REM, you can scrap the entry program once the REM is ready and then MERGE the line 1 with your final BASIC application program for subsequent use.

Assemblers and disassemblers

The trouble with simple hex loaders is that they don't provide any method of viewing the machine code program, other than as a raw hex listing. The best you can hope for is

3A 08 5C D6 20 18 06

rather than

```
F000      3A 08 5C      LD A,(5C08)
F003      D6 20         SUB 20
F005      18 06         JR, F00C
```

let alone

```
F000      3A 08 5C      LD,A (LAST__K)
F003      D6 20         SUB 20          ;GET KEY VALUE
F005      18 06         JR, PRINT
```

Obviously, the more detailed the final printout becomes, the more complicated the program to produce it must be. To get a proper mnemonic display, you really need to buy a professionally-prepared program, on cassette.

These cassette-based programs come in two basic kinds, the 'assemblers' and the 'disassemblers', and, as usual, there is a trade-off between the two when it comes to using them. The assemblers let you key in the Z80 mnemonics direct, from which they will put together the machine code listing — the 'object code'. With a disassembler, you usually have to key in the object code, but you have excellent facilities for display and editing.

The disadvantage of an assembler lie mostly in the fact that it has to be a very complicated program and may not be very user-friendly. It can be as fiddly to key in the source code and to get all the commas, spaces, labels, etc., right, as it is to look up the object code and enter that in a disassembler. You often have to spend quite a long time debugging the source code in the assembler, before you can check whether the program itself will run! On the other hand, assemblers will usually cope with labels, calculate relative jumps and do many similar chores. But a good disassembler will let you debug much more thoroughly, providing BREAK points, facilities for juggling blocks of code, and so on.

There are some super-programs, which combine the virtues of both, but they nearly always need an 80-column printer, and so are more for the software professional than for us poor mortals.

My personal preference is for the disassembler type of program — I like to feel close to the object code. But I freely admit that this is a personal bias, and the Z80 will probably be one of the last microprocessors on which it will be practical. The next generation of 16-bit processors, like the 68008 in the Sinclair QL, will be just about impossible to handle, except through an assembler.

CHAPTER 2
The Memory

A computer works by moving electrical charges about within the microprocessor chip and the memory chips. There are two sorts of memory 'ROM', or read only memory, and 'RAM', or random access memory: (called 'random access memory' because you can access any memory location you wish within it).

Conventionally, as a working analogy, each is pictured as a long line of numbered boxes, each containing an 8-bit byte.

In the Spectrum, the line begins at 0000h and ends at FFFFh (or 7FFFh, if we are dealing with the 16K Spectrum). Notice, by the way, how computers like to begin their counting at '0', rather than at '1' like us mortals. It is, in fact, more logical and it is not a bad habit to adopt when writing programs: then there should be no confusion.

When dealing with machine code, it is essential to be able to find your way about the various sections of the ROM and RAM and a map, or plan, of the layout comes in very handy.

The memory map in **Figure 2.1** is not complete. Other versions will be found in the Sinclair manual and in most machine code books on the Spectrum. The present version has been tailored to suit our particular needs and it does leave out details which are not relevant here.

Much of it will probably be familiar, but it may still be a help to take a quick canter down 'memory lane'.

The memory map

Starting at the bottom, the section of memory from 0000h to 3FFFh is the ROM. This is the powerhouse of the Spectrum and is unchangeable, although its contents can be studied. It is used in most programs, both BASIC (in which it is invariably used) and machine code, when it can be used if it is helpful. Whole books are written about the ROM and we shall look at some parts of it later on.

Beyond the ROM, memory is more unstable. It can be filled with information by the computer operator, or from a cassette or Microdrive, but it always reverts to a blank when the power is switched off — as many of us know only too well.

|		
FFFF	User graphics	TOP END
(normally UDG→) FF58		
(normally RAMTOP→) FF57	GOSUB & machine stacks	
		SPARE
STKEND →	Calculator stack	
STKBOT →	Editing area & workspace	
	Variables	FLOATING RAM
VARS →		
	BASIC program	
(5CB6 on unexpanded PROG → SPECTRUM)		
	Channel information & Microdrive maps	
5CB6	System variables	
5C00	Printer buffer	
5B00	Attributes file	
5800		FIXED RAM
	Display file	
4000		
	Character set	
3D00		
386D	Calculator	
2C88		ROM
	Operating system & interpreter	
0000		

Figure 2.1: Spectrum Memory Map

The first section immediately above the ROM, is called the 'fixed RAM'. This contains divisions exploited by the ROM, at fixed addresses: a whopping great chunk to hold the display file and colour attributes for the television; a small section where material is assembled for the printer; and the very important section which holds all the addresses used by the ROM (or by us) when operating the Spectrum — the system variables.

Beyond this point, the RAM becomes even more vague, the 'floating RAM'. The sections have no predetermined length, though their addresses are always computed in the ROM and then held in the system variables.

There are sections dealing with channel information and the Microdrive. Beyond them comes any BASIC program we have written, with the variables used in that program. Finally come the locations which the Spectrum uses for dealing with a BASIC program — the workspace, editing area and calculator stack.

Beyond all this is spare space. It may be large or small, depending on how much there is in the BASIC program and its variables. Whatever spare space there is can be filled with other data or machine code. This is the area of RAM which we shall be using most frequently in order to write and execute our machine code routines.

Rounding off the RAM comes the section I have called the 'top end'. This is usually approached from the top downwards. First there is a part designed to hold user-designed graphics. It is normally the address pointed to by the system variable UDG, but you can always change UDG and make it point somewhere else (which can be very useful). Below the UDG comes an address called 'RAMTOP', with some more 'stacks' (storage spaces for numbers) below it. Any RUN, CLEAR, or NEW operation will normally clear the RAM only as far as RAMTOP (only a complete power-off will clear beyond this point) so, by moving RAMTOP down, you can reserve a patch of memory which is safe from being overwritten by a BASIC operation (see Sinclair manual p.168).

The ROM

The most interesting section of memory is undoubtedly the ROM. Everything the Spectrum does when writing or executing a BASIC program is done through the ROM. It has a program for everything — even if it is just 'Sorry, can't cope...'. In a very real sense, the ROM *is* the Spectrum.

All of these programs are written in Z80 machine code and within the main programs there are scores of self-contained subroutines, which get used as needed to carry out specific tasks. These are the goodies we are after, because they can do these same tasks for us and save us a great deal of trouble. We can poach them, like apples from an orchard!

The ones which will be of interest in the programs to be developed in this book are listed in Appendix B, but there are many more. Now you should be able to see the advantage of having to hand a good, annotated disassembly of the Spectrum ROM, making it possible to locate the section you are interested in and work through the listing to see if it can be used as a subroutine in your own programs. Much of the listing is pretty heavy-going, but you can struggle along, trying out bits here and there. At the very least, it is more fun than the average adventure game.

To illustrate how ROM routines can be used in ways which are not exactly those originally intended, here is a short program which performs an automatic SAVE of bytes. This can be very useful if you have a BASIC program which always goes hand-in-hand with a block of data. The data could be a SCREEN$ or a machine code routine — anything, as long as it is SAVEd as bytes.

This program compiles a label from material within the BASIC program — ie this program can calculate a new label, like a date or an index number, every time it is SAVEd (which can be useful too).

Saver

```
   5 DIM w$(10)
   6 DEF FN a$(x)=CHR$ INT (x/256)
   7 DEF FN b$(x)=CHR$ (x-256*CODE FN a$
(x))
  10 INPUT "Label:";w$: PRINT AT 8,0;"He
ader Label: ";w$
  20 INPUT "Start Address:";st
  30 INPUT "Finish Address:";fi
  40 LET le=fi-st
  50 PRINT AT 10,0;"SAVE bytes from ";st
;" to ";fi
  60 LET z$=CHR$ 3+w$+FN b$(le)+FN a$(le
)+FN b$(st)+FN a$(st)+CHR$ 0+CHR$ 0+CHR$
 33+CHR$ 82+CHR$ 0+CHR$ 229+CHR$ 33+FN b
$(st)+FN a$(st)+CHR$ 229+CHR$ 221+CHR$ 3
3+CHR$ 0+CHR$ 91+CHR$ 195+CHR$ 132+CHR$
9
  70 FOR j=1 TO LEN z$: POKE 23295+j,COD
E z$(j): NEXT j
  80 RANDOMIZE USR 23313
```

Because this is a demonstration program, it shows 'W$' (label), 'st' (start address) and 'fi' (finish address) as INPUTs, but you would normally expect these to be provided, or calculated, within your main program.

Everyone must have noticed, when LOADing a program to the Spectrum, that there is a kind of 'mini-program' which gets LOADed ahead of the main program. This mini-program is known as the 'header' and, as well as the program label, which it prints out, the header contains important information about the main program, which enables the Spectrum to LOAD this properly.

So, before we can look at how the Saver program functions, we need to be clear as to how the header is put together. It consists of 17 bytes, arranged like this:

0	1 2 3 4 5 6 7 8 9 10	11 12	13 14	15 16
3	LABEL	le	st	0 0

Byte 0 codes for the type of data being SAVEd: '0' for a BASIC program, '1' and '2' for arrays (numerical or character), '3' for a block of bytes. The next 10 bytes hold the program label, normally entered by hand, but here to be found in 'w$'. The header ends with three pairs of bytes, the first pair holding the length of the block to be saved (our variable, 'le') the next the start address of the block (our variable, 'st') and a final pair which, in the case of a code block, as above, are both zero.

All this material is assembled in the first part of z$, in line 60, which is POKEd into the start of the printer buffer at 5B00h by line 70. The printer buffer has been chosen because it is not in use at this point, and its use avoids having to pollute another piece of RAM which might be required for something else.

The second part of line 60, POKEd into the addresses from 5B11h (23313d) onwards, consists of some machine code instructions. Here is what they look like when they have been put into the printer buffer:

Saver Code

```
5B11  21 52 00       LD    HL,0052
5B11  E5             PUSH  HL
5B15  21 00 F0       LD    HL,F000
5B18  E5             PUSH  HL
5B19  DD 21 00 5B    LD    IX,5B00
5B1D  C3 84 09       JP    0984
```

As you can see, they consist of three addresses, two of them PUSHed on to the stack and one loaded into IX, followed by a jump to a ROM routine. To find out what they are doing, we need to look at the ROM SAVE routine.

Start of SA-CONTRL Routine

```
0970 E5                    PUSH  HL
0971 3E FD                 LD    A,FD
0973 CD 01 16              CALL  1601        OPEN LOWER SCREEN
0976 AF                    XOR   A
0977 11 A1 09              LD    DE,09A1
097A CD 0A 0C              CALL  0C0A        PRINT LABEL: "start tape..." etc.
097D FD CB 02 EE           SET   5,(IX+02)
0981 CD D4 15              CALL  15D4        WAIT FOR KEY

0984 DD E5                 PUSH  IX
0986 11 11 00              LD    DE,0011      DE CONTAINS LENGTH OF
                                              HEADER (= 17d)
0989 AF                    XOR   A
098A CD C2 04              CALL  04C2         MAIN SAVE ROUTINE
                                              (FOR HEADER)
098D DD E1                 POP   IX
098F 06 32                 LD    B,32
0991 76                    HALT                PAUSE 1 SEC.
0992 10 FD                 DJNZ  0991
0994 DD 5E 0B              LD    E,(IX+0B)
0997 DD 56 0C              LD    D,(IX+0C)     DE CONTAINS "le"
099A 3E FF                 LD    A,FF
099C DD E1                 POP   IX
099E C3 C2 04              JP    04C2          MAIN SAVE ROUTINE (FOR
                                               BYTES)
```

We don't need to go into details, but the outline of the routine should be clear from the notes. Before the routine starts, HL must hold the start address of the block to be SAVEd and IX must hold the address of the header information. The routine begins by PUSHing HL and then printing out the 'Start tape...' message in lower screen, before waiting for a key to be depressed. Once this happens, the routine stacks IX and loads DE with the fixed header length (17d = 11h). Then it CALLs the main 'SA_BYTES' subroutine at 04C2h. This subroutine will SAVE the number of bytes in the DE register, starting at IX.

Once this has been done, there is a pause. The routine then loads DE with the data block length, using the IX register to pick out the information from the header (IX points to the start of the header). It then POPs to IX the last address from the stack, which was PUSHed from HL, the start address of the block to be SAVEd. IX now holds that start address and the routine is ready to SAVE the block, by jumping to 04C2h again.

Let's look at our three addresses in the printer buffer again. The first one on to the stack will be the last one off and will form the return address for the dangling RET at the end of the second 'SA_BYTES' call. (This, you will

remember, was JUMPed into at address 099Eh.) This return address, in fact, just contains another RET — actually, the first to appear in the ROM. This has been borrowed to get us back into BASIC when the whole operation has been completed.

Next on the stack is the start address for the data block (calculated in our BASIC program by FNa(x) and FNb(x). The third is the printer buffer start address, made ready in IX.

If you check 0984h, the ROM address we jump to in the original ROM SAVE routine, you will see that our routine has skipped over all the 'Wait for a key...' information. However, all the necessary addresses are already prepared on the stack and in IX, so that the rest of the ROM routine can go ahead as planned.

To end with, here is the listing as it might appear in a program. You SAVE by 'GOTO 1100' — this will SAVE the program, followed by the CODE bytes. When you come to LOAD, it will LOAD the program and start executing it from line 1000, which immediately LOADs the bytes.

You will have to arrange to CLEAR 'bytes − 1' beforehand.

Saver — example

```
1000 LOAD ""CODE
1010 GO TO 10
1100 SAVE "something" LINE 1000
1110 DIM w$(10)
1120 DEF FN a$(x)=CHR$ INT (x/256)
1130 DEF FN b$(x)=CHR$ (x-256*CODE FN a$
(x))
1140 LET w$="somethingCODE "
1150 LET st=64256: LET le=1280
1160 LET z$=CHR$ 3+w$+FN b$(le)+FN a$(le
)+FN b$(st)+FN a$(st)+CHR$ 0+CHR$ 0+CHR$
 33+CHR$ 62+CHR$ 0+CHR$ 229+CHR$ 33+FN b
$(st)+FN a$(st)+CHR$ 229+CHR$ 221+CHR$ 3
3+CHR$ 0+CHR$ 91+CHR$ 195+CHR$ 132+CHR$
9
1170 FOR j=1 TO LEN z$: POKE 23295+j,COD
E z$(j): NEXT j
1180 RANDOMIZE USR 23313
```

This is just one example of the way in which you can bend the ROM for special purposes. I confess that it takes some courage to tackle anything

much more complicated. It can be difficult to trace the programming, even with the help of a printed disassembly of the ROM, and sometimes trial and error can be both error and trial.

CHAPTER 3
Making Bigger Characters

Printing text

The Sinclair Spectrum uses one of the more attractive and readable fonts of computer type. It uses a matrix of 8×8 bits to produce the letters, which are stored as eight bytes per letter in the ROM character set at 3D00–3FFFh.

This is quite a lavish use of bits to print characters. Many commercial matrix printers use only 5×7, but this means that they have to use some special means to move the print position along, so as to give a space between letters. In addition, the lower case letters cannot have true 'descenders'.

Descenders are the tails of letters such as 'p', 'q' and 'y', which normally hang down below the print line. On a 5×7 matrix it is hard to do this, so the manufacturers 'cheat', as in **Figure 3.1**. The result is awkward-looking and makes for poor legibility.

```
To get a fuller reply we need

rest of the original sentence

inserting a space). It is not
```

Figure 3.1

On the Spectrum, there are proper descenders and the eight-bit width means that characters can be printed side by side, while still leaving proper spaces between letters. You can see in **Figure 3.2**, where a line has been ruled through the bottom bits of the characters, that the tails of

```
gx jx px qx yx
‾‾‾‾‾‾‾‾‾‾‾‾‾
gx ix px gx yx
fx hx kx lx tx
‾‾‾‾‾‾‾‾‾‾‾‾‾
fx hx kx lx tx
```

Figure 3.2

q, y, p, q and j actually cut into the bottom line. The 'ascenders' of the letters, t, h, f, k and l just graze the top line.

Before we start seeing how we can play around with the Spectrum character set, perhaps it would be as well to take a quick look at how printing is actually done in the Spectrum.

The key to nearly all Spectrum printing is the 'Restart 10' instruction, in the Z80 codes. This single opcode, 'D7', is used to lead into the main ROM PRINT routine. This routine controls all the Spectrum printing operations, including the setting of colour and other attributes, print position, and so on (see Appendix B).

(The printing of numbers is another matter. This involves placing the value of the number on the calculator stack, in five-byte floating point form, from which it can be picked and printed with decimal point, or in 'exponent' notation. This is done by a ROM routine starting at 2DE3h. However, for our present purposes, the 'RST 10' routine is the one to stick with.)

In order to use this instruction, you first have to choose what kind of printing is to be done. Usually, when dealing with a USR operation, the Spectrum will be in INPUT mode and will print to the bottom of the screen. To get the Spectrum to print to the main screen, you must open channel 'S', by using the instructions 'LD A,02: CALL 1601' (1601h is the address of the 'open channel' routine).

So, to print the letter 'A' on the screen, you need the routine:

Print 'A' ✓

```
F000  3E  02        LD    A,02
F002  CD  01  16    CALL  1601
F005  3E  41        LD    A,41
F005  D7            RST   10
F008  C9            RET
```

If you change the '02' at F001h to '03', the 'A' will be sent to the printer, instead of to the screen. If you change it to '01', the 'A' will go to the INPUT area — but you may not always see it, as the area is usually cleared as soon as the operation is completed. (You can keep the 'A' on the screen by using the BASIC commands 'RANDOMIZE USR 61440: PAUSE 0'.)

To print the 'A' in a specified position, in specified colours, we have to incorporate the appropriate control codes, found on p.183 of the Spectrum manual. To print a green 'A' on a yellow ground, at line 10, column 16d, we could do the following:

Print String ✓

```
F000 3E 02       LD    A,02
F002 CD 01 16    CALL  1601
F005 3E 16       LD    A,16    AT
F007 D7          RST   10
F008 3E 0A       LD    A,0A    10d
F00A D7          RST   10
F00B 3E 10       LD    A,10    16d
F00D D7          RST   10
F00E 3E 10       LD    A,10    INK
F010 D7          RST   10
F011 3E 04       LD    A,04    4
F013 D7          RST   10
F014 3E 11       LD    A,11    PAPER
F016 D7          RST   10
F017 3E 06       LD    A,06    6
F019 D7          RST   10
F01A 3E 41       LD    A,41    CHR$ 'A'
F01C D7          RST   10
F01D C9          RET
```

This is all very well, once in a way, but it's very long-winded and we wouldn't want to use this system to print up a lot of instructions or text in a program.

Fortunately, Sinclair Research have incorporated a string printing subroutine in the ROM, at 203Ch, which gets over most of the difficulties. You need to have the address of the string in DE and its length in BC, before calling 203Ch. The subroutine is essentially a way of looping through the string, using 'RST 10' to print each character in turn.

Our green 'A' now becomes:

Green 'A'

```
F00A 11 00 F0    LD    DE,F000
F00D 01 08 00    LD    BC,0008
F010 CD 3C 20    CALL  203C
F013 C9          RET
```

E000 = 57344
F000 = 61440

```
DEFB: -
F000 16 0A 10 10 04 11 06 41
```

The same codes for paper, print position, etc., are still there, but are now

grouped together at address F000h. (I have left out the channel selection routine to avoid clutter, but you would still have to incorporate it each time you needed to redirect the printing to the main screen.)

This coding is much more compact, but it can be taken one stage further when there are a lot of messages to be printed in a machine code program.

The first step is to add the length of the string to the front of the data, so that the DEFB (DEFine Bytes) become:

```
DEFB:-
F000  03  16  0A  10  10  04  11  06
F008  41
```

This number can be picked out by the new routine and loaded into BC at the start of the operations.

```
F00A  1A           LD    A,(DE)   FIRST BYTE OF DATA INTO BC
F00B  4F           LD    C,A
F00C  06  00       LD    B,00
F00E  13           INC   DE       POINT TO STRING
F00F  CD  3C  20   CALL  203C
```

Once the string has been printed, you can call 203Ch again, this time to print another string, which restores any colour, or other attributes, to their normal condition, so that the current attributes won't hold over into the next bout of printing, which may require something quite different. This is the final section, to carry out the 'housekeeping':

```
F012  01  0C  00   LD    BC,000C  LENGTH OF 'RESTORING' STRING
F015  11  1C  F0   LD    DE,F01C  ADDRESS OF 'REST.' STRING
F018  CD  3C  20   CALL  203C
F01B  C9           RET

DEFB:-
F01C  10  09  11  08  12  00  13  00
F024  14  00  15  00
```

You can, of course, select what you want for your 'normal' attributes when you restore them at the end.

Now when you call the subroutine you will only have to specify the address: the subroutine will read off its own LEN$, for the control loop. You can group all the messages together in a block.

Notice, by the way, that you have to respecify the whole of BC, at F012h, as it gets corrupted by the 203Ch routine.

Stretching characters

Now, nice as all this is, it doesn't do very much. No sooner have new Spectrum owners run the 'Horizons' tape, than they wonder how they can get their Spectrum to print all shapes and sizes, like the tape.

In fact, the Psion machine code routine for stretching letters is a very good one and, as it is part of the Spectrum package, it is well worth pulling out of the cassette. It's not hard to write a little BASIC program to go with the routine to implement the magnified printing as required. However, the Psion routine is quite long and complicated and suffers from being, if anything, a bit too good. The choices are sometimes too many and too complicated.

For practical purposes, within the framework of an actual program, I find that the most useful enlarged character is one twice the linear size, which looks nice and bold, but is still small enough to display a good line of print (16 characters). But there are many useful variations. Let's see how these can be implemented.

Double-sized

Letters

To produce double-sized letters like the ones above, we have to make a block of four bits grow, where only one bit grew before. This means that, to make space for the big character, we shall have to spread it over four normal-sized characters, printed in a block. **Figure 3.3** shows what it will look like. In the inverse printing, you can see particularly well how the big
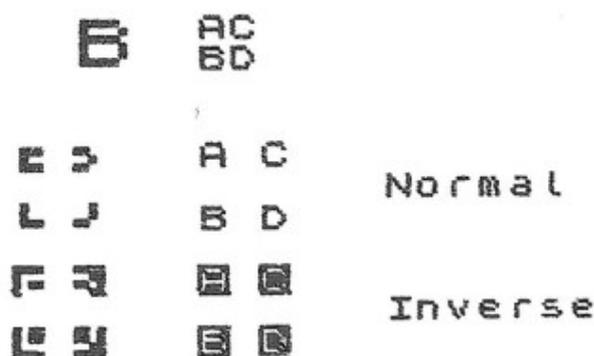
Figure 3.3: **Large Graphics Character Made up of Four Normal-sized Characters**

character is made up of four 'graphics' characters. This is in fact how the routine works. For every big character we print, we manufacture a set of four new graphics characters in UDG positions 'A', 'C', 'B' and 'D', and then print them out, as shown. (To save time, I'll be referring to the UDG positions by the names of the letters normally found there.)

In order to manufacture these graphic characters, we need a machine code routine. In principle it works quite simply.

First, we select the letter we want and look it up in the character set. Every character in the Spectrum character set is made up of eight bytes, each coding for a line of pixels on the screen. A set bit corresponds to a 'black' pixel: a zero bit corresponds to a 'white' pixel. For our purposes, we deal with the character four bytes at a time — the top half of the character first and then the bottom half.

First of all, we put the first byte of the chosen character into the A register. This is, in fact, the top line of the character. Unfortunately, the top line of the character consists only of zeros, so we will set up an arbitrary line (**Figure 3.4**) which shows the movements more clearly. (Actually, it is the second line, shifted two bits to the right.) We then PUSH bit 0 into the carry bit, by doing 'RRA' (rotate right A).

```
76543210
00011111        "A" register

RRA  >>>>

76543210
00001111  >  1  (Carry)
```

Figure 3.4

We choose the DE register pair to hold the two new bytes, which we are going to generate from the single byte in A and we shift the carry bit into the '7' position in the D register, with an 'RR D' (**Figure 3.5**). This action, of course, tips bit 0 of D back into the carry, so we pick that up again and get it into E, by doing 'RR E'.

```
76543210
00000000        "D" register

RR D  >>>>

76543210
10000000  >  0  (Carry)

RR E  >>>>

76543210
0  >  00000000  >  0  (Carry)
```

Figure 3.5

As the new character is going to be twice as thick as the old one, we need to double up on the new bit which we have taken from A. This can be done by using 'SRA D'. 'Shift right arithmetic' shifts all the bits along one place, but it also copies into the vacant position at 7 the value of the bit previously held there — which is exactly what we want. To complete the operation, we pick up the previous bit 0, which has dropped into the carry, by doing 'RR E' again (**Figure 3.6**).

```
            76543210
            10000000        "D"  register

   SRA  D  >>>>

            76543210
            11000000  >  0  (Carry)

   RR  E  >>>>

            76543210
       0  > 00000000  >  0  (Carry)
```

Figure 3.6

You can see that, after doing this eight times, we shall have copied all the bits from A as double bits into D and E. All that remains now is to copy D and E into the appropriate bytes of UDG 'A' and 'C'.

Since our new characters will be twice as deep, as well as being twice as thick, we copy D and E for a second time, into the next two bytes of UDG 'A' and 'C': these will form the second line of the character on the screen (**Figure 3.7**).

```
Original  "A"    011111100

      00111111     11110000
      00111111     11110000
```

Figure 3.7

After four of these operations, we shall have finished the top half of the original character and will have filled all the bytes of UDG 'A' and 'C'. But, by continuing with the program, we transfer operations to UDG 'B' and 'D' and fill them with the bottom half of the original character, using the same technique.

Here is the completed listing. The operations between F000h and F00Fh are concerned with getting the character INPUT at the keyboard from the system variable LAST__K, and working out the address in the character set for this character in HL. The rest of the routine generates the four new graphic characters. Notice that, when finding the address for

the UDG characters, we do it indirectly, through the system variable UDG at 5C7Bh. This allows you to select a different address, if you want to.

Double-sized Letters ✓

```
F000  21 00 00      LD    HL,0000       CLEAR HL
F003  3A 08 5C      LD    A,(5C08)      LAST_K
F006  D6 20         SUB   20
                                        GET VALUE OF CHR CODE
F008  6F            LD    L,A                INTO HL
F009  29            ADD   HL,HL    ⎫
F00A  29            ADD   HL,HL    ⎬    MULTIPLY BY 8
F00B  29            ADD   HL,HL    ⎭
F00C  01 00 3D      LD    BC,3D00       START OF CHR SET
F00F  09            ADD   HL,BC         ADDRESS OF CHR

F010  DD 2A 7B 5C   LD    IX,(5C7B)     ADDRESS OF UDG
F014  0E 08         LD    C,08
F016  7E            LD    A,(HL)
F017  06 08         LD    B,08
F019  1F            RRA
F01A  CB 1A         RR    D
F01C  CB 1B         RR    E
F01E  CB 2A         SRA   D
F020  CB 1B         RR    E
F022  10 F5         DJNZ  F019
F024  DD 72 00      LD    (IX),D
F027  DD 72 01      LD    (IX+01),D
F02A  DD 73 10      LD    (IX+10),E
F02D  DD 73 11      LD    (IX+11),E
F030  23            INC   HL
F031  DD 23         INC   IX
F033  DD 23         INC   IX
F035  0D            DEC   C
F036  20 DE         JR    NZ,F016
F038  C9            RET
```

Here are two short BASIC programs, which make use of the double-sized letters routine. One allows you to type in the letters, as on a typewriter. The second prints out a string in the double-sized letters.

Notice that both programs get the required letter into LAST_K; program 1 from the keyboard and program 2 by POKEing the system variable directly. LAST_K is a very good access point for printing techniques which need an interface between a BASIC program and

machine code. It is one of the easiest ways of picking up a character, even though it might seem a little indirect.

Type Double-sized Letters

```
100 LET x=0: LET y=0
110 PAUSE 0
120 IF CODE INKEY$=13 THEN LET x=0: LET
y=y+2: GO TO 110
130 RANDOMIZE USR 61440
140 PRINT BRIGHT 1;AT y,2*x;"██";AT y+1
,2*x;"██"
150 LET x=x+1
160 GO TO 110
```

Print String in ×2 Letters

```
 10 INPUT w$
100 LET x=0: LET y=0
110 FOR j=1 TO LEN w$
120 POKE 23560,CODE w$(j)
130 RANDOMIZE USR 61440
140 PRINT BRIGHT 1;AT y,2*x;"██";AT y+1
,2*x;"██"
150 LET x=x+1
160 NEXT j
```

Using the bones of these techniques, it's simple to devise routines which will generate tall or fat characters — characters which are twice as high, but of normal width, or twice as wide, but of normal height. It is a matter of omitting the unwanted half of the routine — either the 'double shuffle' through D and E, or the double loading of D and E into the graphics characters.

The main differences in these next two programs from the double-size routines come in the loop arrangements, as it is these which control the way in which the bits are presented for the new graphics. All three have identical opening sections to find the address of the wanted characters in the character set. I have addressed the character set indirectly, as well as the UDG, in case you want to use a character set of your own, at an address different from the Sinclair character set.

In the tall letters routine, we don't have to do any bit-shifting. We simply load each bit twice into side-by-side locations in the UDG, addressed by IX. When one UDG has been filled, the routine moves automatically on to the next one.

Tall Letters✓

```
F000  21  00  00       LD    HL,0000
F003  3A  08  5C       LD    A,(5C08)
F006  D6  20           SUB   20
F008  6F               LD    L,A
F009  29               ADD   HL,HL
F00A  29               ADD   HL,HL
F00B  29               ADD   HL,HL
F00C  ED  4B  36  5C   LD    BC,(5C36)
F010  04               INC   B
F011  09               ADD   HL,BC

F012  DD  2A  7B  5C   LD    IX,(5C7B)
F016  06  08           LD    B,08
F018  7E               LD    A,(HL)
F019  DD  77  00       LD    (IX),A
F01C  DD  77  01       LD    (IX+01),A
F01F  23               INC   HL
F020  DD  23           INC   IX
F022  DD  23           INC   IX
F024  10  F2           DJNZ  F018
F026  C9               RET
```

To generate the broad letters, we do the shift (as in the double-sized letters), but not the doubling up.

"Fatties"

```
F000  21  00  00       LD    HL,0000
F003  3A  08  5C       LD    A,(5C08)
F006  D6  20           SUB   20
F008  6F               LD    L,A
F009  29               ADD   HL,HL
F00A  29               ADD   HL,HL
F00B  29               ADD   HL,HL
F00C  ED  4B  36  5C   LD    BC,(5C36)
F010  04               INC   B
F011  09               ADD   HL,BC

F012  DD  2A  7B  5C   LD    IX,(5C7B)
F016  0E  08           LD    C,08
F018  7E               LD    A,(HL)
F019  06  08           LD    B,08
F01B  1F               RRA
```

```
F01C  CB 1A          RR    D
F01E  CB 1B          RR    E
F020  CB 2A          SRA   D
F022  CB 1B          RR    E
F024  10 F5          DJNZ  F01B
F026  DD 72 00       LD    (IX),D
F029  DD 73 08       LD    (IX+08),E
F02C  23             INC   HL
F02D  DD 23          INC   IX
F02F  0D             DEC   C
F030  20 E6          JR    NZ,F018
F032  C9             RET
```

As a final addition to these routines which use shifts and rotations, here is
a program to let you print in 'bold' letters. Bold type, among printers, is
the name for letters which have thicker strokes than normal, so that they
stand out strongly from the page. In this program, we get the same effect
by rotating each byte of the letter and then ORing it with the original byte
(see Chapter 6, Four-bit Characters Entry program). This has the effect
of doubling any bits which are set, smudging, as it were, each line of the
letter. As a change of pace, I have given the BASIC program in an
LPRINT version.

Bold Printing

```
F000  21 00 00       LD    HL,0000
F003  3A 08 5C       LD    A,(5C08)
F006  D6 20          SUB   20
F008  6F             LD    L,A
F009  29             ADD   HL,HL
F00A  29             ADD   HL,HL
F00B  29             ADD   HL,HL
F00C  ED 4B 36 5C    LD    BC,(5C36)
F010  04             INC   B
F011  09             ADD   HL,BC

F012  ED 5B 7B 5C    LD    DE,(5C7B)
F016  06 08          LD    B,08
F018  7E             LD    A,(HL)
F019  1F             RRA
F01A  B6             OR    (HL)
F01B  12             LD    (DE),A
F01C  23             INC   HL
```

```
F01D 13          INC   DE
F01E 10 F8       DJNZ  F018
F020 C9          RET
```

Print out in Bold Letters

```
100 LPRINT "Print out in ";
110 LET w$="Bold Letters"
120 FOR j=1 TO LEN w$: POKE 23560,CODE
w$(j): RANDOMIZE USR 61440: LPRINT "▓";:
NEXT j
```

The bold version of each letter is loaded, once again, into the long-suffering UDG 'A'.

And, of course, there are still further variations…

This is a line of DOUBLE BOLD!

The routine can also be adapted very simply to 'embolden' the entire screen. The reason you might want to do this is to fill in 'pinholes'. Sometimes you put together a graphics routine which is supposed to fill in a solid figure — by generating a series of curves, for example, each offset from the last by one pixel. All too often these curves don't quite overlap everywhere, leaving the pinholes. The smudging routine will usually fill them in. Here it is:

BOLD Screen

```
F000 21 00 40    LD    HL,4000
F003 01 00 18    LD    BC,1800
F006 7E          LD    A,(HL)
F007 1F          RRA
F008 B6          OR    (HL)
F009 77          LD    (HL),A
F00A 23          INC   HL
F00B 0B          DEC   BC
F00C 78          LD    A,B
F00D B1          OR    C
F00E 20 F6       JR    NZ,F006
F010 C9          RET
```

CHAPTER 4
Even Bigger Characters

To print the next size of letters ($\times$ 4, rather than $\times$ 2) calls for a rather different technique.

We could simply extend the rotating system, so as to produce four graphic characters per line rather than two. However, Sinclair have conveniently provided a complete set of 4 pixel by 4 pixel graphics. The only problem is to access the ones we want for each piece of our big character. And here, again, the way in which the Sinclair graphics are set out makes this exceptionally easy, as I'll explain.

Figure 4.1 shows the letter 'B' printed up with a grid, which breaks it up into the constituent Sinclair graphics.
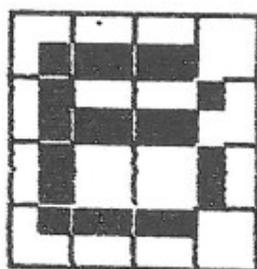
Figure 4.1

If you look at p.186 of the Sinclair manual, you will find that the graphics are spread between 128 and 143, which correspond to the hex notations '80 + 0' and '80 + F'. Now, if you number the four quarters of each character as in **Figure 4.2** you will find that the '+' number for each graphic character is always the sum of the numbers of the 'black' squares.

Figure 4.2

For example, the graphic character ⊞ (corresponding to the top

lefthand corner of the big letter 'B' above) has the code 132, which is the same as 80h + 4, and '4' is the number in the righthand bottom corner of our numbered square. Similarly, the next along in the big 'B' has code 140, which is 80 + C, where 'C' is 12, the sum of the bottom two numbers.

It's extremely easy to calculate these numbers for each graphic character (it was, of course, designed to be!). For example, if you peel off bits 7 and 8 of the first byte of CHR 'B', you get '0 0'. Doing the same for the second byte yields '0 1'. Put them together in the order in which they are peeled off (right to left and bottom to top) and they wind up as '0 1 0 0' which (surprise, surprise!) has the value '4'.

Now we can put together a machine code routine to do this automatically for each block of four bits.

The trick is to get pairs of bytes from the character set into D and E, and then, with a couple of rotate operations, first for E and then for D, slide the four required bits, in the right order, into A. All we then have to do is to set bit 7 of A (which gives the '80+x' value, which applies to the grahics) — and print it.

The full routine is given below. Once again the instructions from F000h to F011h are concerned with getting the right address into HL.

The section from F02Ch to F03Bh takes care of moving the print position down one line and four columns to the left, so that the next group of four graphic characters is printed under the last. As we haven't done this before, I'll describe how it operates: it is a useful routine to have in hand. You will still, however, have to get your BASIC program to reassign the print position for each letter.

Bigger!

```
F000  21 00 00        LD    HL,0000
F003  3A 08 5C        LD    A,(5C08)
F006  D6 20           SUB   20
F008  6F              LD    L,A
F009  29              ADD   HL,HL
F00A  29              ADD   HL,HL
F00B  29              ADD   HL,HL
F00C  ED 4B 36 5C     LD    BC,(5C36)
F010  04              INC   B
F011  09              ADD   HL,BC

F012  0E 04           LD    C,04
F014  06 04           LD    B,04
```

```
F016  56              LD    D,(HL)
F017  23              INC   HL
F018  5E              LD    E,(HL)

F019  23              INC   HL
F01A  AF              XOR   A
F01B  CB  13          RL    E
F01D  17              RLA
F01E  CB  13          RL    E
F020  17              RLA
F021  CB  12          RL    D
F023  17              RLA
F024  CB  12          RL    D
F026  17              RLA
F027  CB  FF          SET   7,A
F029  D7              RST   10
F02A  10  EE          DJNZ  F01A

F02C  C5              PUSH  BC
F02D  E5              PUSH  HL
F02E  ED  4B  88  5C  LD    BC,(5C88)    S POSN
F032  05              DEC   B            NEXT LINE
F033  0C              INC   C
F034  0C              INC   C            BACK 4 COLS
F035  0C              INC   C
F036  0C              INC   C
F037  CD  D9  0D      CALL  0DD9         SETS PR_POS
F03A  E1              POP   HL
F03B  C1              POP   BC

F03C  0D              DEC   C
F03D  20  D5          JR    NZ,F014
F03F  C9              RET
```

To change the print position, you first have to find the *existing* position. The current column and line print positions are held in the system variable S__POSN. This holds the numbers we normally use when we print 'AT y,x', in the form 33−x and 24−y (where x = column and y = line). But, to change the print position, it is not enough simply to alter these two numbers — the system variable DF__CC has to be changed in step. This last holds the address in the display file of the first byte of the character — and, as we shall find to our cost, this is not the easiest number to calculate.

However, once we have the new values for S__POSN — which are

quite easy to calculate — if we put them into BC and CALL the ROM routine at 0DD9h, this routine will calculate DF_CC and load all the system variables, as required.

This can be very useful when planning machine code printing operations. Remember, it's (24 − line) into B; (33 − column) into C and then CALL 0DD9h. Most of the registers are altered by this operation, so be sure to PUSH and POP any that you want to keep.

The last of the 'big characters' I want to deal with increases the linear dimensions eight times. This uses a complete character square for every pixel of the original, which makes a big, bold character, but one you can only use sparingly — you can only get four into a line, after all.

This character is one of the easiest to generate, as we only have to run through the bytes for the character in order and arrange to print a black square, when a set bit is found, and a white square otherwise.

It is neater to use the 'graphic space' — CHR$ 80h — rather than the normal space, CHR$ 20h. The black square is CHR$ 8Fh, so it becomes a matter of changing the second nibble only to get the results we want. In the routine below, this is done at F017h–F01Fh, using the carry, generated by an RLA (rotate left A) operation, to jump over the operation not required.

The same technique for restoring the print position is used, but now we have to move it back *eight* positions and down *one*.
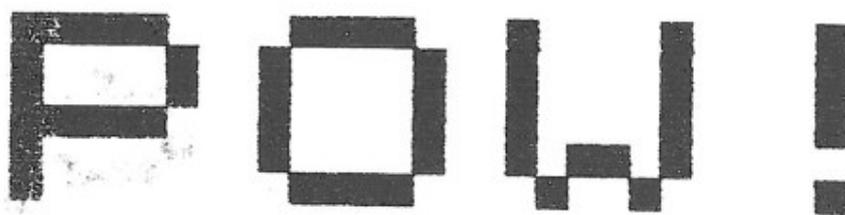

```
F000  21 00 00      LD    HL,0000
F003  3A 08 5C      LD    A,(5C08)
F006  D6 20         SUB   20
F008  6F            LD    L,A
F009  29            ADD   HL,HL
F00A  29            ADD   HL,HL
F00B  29            ADD   HL,HL
F00C  ED 4B 36 5C   LD    BC,(5C36)
F010  04            INC   B
F011  09            ADD   HL,BC

F012  0E 08         LD    C,08
F014  06 08         LD    B,08
F016  7E            LD    A,(HL)
```

```
F017 23                 INC   HL
F018 17                 RLA
F019 F5                 PUSH  AF

F01A 3E 80              LD    A,80          GRAPHIC 'SPACE'
F01C 30 02              JR    NC,F020
F01E C6 0F              ADD   A,0F          GRAPHIC 'BLACK SQUARE'
F020 D7                 RST   10            PRINT
F021 F1                 POP   AF
F022 10 F4              DJNZ  F018

F024 C5                 PUSH  BC
F025 E5                 PUSH  HL
F026 ED 4B 08 5C        LD    BC,(5C08)
F02A 05                 DEC   B             NEXT LINE
F02B 79                 LD    A,C       ⎫
F02C C6 08              ADD   A,08      ⎬   BACK 8 COLS.
F02E 4F                 LD    C,A       ⎭
F02F CD D9 0D           CALL  0DD9
F032 E1                 POP   HL
F033 C1                 POP   BC

F034 0D                 DEC   C
F035 20 DD              JR    NZ,F014
F037 C9                 RET
```

Since we are now dealing with complete print positions to build up our big
characters, there is no reason why we should not get the same effect by
using the attributes file, rather than the display file, to hold the enlarged
graphics. The routine needs very little alteration — just setting up the
address in the attributes file in DE and doing the re-addressing between
lines by a simple addition, rather than the ROM routine used for the
display file.

×8 Letters with Attributes only

```
F000 21 00 00           LD    HL,0000
F003 11 18 58           LD    DE,5818
F006 3A 08 5C           LD    A,(5C08)
F009 D6 20              SUB   20
F00B 6F                 LD    L,A
F00C 29                 ADD   HL,HL
F00D 29                 ADD   HL,HL
F00E 29                 ADD   HL,HL
F00F 01 00 3D           LD    BC,3D00
```

```
F012  09           ADD   HL,BC

F013  0E 08        LD    C,08
F015  06 08        LD    B,08

F017  7E           LD    A,(HL)
F018  23           INC   HL
F019  17           RLA
F01A  F5           PUSH  AF
F01B  3E 00        LD    A,00        INK 0; PAPER 0
F01D  38 02        JR    C,F021
F01F  3E 3F        LD    A,3F        INK 7; PAPER 7
F021  12           LD    (DE),A
F022  13           INC   DE
F023  F1           POP   AF
F024  10 F3        DJNZ  F019

F026  C5           PUSH  BC
F027  EB           EX    DE,HL
F028  01 18 00     LD    BC,0018     (32 – 8) PRINT POSITIONS
F02B  09           ADD   HL,BC
F02C  EB           EX    DE,HL       INTO DE
F02D  C1           POP   BC

F02E  0D           DEC   C
F02F  20 E4        JR    NZ,F015
F031  C9           RET
```

The fact that we are not using the display file, even though the result looks like printing, opens up some curious and interesting possibilities. If, when you have entered the machine code routine above, you can bring yourself to enter the rather shaming little program below and RUN it, with the printer hooked up, you will get a result which, though predictable, still makes you think....

More of these uses of the attributes and display file in a later chapter.

Peekaboo

```
  10 PRINT AT 3,10;"Peekaboo!"
  20 LET w$="POW!"
  30 FOR j=1 TO LEN w$: POKE 23560,CODE
w$(j)
  40 POKE 61440,(j-1)*8: RANDOMIZE USR 6
1440: NEXT j
  50 PAUSE 0
  60 COPY
```

CHAPTER 5
Sideways Characters

New character sets

There are a number of other variations on the Spectrum printing schemes which we can try out. So far, we have only considered techniques which produce letters in an *ad hoc* way, as they are needed for the display. This is perfectly adequate when the new lettering is needed only now and then, but the routines tend to be on the slow side, as they work a letter at a time, and this could be a disadvantage if you wanted to produce large parts of the display in the new lettering.

The new letterings we shall be discussing now are better adapted to use as completely new character sets, created in advance. They can be used as required, by POKEing the system variable CHARS with the new address *minus* 100h (256d). The set in the ROM can always be recovered by POKEing the same variable with its usual address, 3C00h. (The actual character set starts at 3D00h.)

One feature which all of these new sets of characters have in common is that they are all based on the existing Spectrum set: I am not suggesting that you should laboriously type in 96 or so new characters, each of eight bytes. Life is too short. The object of the present chapter will be to show you how to write programs which will generate new character sets on the basis of the old.

Sideways characters

The first altered character set I want to consider is one which uses the ordinary Spectrum letters, but prints them on their sides.

This can be very useful if you want to present results graphically, for business or scientific purposes. It is an absolute must if you have a horizontally-scrolling display, which needs to carry a title at some single location. **Figure 5.1** shows the sort of thing I mean.

In fact, the 'sideways characters' *can* be generated one at a time, as we have been doing so far. But if you have space to spare in RAM for a complete character set (it takes 300h (768d) bytes) this is much quicker and easier to operate.
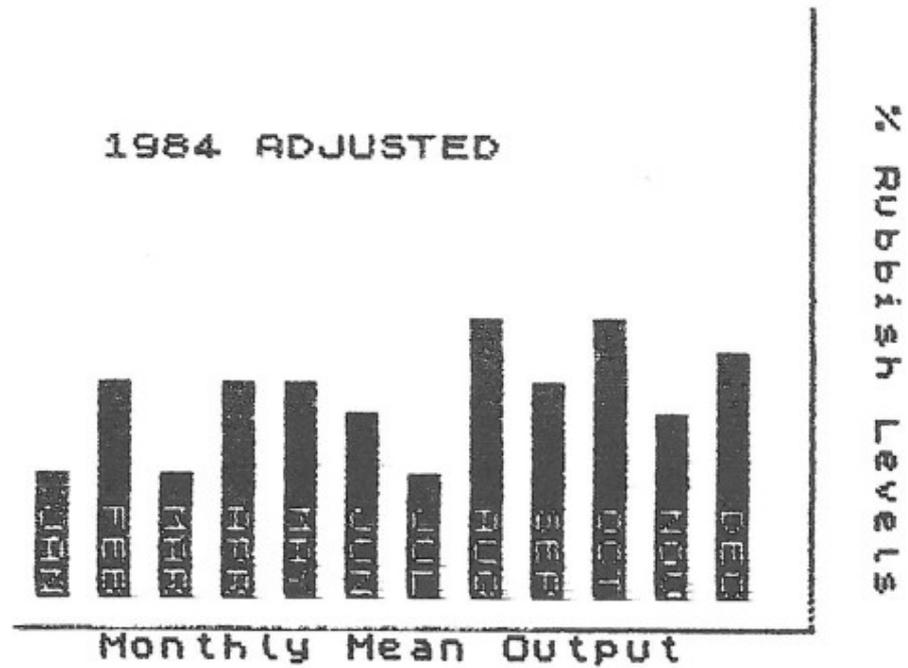
Figure 5.1:   Sideways Character Set

First of all, let's look at what we have to do and how we are going to do it.

Turning letters on their sides

As we know, a character is made up of eight bytes, each coding for a line of the printed character. Capital 'A' looks like **Figure 5.2**.
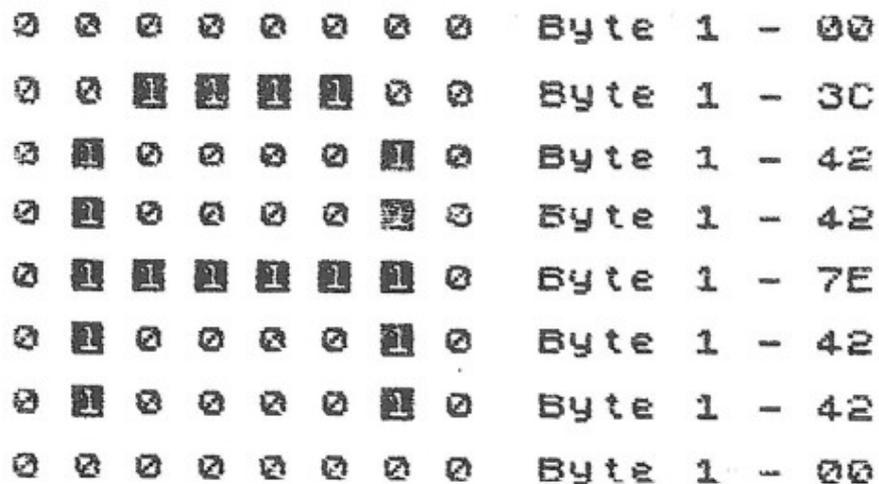
Figure 5.2:   Capital 'A'

To lay the 'A' on its side, we have to strip off the matching bits from each byte, one at a time, and re-form them into eight new bytes, which will look like **Figure 5.3**.

```
0  0  0  0  0  0  0  0    Byte  1  -  00
0  1  1  1  1  1  0  0    Byte  1  -  7C
0  0  0  1  0  0  1  0    Byte  1  -  12
0  0  0  1  0  0  1  0    Byte  1  -  12
0  0  0  1  0  0  1  0    Byte  1  -  12
0  0  0  1  0  0  1  0    Byte  1  -  12
0  1  1  1  1  1  0  0    Byte  1  -  7C
0  0  0  0  0  0  0  0    Byte  1  -  00
```

Figure 5.3:   Capital 'A' on its Side

You can see that the first byte of the new character is a line of zeros corresponding to the lefthand column of bits in **Figure 5.2**. Byte 2 in **Figure 5.3** corresponds to the second column of bits in **Figure 5.2**, and so on.

What this means, in programming terms, is that we have to rotate each of the original bytes in turn, so as to shed a bit at a time into the carry. Each time we do this, we scoop up the carry and transfer it to the new byte, which we are building up.

To do these operations, we have to have a scratch-pad: it is impossible to do any rotations, or other operations, in the ROM. So the first move is to transfer all eight bytes of the original character to a new address (I suggest MEM, the Spectrum calculator memory location, which is as handy as any). Supposing the address of our character at MEM is in HL, we can do an 'RL (HL)', followed by 'INC HL', followed by 'RRA' — and then repeat this seven more times. This will give us our new byte in the A register.

Here is the relevant listing (HL holds the address of the character in the ROM):

Single Sideways Letter

```
F010  11 92 5C       LD    DE,5C92
F013  D5             PUSH  DE
F014  01 08 00       LD    BC,0008
F017  ED B0          LDIR              GET CHR INTO SCRATCH-PAD
F019  E1             POP   HL          SCRATCH-PAD ADDR INTO HL

F01A  11 58 FF       LD    DE,FF58  UDG 'A'
F01D  0E 08          LD    C,08
F01F  06 08          LD    B,08
```

```
F021  E5          PUSH  HL
F022  CB  16      RL    (HL)
F024  23          INC   HL
F025  1F          RRA
F026  10  FA      DJNZ  F022
F028  12          LD    (DE),A          LOAD NEW BYTE INTO UDG
F029  13          INC   DE
F02A  E1          POP   HL               BACK TO START OF
                                         SCRATCH-PAD
F02B  0D          DEC   C
F02C  20  F1      JR    NZ,F01F
F02E  C9          RET
```

To do our usual 'one off' transformation, we just have to add the standard opening, which recovers the code of the character from LAST__K.

Sideways Printing

```
F000  21  00  00  LD    HL,0000
F003  3A  08  5C  LD    A,(5C08)
F006  D6  20      SUB   20
F008  6F          LD    L,A
F009  29          ADD   HL,HL
F00A  29          ADD   HL,HL
F00B  29          ADD   HL,HL
F00C  01  00  3D  LD    BC,3D00
F00F  09          ADD   HL,BC
F010  11  92  5C  LD    DE,5C92
F013  D5          PUSH  DE
F014  01  08  00  LD    BC,0008
F017  ED  B0      LDIR
F019  E1          POP   HL
F01A  11  58  FF  LD    DE,FF58
F01D  0E  08      LD    C,08
F01F  06  08      LD    B,08
F021  E5          PUSH  HL
F022  CB  16      RL    (HL)
F024  23          INC   HL
F025  1F          RRA
F026  10  FA      DJNZ  F022
F028  12          LD    (DE),A
F029  13          INC   DE
F02A  E1          POP   HL
F02B  0D          DEC   C
```

```
F02C  20 F1        JR      NZ,F01F
F02E  C9           RET
```

To create the complete character set, we need an address at which to start the new set, and we also need to make a few modifications to the routine.

It's a good plan to start the new characters at an address ending in '00'. This is because the ROM set starts at 3D00h, so, if the new address also has '00' as its second byte, we only have to alter the first byte to swap addresses. This means changing the number held at 5C37h (CHARS + 1, 23607d) in order to switch the sets. I have suggested 'E000h'.

We use the alternate registers to hold the overall count for the total number of characters in the set — 60h, or 96d — and also to handle the transfer from ROM to scratch-pad for each letter. Notice in the listing how the first action on moving into the alternate registers is to 'PUSH HL': the final action before leaving them for the last time is to 'POP HL' again. This preserves the important address held there by the Spectrum for its own business. Notice also how, after each transfer to the scratch–pad, HL conveniently points to the next letter, because it has been moved up by the 'LDIR' instruction.

Sideways Character Set

```
F000  11 00 E0     LD      DE,E000     START ADDRESS NEW CHRS

F003  D9           EXX
F004  E5           PUSH HL
F005  06 60        LD      B,60
F007  21 00 3D     LD      HL,3D00     START ADDRESS ROM CHRS

F00A  C5           PUSH BC
F00B  11 92 5C     LD      DE,5C92     SCRATCH-PAD
F00E  01 08 00     LD      BC,0008
F011  ED B0        LDIR
F013  D9           EXX
F014  21 92 5C     LD      HL,5C92     SCRATCH-PAD

F017  0E 08        LD      C,08
F019  06 08        LD      B,08
F01B  E5           PUSH HL
F01C  CB 16        RL      (HL)
F01E  23           INC     HL
F01F  1F           RRA
F020  10 FA        DJNZ    F01C

F022  12           LD      (DE),A
```

```
F023 13              INC   DE
F024 E1              POP   HL
F025 0D              DEC   C
F026 20 F1           JR    NZ,F019

F028 D9              EXX
F029 C1              POP   BC
F02A 10 DE           DJNZ  F00A        JUMP TO NEXT CHR

F02C E1              POP   HL
F02D D9              EXX

F02E C9              RET
```

Once the routine has been entered and RUN, you will create a complete new character set starting at E000h. If you write a BASIC program and incorporate the line 'POKE 23607,223' (= DFh, 100h — or 256d less than the start address for the characters) everything you print will be sideways.

But remember to incorporate 'POKE 23607,60' somewhere, or you may be stranded with some very odd listings (see **Figure 5.4**)!


Figure 5.4

One final point — the key operations are 'RL (HL)' and 'RRA'. By changing these, you alter the sequence in which the bytes are stripped and re-assembled, so that you can make the letters face in various different

directions, though always on their sides. **Figure 5.5** shows the four possible configurations, two mirror–image and two normal.

```
RL  (HL)  +  RRA    ⍵ʊⴅⴖⴑ⊣ⵊⵛⵣ
RR  (HL)  +  RRA    ⴖⴒⴤⴑ⊣ⵊⵛⵣ
RL  (HL)  +  RLA    ⍵ʊⴅⴖⴑⵏⵌⵜⵣ
RR  (HL)  +  RLA    ⴖⴒⴤⴑⵏⵌⵜⵣ
```

Figure 5.5:   Letters Facing in Various Directions
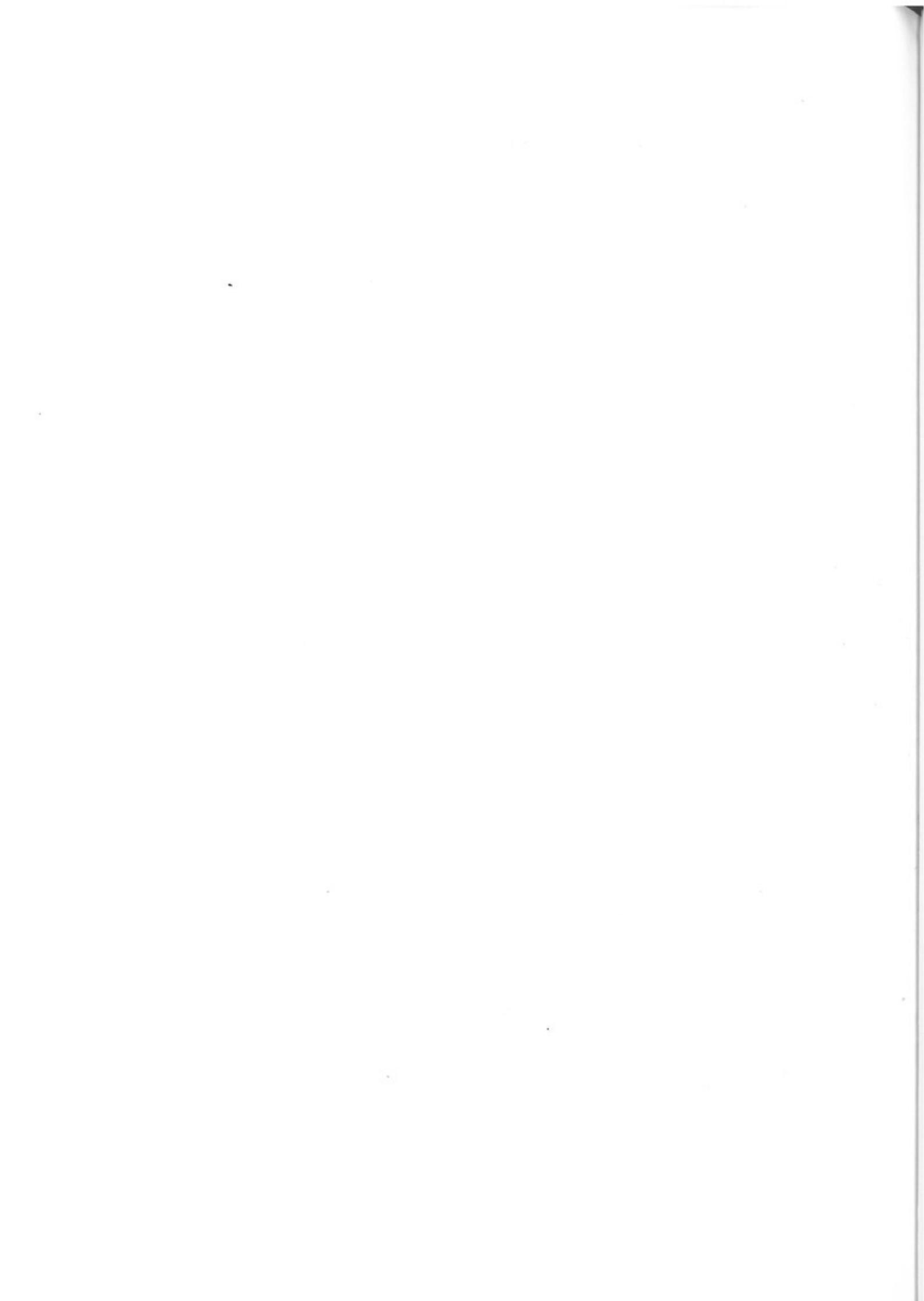
CHAPTER 6
Small Characters

As I pointed out earlier, the Spectrum character set uses a good many more bits than are absolutely necessary to produce a legible set of characters. By reducing the number of bits used horizontally to produce a character — that is, by squeezing the character sideways — we should, theoretically, be able to print more characters per line. This could be a big advantage, especially when we all get our Microdrives and have plenty of memory to splash about in.

One of the things we would like to do would be to enter sizeable chunks of text, but, even if we get them into memory, when printed out at 32 characters per line we can't get much on to the screen. It becomes more like reading a telegram than a page of print.

Six-bit characters

It is really quite simple to produce squashed-up versions of the standard Spectrum character set. You can generate the new set by picking out just six (let us say) of the eight available bits, which make up each line of a character (**Figure 6.1**).
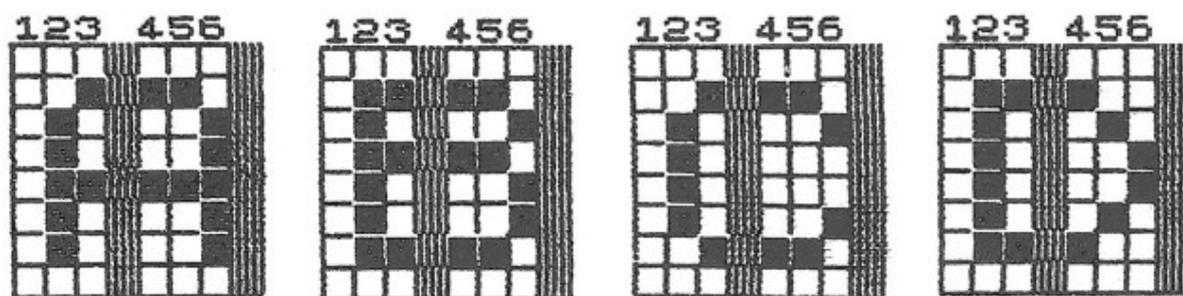

Figure 6.1

The standard last bit is always blank, anyway, so you are really only scrapping one bit per byte. **Figure 6.2** is a complete set of capital letters, generated in this way. They look quite convincing, with the exception of

ABCDEFGHIJKLMNOPQRS⁻UVWXYZ

Figure 6.2

the 'T' and the 'Y', which have suffered serious losses, and the 'I' which is lop-sided.

The machine code principle behind the making of these letters lies with our old friends, the 'rotates' and the 'shifts'. The routine gets each byte of the standard characters into the A register and then slides off the component bits, one by one, into the carry. Then it picks up the bits needed for the new character from the carry and transfers them to a waiting byte at a new address, which is addressed by HL.

6-bit Characters

```
F000  11 00 3D     LD    DE,3D00
F003  21 00 E0     LD    HL,E000
F006  01 00 03     LD    BC,0300
F009  1A           LD    A,(DE)
F00A  07           RLCA
F00B  CB 16        RL    (HL)
F00D  07           RLCA
F00E  CB 16        RL    (HL)
F010  07           RLCA
F011  CB 16        RL    (HL)
F013  07           RLCA
F014  07           RLCA
F015  CB 16        RL    (HL)
F017  07           RLCA
F018  CB 16        RL    (HL)
F01A  07           RLCA
F01B  CB 16        RL    (HL)
F01D  CB 26        SLA   (HL)
F01F  CB 26        SLA   (HL)
F021  ED A0        LDI
F023  E0           RET   PO
F024  18 E3        JR    F009
```

The automatic loading instruction 'LDI' is ideal for this, as it both 'increments' the HL and DE registers and 'decrements' the count in BC.

While the capital letters — and even the numerals — can be transformed quite effectively using this particular mix of bits, the lower case letters don't do quite as well (**Figure 6.3**): 'a' and 'b' are all right, but as for 't' and 'f' — oh dear!

I was, in fact, exaggerating a little when I said that all these characters could be generated automatically from the original Sinclair set. Most of them can be, but there will always be mavericks which will have to be adjusted individually.
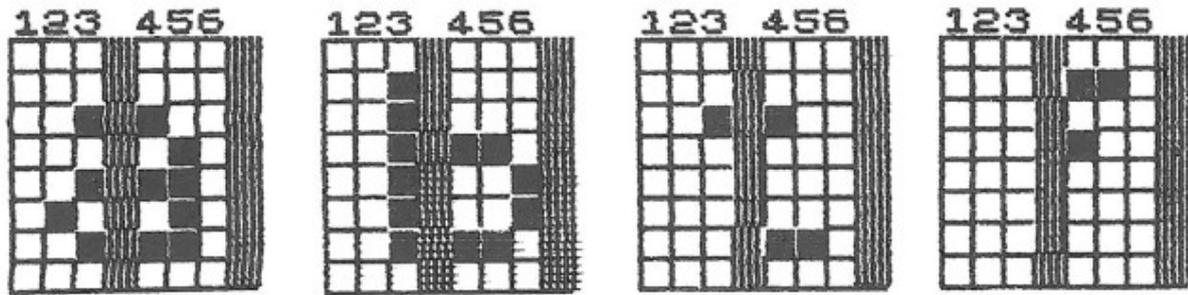
Figure 6.3

To do the adjusting, I have devised a program called Titivator which will let you refurbish any character you want to, once you have automatically generated the rough versions starting at E000h. The program will display enlarged versions of any of the new characters you choose and will let you enter the binary codes of each line, so as to build up a revised character. **Figure 6.4** shows how the display appears.
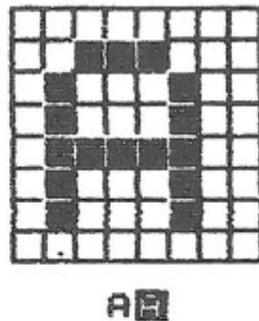
Figure 6.4

A single keystroke will display your chosen letter, enlarged and normal size (in true and inverse video) with the complete character set below that. If you want to alter a character, you can do so by keying in ENTER, when you will be able to enter each line (ie byte) in turn, in binary. (You need not enter all the terminal zeros — '0101' is the same as '01010000'.) Before writing the program, you must put the code for '×8 Letters' (at the end of Chapter 4) into a REM statement, in line 1 (this will take 54 spaces). Then fill in the rest of the BASIC program.

Titivator

```
5 POKE 23660,5
6 POKE USR "a",255: FOR j=1 TO 7: POK
```

```
E USR "a"+j,128: NEXT j
   7 POKE USR "b",255: FOR j=1 TO 7: POK
E USR "b"+j,0: NEXT j
   8 FOR j=0 TO 7: POKE USR "c"+j,128: N
EXT j
   9 INPUT "INPUT start of new character
 set";ad: LET ad=ad-256: LET ad2=INT (ad
/256): LET ad1=ad-256*ad2
  10 PRINT #1;AT 0,0;"Character?": PAUSE
 0: LET y$=INKEY$: PRINT #2
  20 IF CODE y$=13 THEN GO TO 200
  25 LET chr=CODE y$
  30 POKE 23606,ad1: POKE 23607,ad2: POK
E 23560,chr
  50 LET z$="ЙЙЙЙЙЙЙЙ"
  60 FOR j=0 TO 7: PRINT AT 2+j,11;z$: N
EXT j
  70 PRINT AT 10,11;"ВВВВВВВВ"
  80 PRINT OVER 1;AT 2,11;: RANDOMIZE US
R (5+PEEK 23635+256*PEEK 23636)
  90 PRINT AT 11,14;CHR$ chr; INVERSE 1;
CHR$ chr
 100 PRINT AT 18,0;: FOR j=32 TO 127: PR
INT CHR$ j;: NEXT j
 110 POKE 23606,0: POKE 23607,60
 120 GO TO 10
 200 DIM b$(8)
 210 FOR j=0 TO 7
 220 INPUT "Line No:";(j+1)," BIN ";b$
 230 IF b$(1)<>"0" AND b$(1)<>"1" THEN S
TOP
 240 LET x=0: FOR i=1 TO 8: LET x=x*2+(b
$(i)="1"): NEXT j
 250 POKE ad+256+8*(chr-32)+j,x
 260 NEXT j
 270 GO TO 30
```

I don't think that the BASIC program should be puzzling, but here are a few notes.

Line 5: Stops the automatic listing from getting bogged down with an 'unlistable' line 1, due to the machine code. It POKEs the system variable 'S_TOP' with a number greater than 1.

Lines 6–8: Generate the UDGs to draw a grid on the enlarged letter.

Line 10: Changes the channel so as to print on the lower screen, and then changes it back again.

Line 30: Switches to the new character set (at 57344d) and places the required character CODE in 'LAST__K'.

Lines 50–70: Print the grid.

Line 80: Gets the start of the machine code program. The address of the BASIC program, in the system variable PROG, can change, although in practice it always stays the same unless the Interface 1 is connected. Without the interface, you could make it 'RANDOMIZE USR 23760', but it is probably safer to use the indirect addressing.

Line 110: Switches back to normal characters.

Figure 6.5 shows what the six-bit character set looks like after half an hour with Titivator. It is greatly improved. I leave it to you to attend to the punctuation marks etc....

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
@123456789
```

Figure 6.5:   Six-bit Character Set

As it stands, the new set has no advantages over the old, because each letter still occupies a full eight-bit print position. We still need to reduce the spacing between the letters. This can be done, but it needs some solid machine code programming, which is better left to the next chapter. In the meantime, let's look at another character set which is, if anything, handier than the six-bit set — the four-bit set.

Four-bit characters

It may seem surprising, but you can generate letters using only a width of *three bits*, plus an extra bit for the space between letters. They may not be the prettiest letters in the world, but you can read them, and you can get a full 64 letters to the standard Sinclair line! That's as many as on a normal printed page. See **Figure 6.6**.

The beauty of this particular system is that two four-bit letters fit neatly into a single eight-bit print position. And the double character can be produced quite easily with a little ANDing and ORing, as we shall see.

But first, let's generate the characters.

This is an example of printing, using a 4-Bit Character set.
It is, really, quite readable and allows one to enter a full-
length line of text, using the normal Sinclair SPECTRUM display
facilities.

Each character uses only three Bits, horizontally, with the 4th
Bit as a standard space between letters.

A full set of characters can be generated, together with the
numerals, 0123456789, and the usual punctuation marks.

Figure 6.6: Four-bit Characters

The machine code listing is very much the same as that for the six-bit characters. However, since we have only three bits to play with, the selection of the right three bits has to be judicious. In fact, I find that it's best to take two bites (no pun intended) — the first selection to cover the lower case letters and the second for the capitals and numerals. For the other characters, you must judge for yourself.

Here is the listing which I have found best for lower case letters.

Four-bit Characters

```
F000  11 00 3D    LD    DE,3D00
F003  21 00 FC    LD    HL,FC00
F006  01 00 03    LD    BC,0300
F009  AF          XOR   A
F00A  77          LD    (HL),A
F00B  1A          LD    A,(DE)
F00C  07          RLCA
F00D  CB 16       RL    (HL)
F00F  07          RLCA
F010  CB 16       RL    (HL)
F012  07          RLCA
F013  07          RLCA
F014  CB 16       RL    (HL)
F016  07          RLCA
F017  07          RLCA
F018  CB 16       RL    (HL)
F01A  07          RLCA
F01B  ED A0       LDI
F01D  E0          RET   PO
F01E  18 E9       JR    F009
```

There are two things to note here. First, I suggest placing the new character set at FC00h: since you will be using 300h bytes, this is about as high as you can get without crashing the UDGs. You will want to save the

set once it is made, probably with the operating program, and it is more convenient to save a chunk of data up to the top of RAM.

The second small point is that I have incorporated two instructions, at F009h and F00Ah, which clear the byte you are going to use in HL. If you don't do this, you will wind up with pieces of whatever was there before, incorporated in your characters.

Figure 6.7 shows what the set looks like after the first operation, if you have got Titivator in place, with line 30 reading 'POKE 23607,251…'.

Figure 6.7

To get the capitals and numerals, switch the instructions at F018h and F01Ah, so that they read 'RLCA   RL (HL)', rather than 'RL (HL)   RLCA'. You'll also have to make line F006h read 'LD BC,0200h'.

After running the routine, your combined set should now look like **Figure 6.8**: a bit rough here and there, perhaps, but beginning to be recognisable and readable.

Figure 6.8

After a final session on Titivator, you might wind up with something like **Figure 6.9**. It will never be perfect, of course. The most difficult letters to manage are, 'H', 'M', 'N, 'W', 'n' and 'm'. There simply isn't enough information in three bits to differentiate them. You have to cheat, and trust that people's eyes will see what they expect to see — and, by and large, they do.

Figure 6.9

Here are my designs for the problem letters (**Figure 6.10**); they look very unconvincing when enlarged, but work pretty well in a piece of text. You may consider that some variations on these would look better.

Having achieved your four-bit character set and stored it safely on a cassette, you have next to consider how to use it.
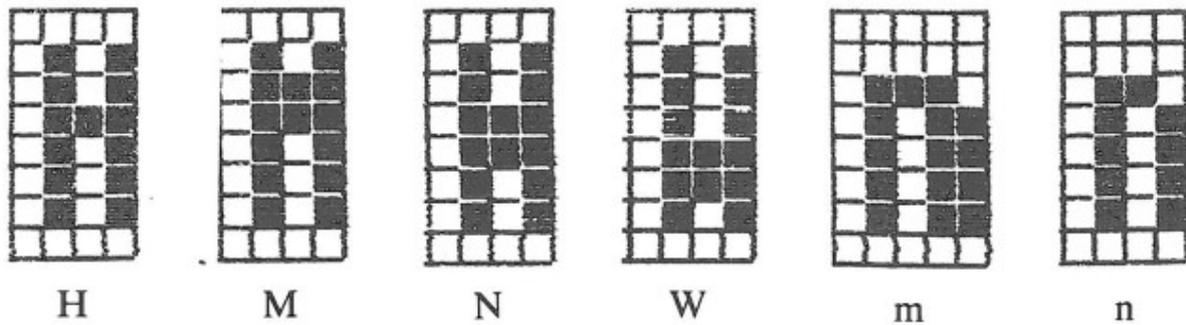
Figure 6.10: Designs for Problem Letters

There are various options. You may want to use the characters to print out text, or data, which you have stored in the RAM. You may want to use the letters as part of a system for inputting text to the RAM. You might even combine the two and have the elements of a word processor.

We'll consider only the first option here, printing out text. I've called this 'Typewriter'.

Typewriter

Here, we have to pick up letters from the keyboard, as we have done before, find their address in the character set, and then...

And then comes a difference from what we did before. We have to find some way of making a composite character out of two consecutive keystrokes. To do this, it is simplest to split the program into two parts. A subroutine, called from the main part, finds the character (and looks after a few other things, like breaking out of the program and coping with erasures and line feeds). The main routine combines the characters and prints them.

The subroutine has the most familiar material, so let's deal with it first.

Four-bit Characters — subroutine

```
F130 FD CB 01 AE   RES   5, (IY+01)
F134 FD CB 01 6E   BIT   5, (IY+01)
F138 28 FA         JR      Z,F134
F13A 3A 08 5C      LD    A, (5C08)

F13D FE 5E         CP    5E            BREAK KEY = '↑'
F13F C8            RET   Z

F140 FE 0D         CP    0D            NEW LINE
F142 28 2B         JR      Z,F16F

F144 FE 0C         CP    0C            BACKSPACE
```

```
F146  28 12        JR      Z,F15A
F148  FE 06        CP      06          CAPS SET
F14A  28 19        JR      Z,F165
F14C  D6 20        SUB     20      ┐
F14E  6F           LD      L,A     │
F14F  26 00        LD      H,00    │
F151  29           ADD     HL,HL   │
F152  29           ADD     HL,HL   ├ FIND CHR ADDRESS
F153  29           ADD     HL,HL   │
F154  01 00 FC     LD      BC,FC00 │
F157  09           ADD     HL,BC   │
F158  0C           INC     C       │
F159  C9           RET             ┘
F15A  3E 08        LD      A,08        BACKSPACE
F15C  D7           RST     10
F15D  3E 20        LD      A,20        'SPACE'
F15F  D7           RST     10
F160  3E 08        LD      A,08        BACKSPACE
F162  D7           RST     10
F163  18 0B        JR      F170
F165  3A 6A 5C     LD      A,(5C6A)
F168  EE 08        XOR     08          SET/RESET CAPS LOCK
F16A  32 6A 5C     LD      (5C6A),A
F16D  18 C1        JR      F130
F16F  D7           RST     10
F170  E1           POP     HL          DISCARD RETURN ADDRESS
F171  C3 00 F1     JP      F100
```

The first four instructions are a 'wait for a key' routine. This is a machine code alternative to the BASIC 'PAUSE 0', etc., which we used in the demonstration programs earlier. It works by first resetting byte 5 of the system variable FLAGS at 5C3Bh (23611d). As the IY register in the Spectrum normally always holds the address 5C3Ah, all the system variables can be accessed as offsets from this base.

Next, bit 5 of FLAGS is tested repeatedly. As long as no key has been struck, bit 5 remains zero, but, as soon as there is a key input, the bit gets changed to '1' and the 'JRZ' fails, so that the A register gets loaded from LAST__K (5C08h).

What we've discussed here is another handy little routine to allow you to access the keyboard.

Immediately after this, the routine tests for four special cases, which sort out four individual key codes for attention elsewhere.

5E — I have chosen as the BREAK key. It is ' ↑ ' and, by holding on to the zero flag, when the input subroutine returns to the main routine, it will cause a RETURN from the entire program.

0D — is the code for ENTER — or new line/carriage return. It simply prints itself (which gives the new line) and jumps back to the start of the main routine. This is done by POPping the normal address off the stack and doing a straight jump.

0C — is the code for DELETE. When this is matched, the routine jumps to F15Ah. Here, the routine first prints a backspace (code 8, cursor left), then prints a space, to blot out whatever was there before, and finally backspaces again, to restore the print position. The routine then jumps back to the start, using the same route as before.

06 — this, to my surprise, is the code from CAPS LOCK. The routine jumps to do precisely that — locking, or unlocking, the CAPS LOCK by an XOR operation (see later in this chapter). Then it jumps back to the start, to see what you actually want to print.

The rest of the subroutine from F14Ch, should be familiar, too. It calculates the address of the character in the new character set and returns with this information to the main routine.

The main routine CALLs the subroutine twice — once for each of the characters which make up the composite 'double character'. As so often, these characters are put together in the UDG. Here is the listing.

Four-bit Characters Entry Code — main routine

```
F100  11 58 FF       LD    DE,FF58     UDG 'A'
F103  CD 30 F1        CALL  F130        SUBROUTINE
F106  C8              RET   Z           BREAK-ROUTINE ENDS
F107  06 08           LD    B,08
F109  7E              LD    A,(HL)
F10A  07              RLCA
F10B  07              RLCA
F10C  07              RLCA
F10D  07              RLCA
F10E  12              LD    (DE),A
F10F  13              INC   DE
F110  23              INC   HL
F111  10 F6           DJNZ  F109
```

```
F113  3E  90          LD    A,90
F115  D7              RST   10              PRINT UDG 'A'
F116  3E  08          LD    A,08
F118  D7              RST   10              PRINT BACKSPACE

F119  11  58  FF      LD    DE,FF58
F11C  CD  30  F1      CALL  F130            SUBROUTINE
F11F  C8             RET   Z               BREAK-ROUTINE ENDS

F120  06  08          LD    B,08
F122  1A              LD    A,(DE)
F123  B6              OR    (HL)            COMBINE TWO CHARACTERS

F124  12              LD    (DE),A
F125  13              INC   DE
F126  23              INC   HL
F127  10  F9          DJNZ  F122
F129  3E  90          LD    A,90
F12B  D7              RST   10              PRINT UDG 'A'
F12C  18  D2          JR    F100
```

Both halves of the main routine start by getting the address for UDG 'A' into DE. Then they CALL the subroutine.

The subroutine is arranged so that it only returns with the zero flag set, if the BREAK character has been found. In all other cases, the flag is reset. The 'INC C' at F158h has no other purpose but to ensure that the zero flag is not set when the subroutine returns.

If the zero flag *has* been set, the 'RET Z' instruction ensures that we drop out of the program.

If all is well, the first part of the main program then eases our four-bit character over into the left-hand nibble and loads it into UDG 'A'. (There is, of course, nothing magical about UDG 'A'; it could equally well be UDG 'U', or any of the other UDG characters.) The program then prints the new character and immediately prints a backspace, so as to restore the print position.

In the second half of the program, which deals with the next character input, we fetch the eight bytes for the first four-bit character in DE, OR them with the new character in HL, and load the result back into DE. This is UDG 'A', which we print again. Then we jump back, to start all over again.

The effect on the screen is that of printing each four-bit character in turn. The DELETEs and 'new lines' work much as you would expect, except that the DELETE rubs out two characters at a time, because it can only deal with a complete print position.

Logical operations

The logical ORing, as carried out by the microprocessor, is a neat way of combining the two characters. Two letters, when they are ready to be brought together, look like **Figure 6.11**.
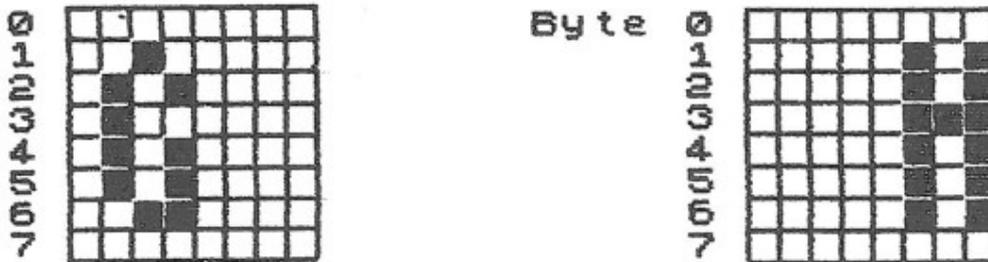

Figure 6.11

The 'G' has been shifted over to the left by the operations at F10Ah to F10Dh. It is now in the UDG, addressed by DE. The 'H' is in the new character set, addressed by HL. The A register picks up the letter 'G', a byte at a time, and compares it with the corresponding byte, held in HL.

The two first bytes aren't much use for looking at, as they are both blank, but if we take byte 1:

A holds 0010 0000   (HL) is 0000 0101

The OR operation, here, is exactly what it sounds like —the two bytes are compared and, if (HL) *or* A have a bite set in a particular position, then the final bit will be set, too. So the final byte — still in A — will be

0010 0101

This is loaded back into DE, to replace the existing character. (For the sake of interest, if we had been ANDing, then both (HL) *and* A would have to have a bit set in the same position for the final bit to be set. In this case, the final byte would have been '0000 0000'.)

When all the bytes in both characters have been ORed together, the result will look like **Figure 6.12**.

As we are discussing logical operations, we might just look at the XOR which I said, rather airily, a few pages back, was used to set, or reset the CAPS LOCK.
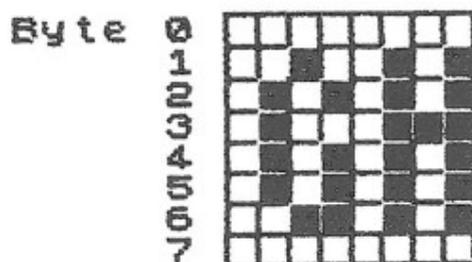
Figure 6.12

The operation in question went like this:

Set/Reset CAPS LOCK

```
F165  3A 6A 5C      LD    A,(5C6A)
F168  EE 08         XOR   08
F16A  32 6A 5C      LD    (5C6A),A
```

A collects the byte from the system variable FLAGS2 at 5C6Ah (23658d). Bit 3 of this byte controls the CAPS LOCK; it's *on* if the bit is set, *off* if it's not set. Let's assume it is not set at the moment, so the byte will look like this:

xxxx 0xxx

('x' means that we don't know, or care, what is going on in that particular bit.)

Now, we are going to XOR it with 08, which is,

0000 1000

At this point, XOR works just like an OR operation: that is, if the byte in A *or* the other byte have a bit set, the final bit will be set, too. So our final byte will be,

xxxx 1xxx

We load it back to the system variable and hey presto! WE ARE IN CAPITALS!

But now what happens when we do it again, next time round? XOR is not the same as OR: if *either* the bit in A *or* the corresponding bit is set, the final bit will be set. But *not if they are both set* — then XOR resets the bit. So when 'xxxx 1xxx' is put up against '0000 1000' and XORed, you finish with 'xxxx 0xxx' — and we are back in lower case again.

The 'x' bits always remain unaffected, because they are XORed with 0, so they will be set or reset only according to the contents of the byte in A.

You can see that the XOR operation is a classic way of switching bits on and off in the course of a program.

CHAPTER 7
Printing Six-bit Characters

Printing six-bit letters so as to display them at 40 characters to the line is rather more complex. They have to occupy three-quarters of the space, as compared to a normal eight-bit letter. The illustrations will make it clear what we are up against.

**Figure 7.1** shows the letters, as stored in the new character file.
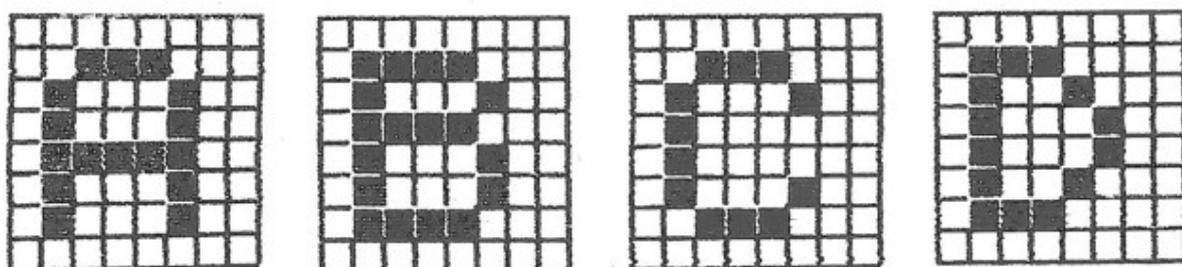
Figure 7.1

We need to rearrange a block of four letters, so that it is printed like **Figure 7.2**.
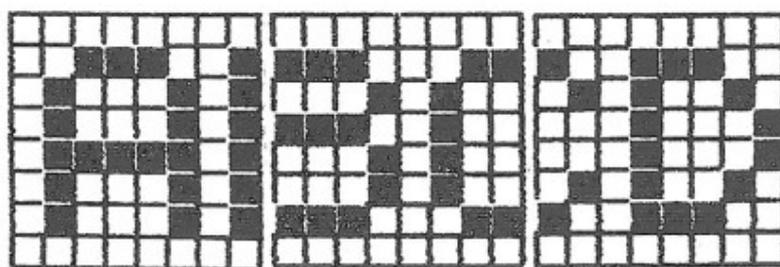
Figure 7.2

First, you must realise clearly that there is no way, on the Spectrum, that you can directly print a character offset to the left or right. The print positions are laid down hard and fast.

To print a character which is laterally offset, you have to spread the character over *two* print positions — to generate two new characters, one containing the left part of the old character and the other holding the righthand part. Then you print these, side by side — et voilà!

In the present case, you can see that the letter 'B' has been shifted to the left by two bits, the letter 'C' by four bits, and the letter 'D' by six bits.

So that we have, as it were, three composite characters holding the four original letters.

We'll look at the shifting processes required to move over the characters later, but it is plain that we shall have to organise some kind of counter to deal with the number of bits shifted — two, four or six.

Organising a counter

Let's consider this counter first. Since the routine is going to be called to cope with each letter in turn, the counter will have to be kept in a 'fire-proof' location, where we can always get at it, but where it won't be lost whenever we go back to BASIC.

One address which answers this description is the 'unused' system variable location at 5C81h (23681d). We could initialise this before we start using the printing routine, and arrange for it to have 2 added to it on each pass of the routine. The routine must also recognise when it gets greater than 6 and zero the byte again.

Here is a short piece of code, which will do that:

```
21 81 5C      LD  HL,5C81
34            INC  (HL)
34            INC  (HL)
7E            LD  A,(HL)
CB 5F         BIT  3,A
28 02         JR   Z, Next Op.
CB 9E         RES  3,(HL)
```
Next Op.

Notice how we spot that the counter has gone past '6', and how we zero it. When the routine adds 2 to a '6' in the byte addressed by HL it increases to 8, which means that it sets bit 3 for the first time and resets the lower bits in the byte. By loading A from HL and testing bit 3, we can spot when it has reached '8'. Then, by resetting bit 3 in HL, we clear the byte in the counter. We could, of course, test the bit in HL directly, but we need it in A to use it as a counter, so we might as well test it in A.

Shifting the letters

The business of getting the letters shifted into their new positions is fairly simple but laborious — like a lot of machine code.

We need three UDG characters to operate with (I have chosen UDG 1, 2 and 3 — 'A', 'B' and 'C'). Of course, in the routine we have to operate a byte at a time, but the principle shows up more clearly if we illustrate it with the complete character position of eight bytes.

If character 'A' is the first letter in the block of four and character 'B' is the second, then we first place 'A' in UDG 1 and 'B' in UDG 2. Then we start shifting 'B' to the left, with an arithmetical shift, which places a '0' in bit '0' and pushes bit 7 off into the carry. When that has been done, we scoop up the carry with a rotate left operation and transfer it into the blank UDG 3 bytes. After this has been done twice (for the whole character position), the two characters will look like **Figure 7.3**.


UDG 2   UDG 3
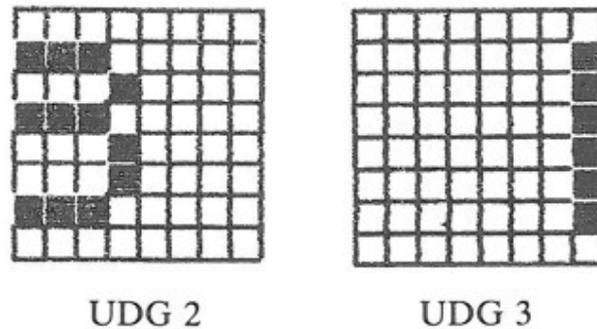
Figure 7.3

Now we can OR UDG 3 with UDG 1 (byte by byte, of course) and we have our first two characters correctly spaced in UDG 1 and UDG 2, where we can print them (see **Figure 7.4**).

UDG 1   UDG 2
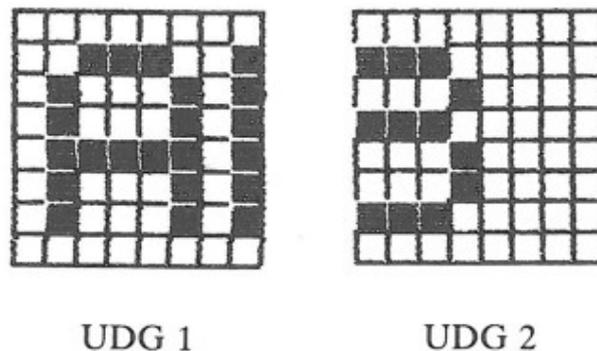
Figure 7.4

Two things remain to be done, before we can move on to the next letter in our block. First, we must backspace the print position so that it points to the place where we printed UDG 2. We shall be printing pairs of UDG 1 and UDG 2 all the time, but each time backspacing so that we pick up the former UDG 2 position.

Next, we have to get the former UDG 2 into the new UDG 1, leaving UDG 2 available for the next letter.

The beginning of the next print cycle sees the two UDGs looking like **Figure 7.5**.

Now we can shift the 'C', using the same techniques, but this time for
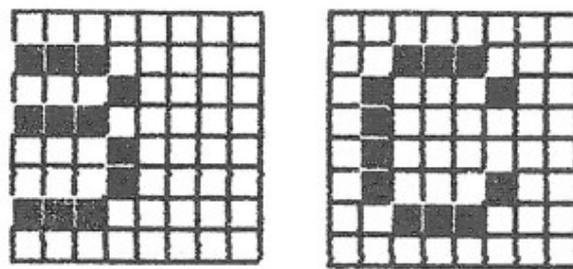
UDG 1   UDG 2

Figure 7.5

four bits, rather than two. When that is done, we adopt the same system
to deal with 'D', making a 6-bit shift, after which our printed block will be
complete and we start the cycle over again.

Here is the complete listing:

Six-bit Code Entry

```
FBA0  3A 08 5C      LD    A,(5C08)    GET ADDRESS IN NEW CHAR.-
FBA3  D6 20         SUB   20          SET FOR CHR IN LAST__K
FBA5  6F            LD    L,A
FBA6  26 00         LD    H,00
FBA8  29            ADD   HL,HL
FBA9  29            ADD   HL,HL
FBAA  29            ADD   HL,HL
FBAB  01 00 FC      LD    BC,FC00
FBAE  09            ADD   HL,BC

FBAF  11 60 FF      LD    DE,FF60  ⎫
FBB2  01 08 00      LD    BC,0008  ⎬  LOAD NEW CHR INTO UDG 2
FBB5  ED B0         LDIR           ⎭

FBB7  06 08         LD    B,08     ⎫
FBB9  AF            XOR   A        ⎪
FBBA  12            LD    (DE),A   ⎬  CLEAR UDG 3
FBBB  13            INC   DE       ⎪
FBBC  10 FC         DJNZ  FBBA     ⎭

FBBE  21 81 5C      LD    HL,5C81
FBC1  34            INC   (HL)         INC COUNT
FBC2  34            INC   (HL)
FBC3  7E            LD    A,(HL)
FBC4  CB 5F         BIT   3,A
FBC6  28 02         JR    Z,FBCA       CHECK? COUNT = 8
FBC8  CB 9E         RES   3,(HL)       JUMP IF SO
FBCA  21 60 FF      LD    HL,FF60
FBCD  20 22         JR    NZ,FBF1
```

```
FBCF   06  08        LD     B,08
FBD1   1E  68        LD     E,68
FBD3   F5            PUSH   AF
FBD4   CB  26        SLA    (HL)
FBD6   EB            EX     DE,HL
FBD7   CB  16        RL     (HL)
FBD9   EB            EX     DE,HL
FBDA   3D            DEC    A
FBDB   20  F7        JR     NZ,FBD4
FBDD   F1            POP    AF
FBDE   23            INC    HL
FBDF   13            INC    DE
FBE0   10  F1        DJNZ   FBD3

FBE2   EB            EX     DE,HL
FBE3   2E  58        LD     L,58
FBE5   06  08        LD     B,08
FBE7   1A            LD     A,(DE)
FBE8   B6            OR     (HL)
FBE9   77            LD     (HL),A
FBEA   23            INC    HL
FBEB   13            INC    DE
FBEC   10  F9        DJNZ   FBE7

FBEE   3E  90        LD     A,90
FBF0   D7            RST    10
FBF1   3E  91        LD     A,91
FBF3   D7            RST    10
FBF4   3E  08        LD     A,08
FBF6   D7            RST    10

FBF7   1E  58        LD     E,58
FBF9   0E  08        LD     C,08
FBFB   ED  B0        LDIR
FBFD   C9            RET
```

SHIFT CHR POSITION IN UDG 2/3

'OR' UDG 1 AND UDG 3

PRINT UDG 1

PRINT UDG 2

PRINT BACKSPACE

TRANSFER UDG 2 TO UDG 1

The actual sequence of operations within the routine, is that described a few pages back, but there is a certain amount of fancy footwork in the addressing, which perhaps needs some comment.

Saving bytes

The first thing to note is that most of the addresses deal with the three user-defined graphics, UDG 1, 2 and 3. All these have addresses beginning 'FF..'. In addition, these addresses are confined to the two

registers HL and DE, where they are used to transfer our characters at various stages. This means that, once these registers have been initialised, the first byte (the byte in H and D) remains unchanged so that, instead of using the three-byte instructions 'LD HL,xxxx' and 'LD DE,xxxx', we can use the two-byte instructions 'LD L,xx' and 'LD E,xx'.

Another byte-saving operation comes at FBB7h, where we clear UDG 3, in order to make it ready to receive our new character. UDG 3 starts at FF68h, and it so happens that DE holds this address at the end of the LDIR operation in the previous section. So we don't need to re-address a register in order to do the clearing operation. Another three bytes saved.

Similarly, at the end of the shift operation DE points to UDG 3. We actually want this address in HL, for the next section, so we get it there by doing 'EX DE,HL', after which we can re-address the new DE.

Again, at FBF7h, the two registers HL and DE are unaltered by the 'RST 10' operations. HL already points to UDG 2 (after the end of the previous section), so we only have to change the E register to get both of the required addresses for the final transfer.

The way we check and zero the count is worth a glance, too, as it is a slight variation on the basic method explained. The counter number is held in 5C81h. First we increase this number by 2, and get it into A. We test bit 3 of A, to see if the number has reached 8. If bit 3 is set, the number is '8' and the zero flag will be *reset* by the 'bit' operation. If bit 3 is not set, the number has not reached 8 and the zero flag will be *set*.

This zero flag is used to control two conditional jumps, which combine to route the program in the way required.

In the first case, if the zero flag is *not* set, the first jump, at FBC6h, is skipped. The routine goes on to 'RES 3,(HL)' and 'LD HL,FF60'. Neither of these operations affects the flags, although you might think that the reset operation would do so. So the zero flag is still in control and will cause a jump to 'PRINT UDG 2', at FBF1h. When we get to the final transfer at FBF7h, HL will be ready with the right address.

In the other case, where the zero flag *has* been set (ie the count is less than 8), the roles of the two jumps are reversed. The 'JR Z' at FBC6h will cause a jump to 'LD HL,FF60' at FBCAh, skipping the reset operation. Then the 'JRNZ', at FBCDh, will be ignored and the routine will continue with the character shifting operation.

This double shuffle is needed because both sections of the routine require the same address in HL, but we can't load HL before the bit test, because it holds the address of the bit we are testing! The technique does not require any extra instruction in the program, and it saves three bytes for another 'LD HL,xxxx' instruction.

All this cheese-paring over bytes is not absolutely necessary, but the savings can mount up over a long program. In the present case, it saves

about 16 bytes over a fully-addressed routine, equivalent to about 15%. The way to do this is to get the program working in the simple extended mode and then analyse it to see whether there is scope for worthwhile compression. It's unwise to try and plunge right in with the clever stuff.

Using the six-bit characters

Even though we have achieved the '40-character line', there are certain restrictions on using this technique. Although it is possible to include a new-line facility, much as we did for the four-bit routine, there isn't a really satisfactory way of doing a backspace to correct an error, because of the way the letters overlap the bytes. In fact, I cannot see much point in using this technique to produce a printout on the screen, in the typewriter mode.

The real future of the six-bit character set lies in its use for printing out labels or text. For example, if you have an address book or a telephone directory program, the entries could be printed out much more neatly and economically using the new character set. This means that you would be picking up and printing existing strings, or other data, where the lack of correction capabilities would not matter. **Figure 7.6** shows the gain in compactness.

```
This is a 32 character line ****
This is a 40 character line, in 6-BITxxx
```

Figure 7.6

To access the routine from an existing string, you would need a BASIC program something like this:

Six-bit Printing — demonstration

```
10 LET w$="This is a 40 character line
in 6-BIT***"
20 PRINT AT 9,0;w$
30 POKE 23681,6
40 FOR j=1 TO LEN w$(j): RANDOMIZE USR
64416
60 NEXT j
```

The POKE in line 30 initialises the counter at '6'. This means that the routine will print the first letter as the start of a block of four in the position of letter 'A' in Figure 7.2.

These printing suggestions would not be of much practical use if we

could not use the printer. Here again, this can be done, but you have to be slightly devious to manage it.

The trouble is that the LPRINT command, which outputs to the printer buffer, seems unable to cope with the backspacing and other peculiarities required to print our pairs of UDGs. The solution is to prepare a complete line in advance in the display file and then send it to the printer, using a modification of the COPY command.

As constituted in the ROM, the COPY routine sends the entire display to the printer, a pixel line at a time, jumping around in the display file, so as to achieve consecutive lines, in spite of the awkward arrangement of the file (see Chapter 9).

We only want to send eight lines of pixels, so the re-addressing required is minimal. We can butcher the ROM COPY routine (which starts at 0EACh) to make a simple version to pick one line off the screen and COPY it to the printer.

The line of six-bit printing has to be put on the screen somewhere, before it can be COPYed, and perhaps the most suitable place is in the lower screen, where it won't interfere with the main display. This merely involves including 'PRINT #1;' in the BASIC program given above.

The reorganised COPY routine goes like this:

PRINT Line 1 in Lower Screen

```
FF00  F3              DI
FF01  06 08           LD    B,08
FF03  21 E0 50        LD    HL,50E0    ADDR OF LINE 1 IN LOWER
                                       SCREEN
FF06  E5              PUSH  HL
FF07  C5              PUSH  BC
FF08  CD F4 0E        CALL  0EF4       CALL PRINTER SUBROUTINE
FF0B  C1              POP   BC
FF0C  E1              POP   HL
FF0D  24              INC   H
FF0E  10 F6           DJNZ  FF06
FF10  CD DA 0E        CALL  0EDA       FINAL SECTION OF COPY ROUTINE
FF13  06 02           LD    B,02
FF15  CD 44 0E        CALL  0E44       CLEAR LINE ROUTINE
FF18  C9              RET
```

It starts off with a 'disable interrupts' instruction, as does the original ROM COPY routine. We provide an eight-digit counter in B and load HL with the start address of the first line in the lower screen, which is 50E0h. The actual printer subroutine is at 0EF4h, which sends a single line from the screen to the printer buffer. 'INC H' re-addresses HL to the next line of display. The routine goes back to the ROM at 0EDAh, for the end section of the ROM COPY routine. This actually LPRINTs the line

and finalises. The last two instructions before RET are for good housekeeping. They clear the bottom two lines of the display, ready for another bout of printing.

The routine used for this purpose is the ROM CL__LINE subroutine at OE44h. This clears the number of lines specified in B, from 1 to 24, starting at the bottom of the display. It is an extremely useful supplement to the full CLS routine at 0D6Bh.

Assuming your coding is grouped as follows, we are ready for the final BASIC program.

FBA0–FBFD	6-bit print	64416d start
FC00–FEFF	character set	
FF00–FF18	COPY	65280d start

Here is the BASIC demonstration program:

Six-bit Character Printout

```
10 LET w$="This is a 40 character line
in 6-BIT***"
20 PRINT AT 9,0;w$
30 POKE 23681,6   .
40 FOR j=1 TO LEN w$(j): RANDOMIZE USR
64416
60 NEXT j
70 RANDOMIZE   USR 65280
```

If the sight of lines of text flashing on and off at the bottom of the screen as they are sent to the printer bothers you, you can add 'INK 7;' at the end of line 30.
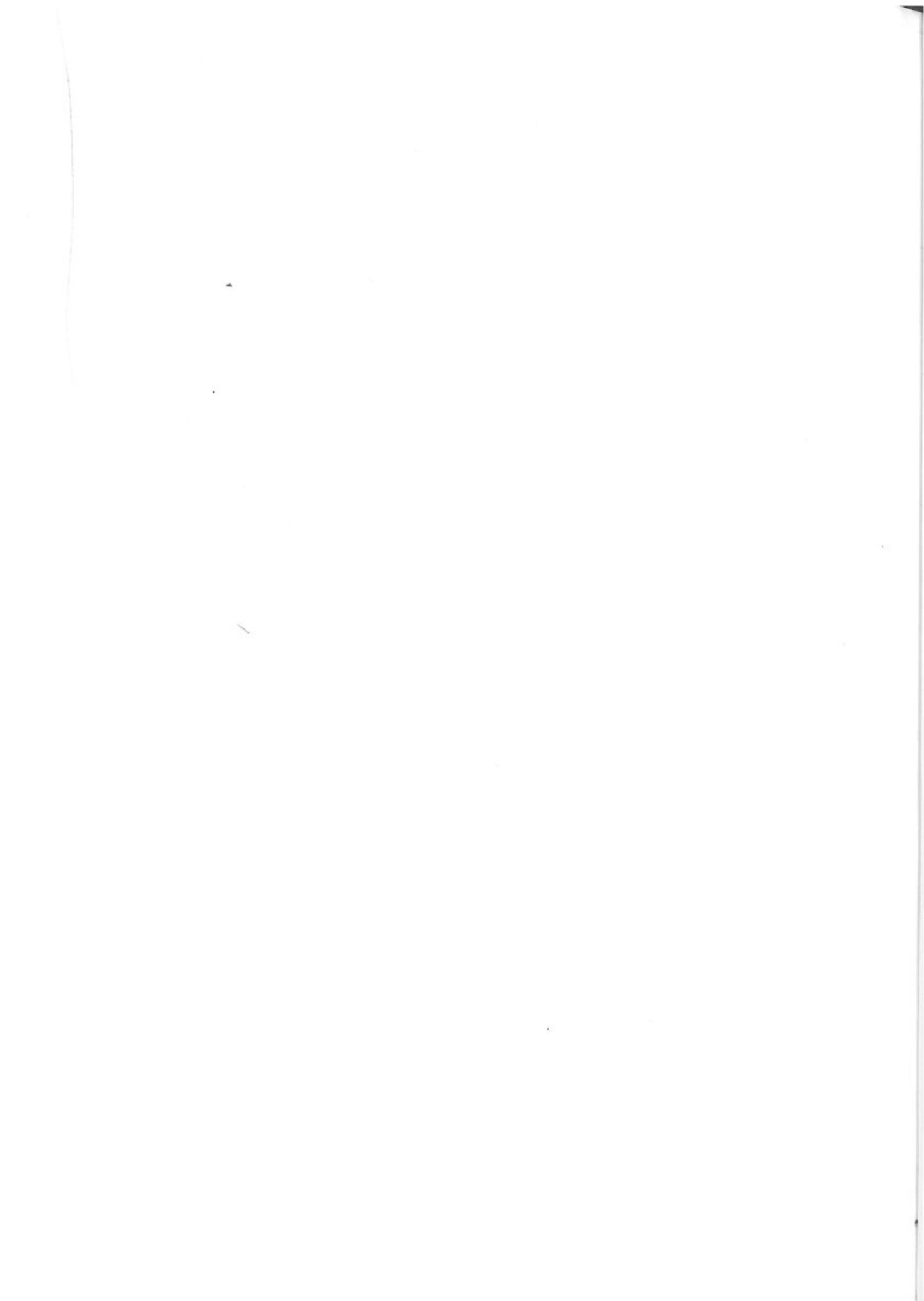
CHAPTER 8
Sprites — Animation

Everybody has to have sprites these days: a group of bytes — usually containing a graphic image — which can be moved about the screen independently of anything else on view.

The more ambitious sorts of sprite are under the control of a special chip, which looks after the coordination of the separate bytes, so that all the programmer has to do is to indicate the direction and range of the movement. The Spectrum does not have such a chip, so all 'spritely' movements have to be done under the control of software.

There are three types of movement we shall have to consider — movement within the sprite (or animation); movement about the screen; and (what may be slightly different) movement in front of, or behind, other sprites or pieces of the background.

In this chapter, we'll deal with the first type of movement, movement within the sprite.

Drawing sprites
For any kind of animation, you need to prepare a cycle of drawings, each one differing slightly from the last, so that when the cycle is run through repeatedly it looks like movement on the screen.

It is perfectly possible to animate within a single character — the UDG characters are very handy for this kind of thing, as well as for other purposes. Here is an example — a nasty little face called Snapper which I have used in games.

DATA for snapper graphics

Face 1	Face 2	Face 3	Face 4
126	126	126	126
255	219	255	255
165	255	153	153
129	165	255	153
129	129	255	255
165	165	165	102
102	102	102	60
60	60	60	0

Compile Snapper Graphics

```
SNAPPERS

    5 DATA 126,255,165,129,129,165,102,60
,126,219,255,165,129,165,102,60,126,255,
153,255,165,165,102,60,126,255,153,153,2
55,102,60,0
   10 FOR j=0 TO 31
   20 READ n: POKE USR "A"+j,n
   30 NEXT j
```

This shows what the snappers look like.

Snapper

```
100>FOR j=0 TO 3
110 PRINT AT 10,10;CHR$ (144+j)
120 PAUSE 5: NEXT j
130 PAUSE 10: GO TO 100
140 REM SNAPPERS: ☺ ☻ ☻ ☻
```

But really to expand your artistic talents, you need more than one character space — you need a sprite. Let's say a block of nine character spaces, making a 3 × 3 square.

Drawing within a 3 × 3 square has to be slightly more complicated than drawing within one character. There are a number of drawing programs available, but here is a simple one which I have set up to produce the kind of thing we want.

Draw a Sprite

```
    5 LET n=1: LET a=0
   10 LET x=12: LET y=163
   15 PRINT AT 0,(x-12)/8; PAPER 5;" "; P
APER 6;" "; PAPER 5;" "
   16 PRINT AT 1,(x-12)/8; PAPER 6;" "; P
APER 5;" "; PAPER 6;" "
   17 PRINT AT 2,(x-12)/8; PAPER 5;" "; P
APER 6;" "; PAPER 5;" "
   20 PLOT INVERSE a;x,y
   25 PRINT AT 18,0;"Draw frame No. ";n'"
3=</ 4=<\ 9=>/ 0=>\"
   30 PAUSE 0
   40 LET y$=INKEY$
   45 IF y$="z" THEN LET a=NOT a
```

```
 50 IF y$="5" THEN LET x=x-(x>0)
 51 IF y$="8" THEN LET x=x+(x<23)
 52 IF y$="6" THEN LET y=y-(y>152)
 53 IF y$="7" THEN LET y=y+(y<175)
 54 IF y$="9" THEN LET x=x+(x<23): LET
y=y+(y<175)
 55 IF y$="0" THEN LET x=x+(x<23): LET
y=y-(y>152)
 56 IF y$="4" THEN LET x=x-(x>0): LET y
=y+(y<175)
 57 IF y$="3" THEN LET x=x-(x>0): LET y
=y-(y>152)
 58 IF y$=CHR$ 13 THEN GO TO 100
 60 GO TO 20
```

When the program is run, the cursors will print a line, a pixel at a time, right, left, up and down. Keys '3', '4', '9' and '0' make the diagonal lines, as shown.

Lines 15, 16 and 17 print a check in pale blue and yellow, which helps you to find your whereabouts when plotting. The $(x > 0)$, $(y < 175)$, etc., is to stop the dots running off the square. These expressions are worth 1 if they are true or 0 if they are not true; so they will only change the values of x or y if either x or y are within the defined limits. In line 58, '13' is the code for ENTER, which ends this part of the program.

Keying 'z' INVERTs the PLOT command (line 45). This means that operating the cursors *rubs out* the pixels already written. This goes on until you press 'z' again.

Now you can draw your sprite. It's a help, sometimes, to have it sketched out in advance on a scrap pad. Once you have completed your sprite to your satisfaction, the next thing to do is to arrange to store it somewhere. At the moment, it only has a precarious life in the display file and on the screen.

Here is a program to store your sprite, byte by byte, in the upper RAM, starting at address 61440d. There is nothing magic about this address, but it has the convenience of being F000h, which means that the LSB (least significant byte) is zero. Since you will want to add to this number, it helps to start at the bottom of the ladder.

Store Sprite in Upper RAM

```
  5 LET n=1: LET a=0: LET q=61440
100 FOR k=0 TO 2
110 FOR i=0 TO 2
120 FOR j=0 TO 7
```

```
130 POKE q+j+8*(i+3*k),PEEK (16384+32*k
+256*j+1)
140 NEXT j: NEXT i: NEXT k
150 LET q=q+72
160 LET n=n+1: IF n<5 THEN GO TO 20
```

The program is very short, but it brings in four new variables, which work like this. The start address of each sprite group is pointed to by 'q', which is increased by 72 on each pass (line 150). The loops controlled by 'k', 'i' and 'j' select each byte in the group in order — 'j' picks the eight bytes in the character space, 'i' picks the column position and 'k' the line position.

The '256' tied to 'j' in line 130 is the result of the way in which the display file is organised in the Spectrum (see the next chapter). All the first bytes of the top eight lines of the display are scanned first, then the second bytes, and so on (see p.164 of the Spectrum manual). This may answer a deep-felt need in the Spectrum ROM, but it can make life a misery for programmers, especially in machine code. You are always having to add 256 and remember if you are in line 8 or 9.

The variable 'n' just controls the number of sprites we are generating. I have picked four. Three is the absolute minimum for a reasonable animation cycle, but four makes it much smoother.

Enter this program and MERGE it with the previous one. When you come to RUN it, you will get a chance to design and store four sprites. The second part of the program, which stores the sprites, takes a little time to execute — wait for the next number to appear before using the cursor again.

In this program, the current cursor position and the previous pattern are preserved. Most animation consists of *modifying* a design, leaving some of it unchanged: you can do this by using the 'rub-out'.

If you want to have a blank screen each time, change line 165 to 'GOTO 10'.

A routine to view the finished graphics is quite easy to devise and will fit comfortably into most programs. The trick is to change the system variable UDG at 5C7Bh (23675d), so that it points to the start of each of the sprites in turn. The respective characters will then appear in the UDG positions, 'A,B,C,D,E,F,G,H,I'.

Now you can appreciate the advantage of having a start address at F000h. Each sprite begins 72d bytes on from the last, which corresponds to 48h. So the LSBs of the start addresses become 0, 72, 144 and 216 (00, 48, 90 and D8 in hex). These are all less than 256, so you do not have to alter the MSB.

Here is a demonstration program in BASIC, which will do all this: 'ad1' is the LSB of the address we have been talking about and 'ad2' is the MSB.

Display Animation

```
170 PRINT AT 20,0;"Any key shows animat
ed sequence": PAUSE 0
180 CLS
190 PRINT AT 20,0;"""O""" aborts"
200 LET ad1=0: LET ad2=240
210 FOR j=0 TO 3
220 POKE 23675,ad1: POKE 23676,ad2
230 PRINT AT 10,14;"ABC";AT 11,14;"DEF"
;AT 12,14;"GHI"
240 PAUSE 5: LET ad1=ad1+72
250 IF INKEY$="O" THEN STOP
260 NEXT j
270 GO TO 200
```

This program has been laid out in the form of a loop, controlled by 'j'. However, when used in an actual program — a game, or something like that — you might want to 'POKE ad1' with the appropriate values and print out the sprite at four separate points of your program, so as to distribute the printing evenly through the main loop of your game. In that case, of course, you would drop the 'PAUSE 5' shown in line 240 — a game program would probably supply more than enough delay!

You can obviously print the block of characters anywhere you like and you can easily arrange for the print position to move about under a player's control.

To illustrate what you can do, I have drawn out a little animated cycle of a matchstick man running (**Figure 8.1**). I have done it on squared paper, where each square in the graph corresponds to eight bytes, or one character (see next chapter).

Note that the man's body moves forward four pixels (quarter print position) each frame — this means that at the end of the cycle he is ready to start again with the whole sprite moved forward one print position. Obviously, the foot on the ground stays in the same position, but the other foot moves forward through the cycle to come down in the corresponding position, when the sprite moves forward.

If you draw this action for yourself and incorporate it in a BASIC program with a FOR...NEXT loop, which moves the print position one place to the left each time round, you'll be surprised how lifelike it turns out to be. Don't be put off if the drawings are not dead accurate and you
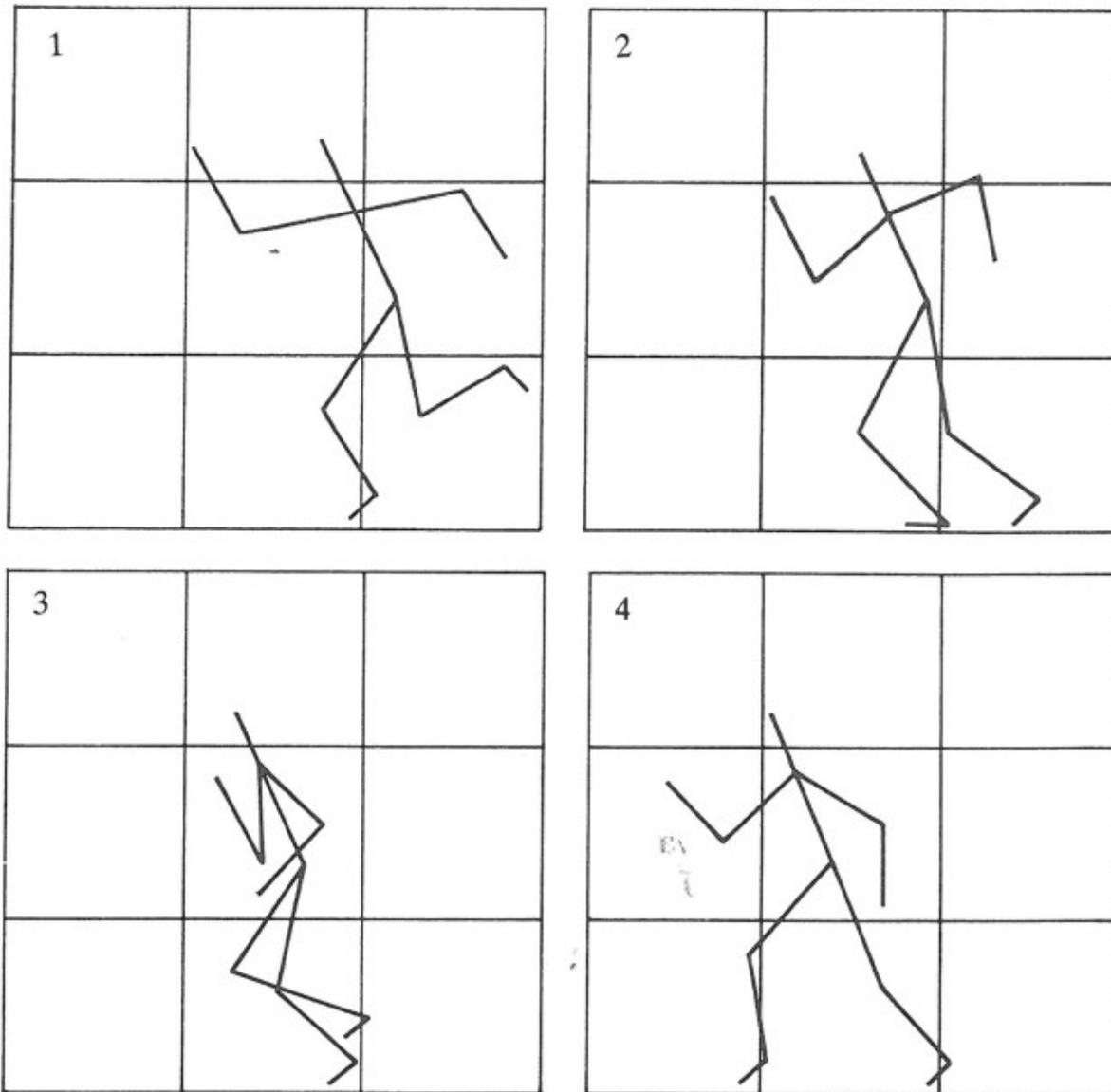
Figure 8.1

make some mistakes. It doesn't seem to show when in action — the odd wiggle often gives it an extra touch of character!

CHAPTER 9
The Moving Sprite

When using animation within a sprite you can generally do most of the programming in BASIC. Using the print position as a unit of movement is usually perfectly satisfactory (as in the running man example in the previous chapter).

However, when it comes to moving a sprite bodily about the screen, things are rather different. What makes a first-class moving sprite so appealing is that it moves smoothly, one pixel at a time.

Display file layout

Using BASIC, there is no w y that you can print to screen in other than the normal print position. But, to move a sprite, we want to be able to print it starting on *any* line of pixels, and at any column of pixels. This means that we have to abandon the Spectrum's programmed print instruction and shoot the bytes directly into the display file.

In itself, this is not very difficult.

I have read many descriptions of the layout of the Spectrum display file, but I still find it hard to visualise. Here is a variation, which I have found as helpful as any.

Think of the display as being in three immensely long lines, each of 256 characters. These will eventually form the three horizontal thirds of the TV screen. Each of the long lines begins at a separate display file address: 4000h, 4800h and 5600h.

Every character in the long line is made up of eight bytes — eight pixel lines — and each of the long lines is stored in the display file, one after the other. So a character in position 3 (as in **Figure 9.1**) will be formed from a byte from address 0 + 3, with below it a byte from address 256 + 3; then a byte from 512 + 3, and so on.


1 pixel line
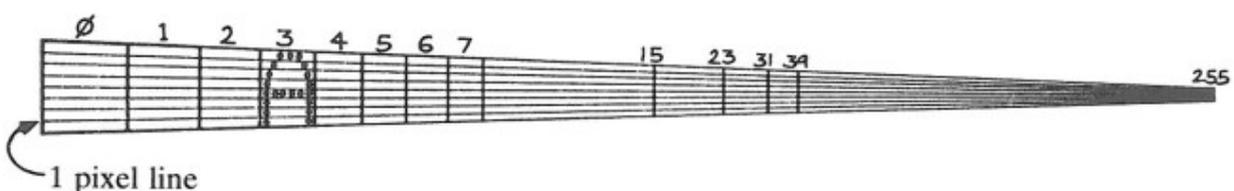
Figure 9.1: One of the Three Imaginary 255-Character Lines

(You may have spotted that the address of each of these bytes takes the form of 'x003', 'x103', 'x203', etc. In fact, to address all the eight bytes which make up a character with its address in HL, we do 'INC H' eight times, to get the eight addresses.)

The actual Spectrum screen is, of course, 32 characters wide. To make the screen, each of the long character lines (eight pixels thick) is 'cut up' to make eight screen lines, each of 32 characters (**Figure 9.2**). But the addressing of bytes on the screen remains the same as it was in the original 256-character lines.
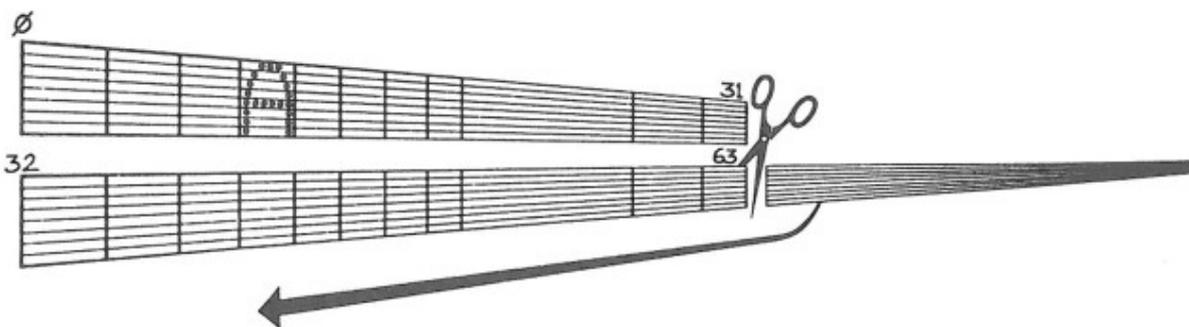

Figure 9.2: **The Character Line is 'Cut up' into 8 32-Character Lines**

You can see that, in normal printing, finding the address of a character and printing each of its bytes in the right position can be done quite neatly.

But suppose that we don't want to start printing on a pixel line 0 — what will happen then? If we want to place the first byte of our character on pixel line 4, say, what do we do about the positions of the other bytes?

The first four bytes go into place easily enough, using the system outlined above. But then we get to the bottom of our long character line, and we still have four bytes in hand. Where do they go? There must be *something* under this print position (unless it is the very bottom of the screen) but what can it be?

The answer is that, because of the 'cutting up' of the long lines, the next print position is at the *top* (pixel line 0) of the same long character line, but 20h (32d) characters long (see **Figure 9.3**)! So if, for example, we are dealing with an address in DE, we would take away 700h (to get back to pixel line 0), decrementing D, and add 20h to E.

If we want to cross over from one long line to the next, we have to add 0800h (2048d) to the base address.

As you can imagine, writing a program to do all this — and then augmenting it to deal with a sprite of nine characters — with loops within loops within loops all needing to be kept track of, has the registers PUSHing and POPping like a bowl of breakfast cereal!

But cheer up! Rescue is at hand! Sinclair Research have done all the work and have included their elegant result in the Spectrum ROM. There
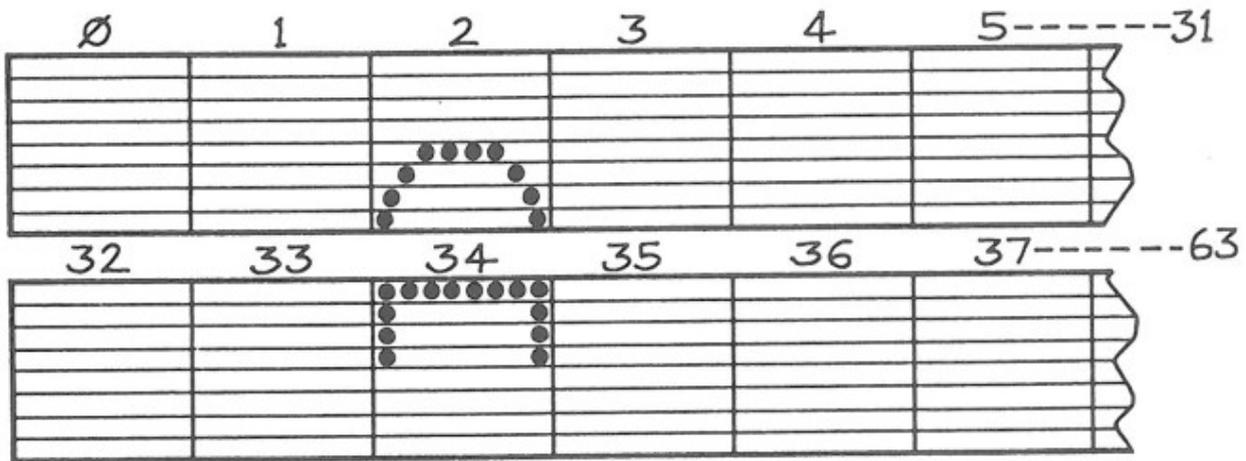
Figure 9.3:   Printing a Character across Two Different Character Squares

is a subroutine at 22AAh, called PIXEL__AD, which does éverything we could ask for.

This routine is used when executing PLOT and finding POINT. We take the x and y coordinates of a pixel position and put them into the BC register pair. Then we call PIXEL__AD, and back comes the address in HL to which the byte must be loaded, so as to get it into the display file at the correct position. Better than that, the A register contains a number which indicates the position *within* that byte of the pixel we have addressed.

So HL contains the exact pixel line (the y coordinate) plus the x axis print position (the 'coarse' x coordinate), while A contains the exact pixel position (the 'fine' x coordinate).

(Before we go on to see how we can use these goodies in practice, you may be interested to take the figures we've been dealing with a little further. Each of the long character lines we started with contains 256 characters, each made up of eight bytes. As each byte contains eight pixels, this means that each of our character lines contains $256 \times 8 \times 8 = 16,384$ pixels. As the screen is made up of three of these character lines, this gives us $3 \times 16,384 = 49,152$ pixels on the Spectrum display screen, every one of which can be individually addressed.)

Printing a vertically offset character

Let's deal with the simplest case first. Suppose we print a single character to the screen, using a routine which selects any pixel line (y coordinate) we choose.

The principle is to get the coordinates of the top lefthand corner of the character into BC, CALL PIXEL__AD and load the first byte of the character into the address held by HL. We then move to the next byte of the character, *decrease* the y coordinate by one (the y coordinates are read from the *bottom* of the screen to the *top*) and call PIXEL__AD

again, before printing the next byte. And so on, through all eight bytes of the character.

Obviously, this is a case for a control loop, to count the eight bytes, and the neatest loop is the one operated by DJNZ. This requires a control number in B, but we already have to use BC for the pixel address. We can't easily store the register with a PUSH, because we really need to *exchange* the registers for use at different points. So that is what we shall do — we'll use the alternate registers and access BC'.

Here's the program.

Print 'A' at Pixel Coordinates 88,172

61440

```
F000  01 58 AC      LD    BC,AC58      COORDINATES
F003  ED 5B 7B 5C   LD    DE,(5C7B)    ADDRESS OF UDG IN
                                        SYST. VARS
F007  D9            EXX
F008  06 08         LD    B,08         BYTE COUNT
F00A  D9            EXX
F00B  C5            PUSH  BC
F00C  CD AA 22      CALL  22AA         CALL PIXEL_AD
F00F  1A            LD    A,(DE)
F010  77            LD    (HL),A
F011  13            INC   DE
F012  C1            POP   BC
F013  05            DEC   B
F014  D9            EXX
F015  10 F3         DJNZ  F00A
F017  D9            EXX
F018  C9            RET
```

Notice that BC prime holds AC58 at the start — this is hex for 172, 88. Also, at line F003h we have, once again, pointed DE to the address for UDG in the system variables, at 5C7Bh, so that if you wanted to re-address the graphics to another address by altering UDG, the routine would still work. This could be useful for animation.

Here is a BASIC (very basic) program to demonstrate the routine:

Off-line Printing — demonstration

```
 10 PRINT "XXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXX"
 20 RANDOMIZE USR 61440
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

The line of Xs gives a reference line and you can see how the graphic 'A' has been offset downwards.

Printing a horizontally offset character

The next thing to consider is how to offset the graphic character to one side. So far, we have only accomplished an up-and-down movement. We have to tackle this in a rather different way.

As I pointed out in Chapter 7, there is no way, on the Spectrum, that you can directly print a character offset to the left or right, as the print positions are laid down hard and fast.

To print a character which is laterally offset, we have to spread the character over *two* print positions.

As before we shall need to use the carry — literally to carry the bits spilled off the end of one byte, as we move it over, into the adjacent byte of the second character.

Each rotation will move the character over one bit (one *pixel*) so, to take up our chosen position, we shall need to use the number left in A after CALL PIXEL__AD (the exact pixel position) to control the number of rotations.

Here is the coding.

Shift Character to Right by Number in A

```
F020  01 AC 5B    LD    BC,5BAC    CO-ORDS.
                                   CALL PIXEL__AD
F023  CD AA 22    CALL  22AA       NO OF DISPLACEMENTS INTO "C"
F026  4F          LD    C,A

F027  A7          AND   A
F028  C8          RET   Z          CHECK FOR ZERO SHIFT

F029  21 58 FF    LD    HL,FF58    UDG "A"
F02C  11 60 FF    LD    DE,FF60    UDG "B"
F02F  06 08       LD    B,08       BYTE COUNT

F031  7E          LD    A,(HL)
F032  1F          RRA              ROTATE 1st CHR
F033  77          LD    (HL),A
F034  1A          LD    A,(DE)
F035  1F          RRA              ROTATE 2nd CHR
F036  12          LD    (DE),A
F037  23          INC   HL
F038  13          INC   DE
F039  10 F6       DJNZ  F031
```

```
F03B  0D         DEC   C
F03C  20 EB      JR    NZ,F029
F03E  C9         RET
```

You can see that, after supplying the start address in BC, we call PIXEL__AD at once and get the magic number from A into C. We have lost interest in the start address for the present, so we can afford to re-use BC for this new purpose. We are also not interested in the new address supplied in HL, so we use HL and DE for the addresses of the first two user-defined graphics, 'A' and 'B': the 48K Spectrum places these at FF58h and FF60h.

The two instructions 'AND A' and 'RET Z' are safety nets. ('AND A' tests to see whether A is zero: if it is, we don't need to do any rotation, so we jump out of the subroutine. If this test were not there, the subroutine would cycle through 256 times!)

The inner loop, controlled by B, does the rotation for each pair of bytes in the characters in turn. The outer loop, controlled by C, repeats this operation the required number of times (in this case, four).

Before running the demonstration program, you should make UDG 'B' a blank, by POKEing 0 to all eight bytes. Then try this:

Off-column Printing — demonstration

```
  10 PRINT "XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXX"
  20 RANDOMIZE USR 61472
  30 PRINT AT 1,10;"AB"
  40 PRINT AT 2,10;"A B"
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                 A
                 F ╕
```

You can see that UDG 'A' appears to have been printed in a position midway between the Xs. In fact, it is not one, but *two* characters, as you can plainly see in the third line of the printout, where there is a gap between the two halves.

Printing a sprite

Armed with the results of these experiments, we are now ready to tackle the case of the sliding sprite. But before we start scaling up the routines, there is a bit of preparatory work to do on the sprite itself.

In the first place, we shall have to make it *four* bytes wide, rather than three, even though the graphics will remain 3 × 3. This is to allow for the extra byte to take up the slack in a horizontal move, as in the previous routine.

Secondly, life is much easier if we rearrange the bytes of the sprite, so that we can have all the bytes for each horizontal row side by side — rather, in fact, as Sinclair have arranged the display file in the Spectrum! **Figure 9.4** illustrates what I mean. This shows the way in which the graphics are arranged at the top of the RAM.
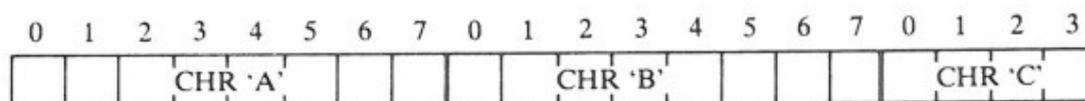

Figure 9.4

We shall rearrange them as in **Figure 9.5**.
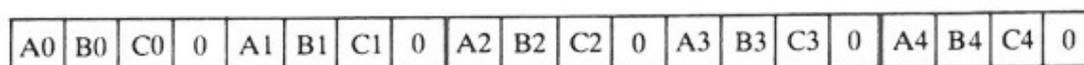
Figure 9.5

This manoeuvre means that we can load the bytes into the display file in one easy movement, rather than hop eight bytes for every operation. Better still, we can do the rotate operations by running down the complete line (like a zip fastener!).

The shuffling routine is quite simple, though there has to be a bit of a hiccup after every eighth move, when we have completed one set of three characters and have to move on to the next set.

You have to choose an address to which to copy this new arrangement —another scratch-pad — and, once again, I suggest using the printer buffer, just below the system variables. It will be free when we need it, it is fixed and unaffected by the Microdrive, etc., it saves a bit of spare RAM, and it leaves the UDGs unchanged.

Rearrange First Nine UDG in PR_BUFF

61696

```
F100  21 00 5B    LD    HL,5B00    PRINTER BUFFER
F103  E5          PUSH  HL         SAVE ADDRESS
F104  AF          XOR   A      ⎫
F105  47          LD    B,A    ⎪
F106  77          LD    (HL),A ⎬  CLEAR PRINTER BUFFER
F107  23          INC   HL     ⎪
F108  10 FC       DJNZ  F106   ⎭
```

```
F10A  D1              POP    DE              RESTORE ADDRESS
F10B  2A 7B 5C        LD     HL,(5C7B)  UDG ADDRESS
F10E  06 03 (06)      LD     B,03
                                              3 × GROUPS OF CHRS
F110  C5              PUSH   BC
F111  0E 08           LD     C,08         8 × BYTES PER CHR
F113  06 03           LD     B,03         3 × CONSECUTIVE CHRS
F115  E5              PUSH   HL

F116  7E              LD     A,(HL)
F117  12              LD     (DE),A
F118  C5              PUSH   BC
F119  01 08 00        LD     BC,0008
F11C  09              ADD    HL,BC          BYTES LOOP
F11D  C1              POP    BC
F11E  13              INC    DE
F11F  10 F5           DJNZ   F116

F121  13              INC    DE
F122  E1              POP    HL
F123  23              INC    HL             SELECT NEXT CHR
F124  0D              DEC    C
F125  20 EC           JR     NZ,F113   CONSECUTIVE CHRS LOOP
F127  0E 10           LD     C,10         ('B' IS ALREADY 0)
F129  09              ADD    HL,BC        SELECT NEXT GROUP
F12A  C1              POP    BC
F12B  10 E3           DJNZ   F110         GROUP LOOP.
F12D  C9              RET
```

The routine starts by clearing the printer buffer (by printing 0 throughout) — we want to have a clean scratch-pad. Then it sets up the three loops, of three, eight and three, and executes them, incrementing the source address and the destination address each time and adding 16d to the source address when it has finished one set of three characters. (It adds 16d, because it is already at position 8 — the end of the character —so that it only needs to hop over two complete characters to get to the start of the fourth.)

Here is a printout of the first 64 bytes in the printer buffer, after this operation. The values of the bytes are not important but you can see clearly how they have been arranged in groups of four, with a blank at the end.

```
5B00  00 00 00 00 3C 7C 3C 00
5B08  42 42 42 00 42 7C 40 00
5B10  7E 42 40 00 42 42 42 00
```

```
5B18  42 7C 3C 00  00 00 00 00
5B20  00 00 00 00  78 7E 7E 00
5B28  44 40 40 00  42 7C 7C 00
5B30  42 40 40 00  44 40 40 00
5B38  78 7E 40 00  00 00 00 00
```

After these preliminaries, the next move is to rotate the bytes on our scratch-pad, so as to shift the characters into the position we want. Continuing with our listing, it looks like this:

Shift Sprite to the Right by Number in A

```
F12D  01 AC 5B        LD    BC,5BAC      PIXEL COORDINATES
F130  C5              PUSH  BC           SAVE
F131  11 00 5B        LD    DE,5B00      PR_BUFF ADDRESS
F134  CD AA 22        CALL  22AA
F137  4F              LD    C,A
F138  A7              AND   A
F139  28 0D           JR    Z,F148       SKIP IF COUNT = 0
F13B  D5              PUSH  DE
F13C  06 60 (co)      LD    B,60    ⎫
F13E  1A              LD    A,(DE)  ⎪
F13F  1F              RRA           ⎬   ROTATE WHOLE PR_BUFF CHRS
F140  12              LD    (DE),A  ⎪
F141  13              INC   DE      ⎪
F142  10 FA           DJNZ  F13E    ⎭
F144  D1              POP   DE
F145  0D              DEC   C
F146  20 F3           JR    NZ,F13B      REPEAT OPERATION
F148  C1              POP   BC           RESTORE COORDINATES
F149  C9              RET
```

You will appreciate that the listing is very similar to the Shift Character to Right listing given earlier in this chapter. If anything, it is rather simpler, as the character bytes have been rearranged in a more accessible order.

When the entire program is run, the same section of scratch-pad will look like this:

```
5B00  00 00 00 00  03 C7 C3 C0
5B08  04 24 24 20  04 27 C4 00
5B10  07 E4 24 00  04 24 24 20
5B18  04 27 C3 C0  00 00 00 00
5B20  00 00 00 00  07 87 E7 E0
5B28  04 44 04 00  04 27 C7 C0
```

```
5B30  04 24 04 00 04 44 04 00
5B38  07 87 E4 00 00 00 00 00
```

Because our chosen coordinates (which were put into BC at address F12Dh) involve a shift of four pixel positions, you will see that the same hex numerals now appear neatly shifted into the next nibble.

The next thing to arrange is for the shifted sprite to be inserted into the display file, so that it appears on the horizontal pixel line we have specified.

The routine is, again, very similar to the Shift Character to Right listing.

Print Sprite to Screen

```
F149  DD 21 00 5B   LD    IX,5B00   PR_BUFF CHRS

F14D  D9            EXX
F14E  06 18         LD    B,18       COUNT FOR 3 × 8 PIXEL LINES
F150  D9            EXX

F151  C5            PUSH  BC
F152  CD AA 22      CALL  22AA       PIXEL_AD
F155  06 04         LD    B,04       COUNT FOR SPRITE WIDTH

F157  DD 7E 00      LD    A,(IX)
F15A  77            LD    (HL),A
F15B  23            INC   HL
F15C  DD 23         INC   IX
F15E  10 F7         DJNZ  F157

F160  C1            POP   BC
F161  05            DEC   B
F162  D9            EXX
F163  10 EB         DJNZ  F150

F165  D9            EXX
F166  C9            RET
```

You may notice that I have used the IX register to hold the scratch-pad address, rather than the obvious DE. This is because the indexing capability is going to be needed later (see Generating Hybrid Characters routine in Chapter 10).

In the meantime, if you set up a BASIC program like this:

Moving Sprite

```
10 FOR j=0 TO 50
```

```
20 POKE 61742,64+j: POKE 61743,64+j
30 RANDOMIZE USR 61696
40 NEXT j
```

you can send your sprite sailing about the screen, like a cloud in the sky. In fact, if you like to draw a cloud and get it into the UDG characters, as I have described, you can do exactly that.

CHAPTER 10
Sprite Backgrounds

If you have been pushing your sprite about the screen, you may have spotted the drawback to the routines as we have developed them so far. If there is any other printing on the screen, it is obliterated by the sprite, which always appears on a white rectangle (**Figure 10.1**).
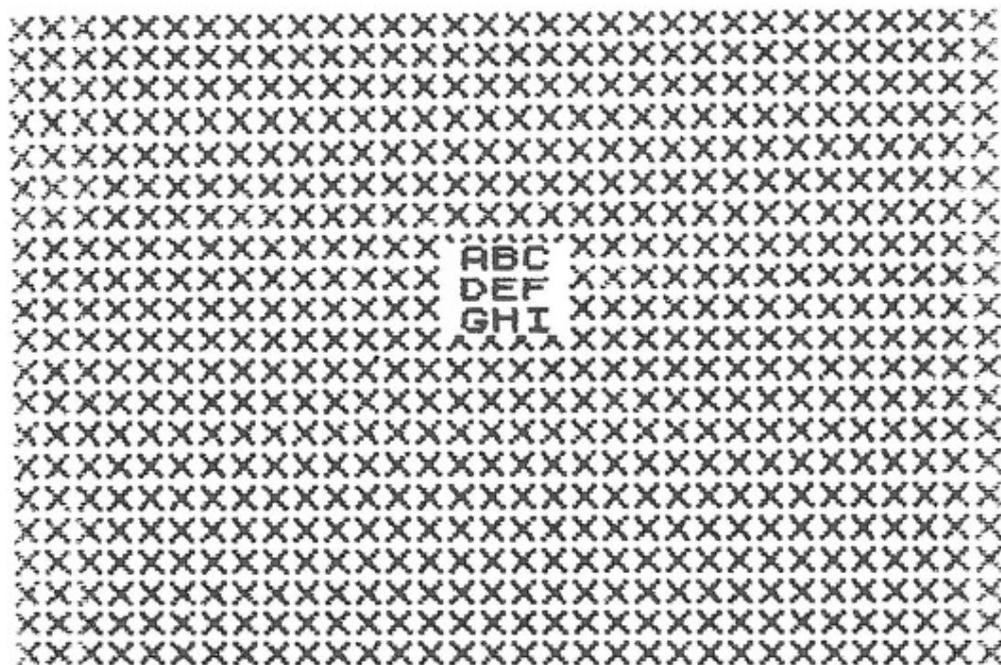

Figure 10.1

Moreover, if you move the sprite about it has a kind of hoovering action — it simply wipes out the background in its way (**Figure 10.2**).

Protecting the background
It is not too difficult to restore the wiped-out background, as the sprite moves. You store a replica of the display file containing your background in a convenient part of the RAM and call it back each time you print the sprite in a new position. This 'screen dumping' technique is another extremely useful piece of machine code, which can be used for a number of purposes, though it means sacrificing a sizeable piece of RAM to hold the replica display file.
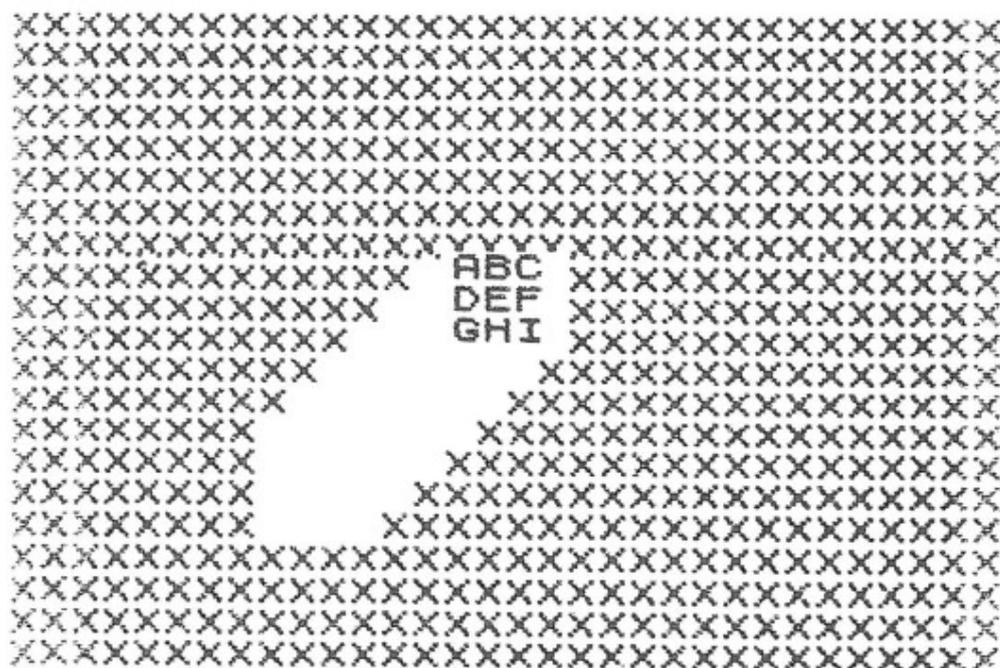
Figure 10.2

The coding could hardly be simpler: it is a straightforward LDIR operation.

Screen Dump

```
F000  21 00 40     LD    HL,4000
F003  11 00 D0     LD    DE,D000
F006  01 00 1B     LD    BC,1B00
F009  ED B0        LDIR
F00B  C9           RET
```

This stores the display file at addresses D000h to EAFFh. To call it back on to the screen needs a little more of the same:

Screen Dump — restore

```
F010  21 00 D0     LD    HL,D000
F013  11 00 40     LD    DE,4000
F016  01 00 1B     LD    BC,1B00
F019  ED B0        LDIR
F01B  C9           RET
```

The dumping/restoring routines operate so fast that they are almost instantaneous. In fact, you can use them to 'blink' on and off a line of text, or a design — simply saving the background by dumping it and then alternately printing the text and restoring the screen.

Free-floating sprites

But this still leaves the sprite on its white rectangle. Instead of having a spaceship zooming about the screen, you have a postage stamp of a spaceship, which is somehow less appealing.

In order to get the sprite floating free against any background, you have to find a way of printing that background right up to the edge of the sprite, no matter what shape your sprite is or where it is placed on the rectangle which carries it.

It is not as difficult as it might sound. In fact, we use a technique which is at least as old as the movie industry — and that is about 80 years old.

Let me tell you about the first motion picture Box Office Smash. It was made in 1903 and is called 'The Great Train Robbery'. It runs for eleven minutes. The story is simple: there's this train with a lot of bullion on it, these robbers jump into the van where it is being held, while the train is on the move, and they rob it. The big scene comes when you see the robbers overpower the guard and make off with the loot, while you can see the countryside flashing by through the open door of the van.

The point I want to make is that there was no countryside when this was filmed — the door was filled with a blank of black cardboard. Then the film was re-exposed from a moving train through another piece of black cardboard, with a hole cut in it, exactly matching the blank door in the studio set of the van. This was the invention of the matte shot!

The technique (or one much like it) is still used today on television, where it is called 'Chromakey'. It is much favoured on news bulletins, where readers can appear against a background of starving children or exploding bombs, while they are still sitting at their desks. You can usually spot the use of it by the fact that a fuzzy line appears round the foreground figures, as though drawn with a blue felt-tip pen.

To achieve a matte process by any technique requires four basic elements. They are shown in **Figure 10.3**.

A and C are the background and the foreground. B is called the 'positive matte' and D is called the 'negative matte'.

To make the composite picture, A and B are first combined, using B to mask off (or matte out) the space for the foreground. Then C is inserted into the blank space, using the negative matte, D, to mask the background area (**Figure 10.4**).

In movies, this is done by physically combining the pieces of film in an optical printer, but, to get the same effect on a computer, we can use the logical instructions in the central processor to mask out bits as required, and add others in their place.

Here is a routine to generate a 'hybrid' character in the user-defined graphics. It takes the first UDG characters, 'A' and 'B', uses a matte set
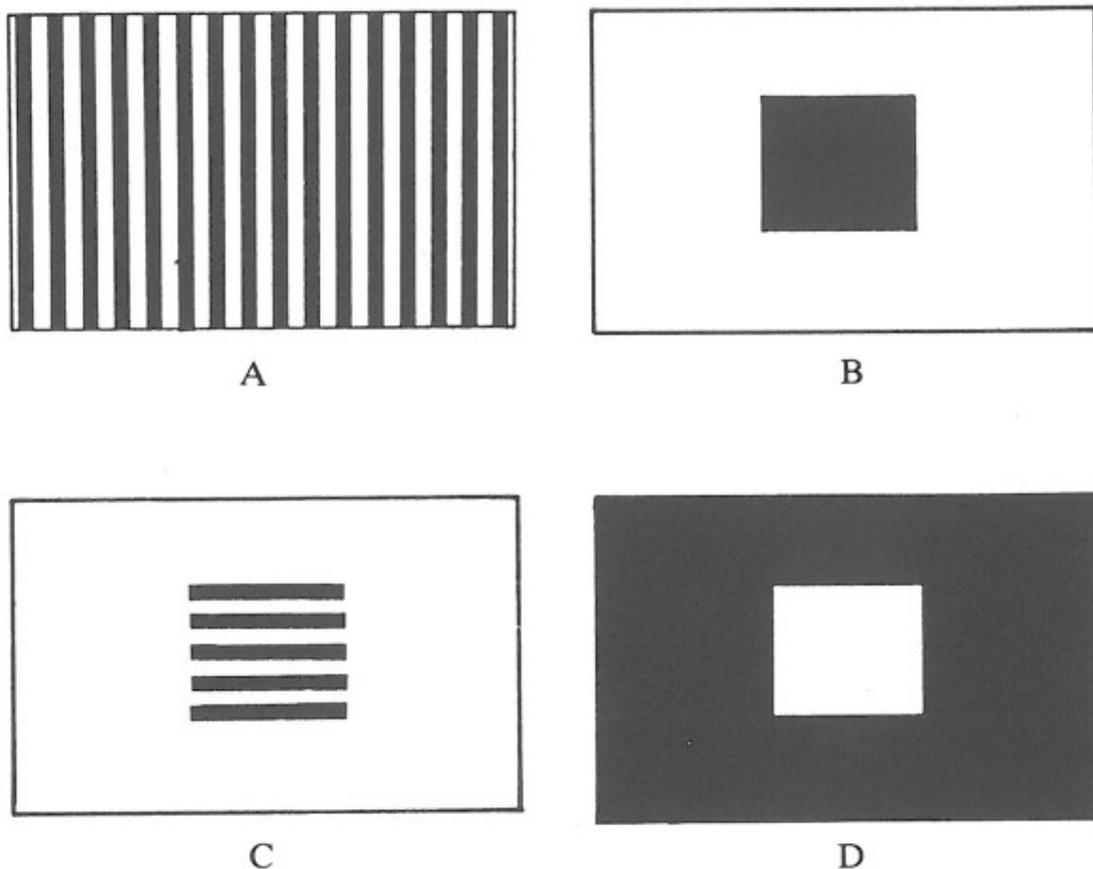
Figure 10.3:   Four Elements of Matte Process
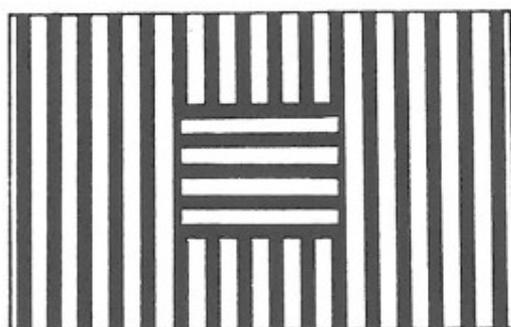
Figure 10.4:   Composite Picture

up in 'C' (actually, the same as CHR$ 133, the fifth graphics character) and puts the finished hybrid into UDG 'D'.

Generate Hybrid Characters

53248

```
D000  DD 2A 7B 5C   LD    IX,(5C7B)   UDG ADDRESS IN SYST. VARS.
D004  06 08         LD    B,08
D006  DD 7E 10       LD    A,(IX+10)   CHARACTER 'C'
D009  2F             CPL
D00A  DD A6 00       AND   (IX)        CHARACTER 'A'
```

```
D00D  4F                LD      C,A
D00E  DD 7E 10          LD      A,(IX+10)   CHARACTER 'C'
D011  DD A6 08          AND     (IX+08)     CHARACTER 'B'
D014  B1                OR      C
D015  DD 77 18          LD      (IX+18),A   CHARACTER 'D'
D018  DD 23             INC     IX          INCREMENT BASE
                                            ADDRESS
D01A  10 EA             DJNZ    D006
D01C  C9                RET
```

```
 10 LET n=0
 20 FOR j=0 TO 7: POKE USR "C"+
j,15: NEXT j
 30 RANDOMIZE USR 53248
 40 PRINT TAB 10;"A B C D"
```

In the BASIC program to demonstrate this, line 20 generates the matte. The combined character shows up more plainly in the second printout, where striped characters have been substituted for UDG 'A' and 'B'.

It's worth analysing how the routine works. D000h gets the starting address of UDG into the IX register. D004h sets a loop control of eight, corresponding to the eight bytes per character we have to deal with. D006h loads the A register with the first byte from the third character, the matte. The next instruction makes the *negative* matte; CPL will invert all the bits of the byte held in A. D00Ah blots out half the bits in UDG 'A' by ANDing the two bytes. D00Dh stores the result in the C register. D00Eh collects the matte for a second time and, in D012h, uses it as a *positive* matte to mask off the opposite half of UDG 'B', at IX + 10. D014h combines the two halves by ORing them and D015h loads the new byte into UDG 'D'. Then we move the address in IX up one byte and go back to do the same thing seven more times.

You can see from this routine the advantage of being able to use the indexed register IX. It allows you to address the same relative positions of all the four elements we are handling, as the program moves through the eight bytes, and you only have to increment the base address.

We shall use the same technique in addressing the sprite and its matte.

Printing sprites on the background
Expanding the routine to deal with the block of 12 bytes means just the minimum juggling with loops. To ensure that we always have access to a 'clean' background — one without sprite images already printed on it —

we make a copy, before running the routine, of the complete display file at a new address in the RAM. We use this copy as the source of the background for the composite sprite. The copying can be done with the screen dump routine, which I have placed at F180h. This loads the display file copy to a block of 1800h bytes, starting at D800h and ending one byte before F000h.

We can always find the address in the copy file corresponding to the sprite address in the actual display file, by adding a constant offset of 9800h to the display file address. This offset is placed in DE in the first instruction below:

Print Sprite with Background

```
F149  11 00 98     LD    DE,9800      OFFSET FOR D/FILE COPY
F14C  DD 21 00 5B  LD    IX,5B00      SCRATCH-PAD
F150  D9           EXX
F151  06 18        LD    B,18
F153  D9           EXX
F154  C5           PUSH  BC
F155  CD AA 22     CALL  22AA         CALL PIXEL_AD
F158  06 04        LD    B,04
F15A  E5           PUSH  HL
F15B  19           ADD   HL,DE        GET D/FILE COPY ADDRESS
F15C  DD 7E 60     LD    A,(IX+60)    MATTE TO 'A'
F15F  2F           CPL                MAKE NEGATIVE MATTE
F160  A6           AND   (HL)         MASK BACKGROUND
F161  4F           LD    C,A
F162  DD 7E 60     LD    A,(IX+60)    MATTE TO 'A'
F165  DD A6 00     AND   (IX)         MASK SPRITE
F168  B1           OR    C            MAKE COMPOSITE
F169  E1           POP   HL           GET ORIGINAL D/FILE ADDRESS
F16A  77           LD    (HL),A
F16B  23           INC   HL
F16C  DD 23        INC   IX
F16E  10 EA        DJNZ  F15A
F170  C1           POP   BC
F171  05           DEC   B
F172  D9           EXX
F173  10 DE        DJNZ  F153
F175  D9           EXX
F176  C9           RET
```

Dump D/File to D800h (51200d)

```
F180  21 00 40     LD    HL,4000
```

```
F183  11  00  D8        LD    DE,D800
F186  01  00  18        LD    BC,1800
F189  ED  B0            LDIR
F18B  C9               RET
```

You may be wondering what an offest of 'IX + 60' is doing in the routine above. This is, in fact, the address of the matte. UDG characters ABCDEFGHI form the sprite and I have allocated JKLMNOPQR for the matte. Each group uses 60h (96d) bytes. This means two small alterations to the earlier routines we have developed:

F10E   LD B,03   06 03   becomes   LD B,06   06 06   (p.75)

and

F13C   LD B,60   06 60   becomes   LD B,C0   06 C0   (p.77)

Both numbers are doubled.
    Putting it all together, here is a test program and result (in **Figure 10.5**).

Graphic characters:

Sprite

Matte

Figure 10.5

Matted Sprite — demonstration

```
  10 FOR j=0 TO 703: PRINT "X"; :
NEXT j
  20 RANDOMIZE USR 61824
  30 POKE 61742,97: POKE 61743,9
7
  40 RANDOMIZE USR 61696
```

And, of course, your sprite need not be solid, Making the central square of the matte a blank, gives the result in **Figure 10.6**.
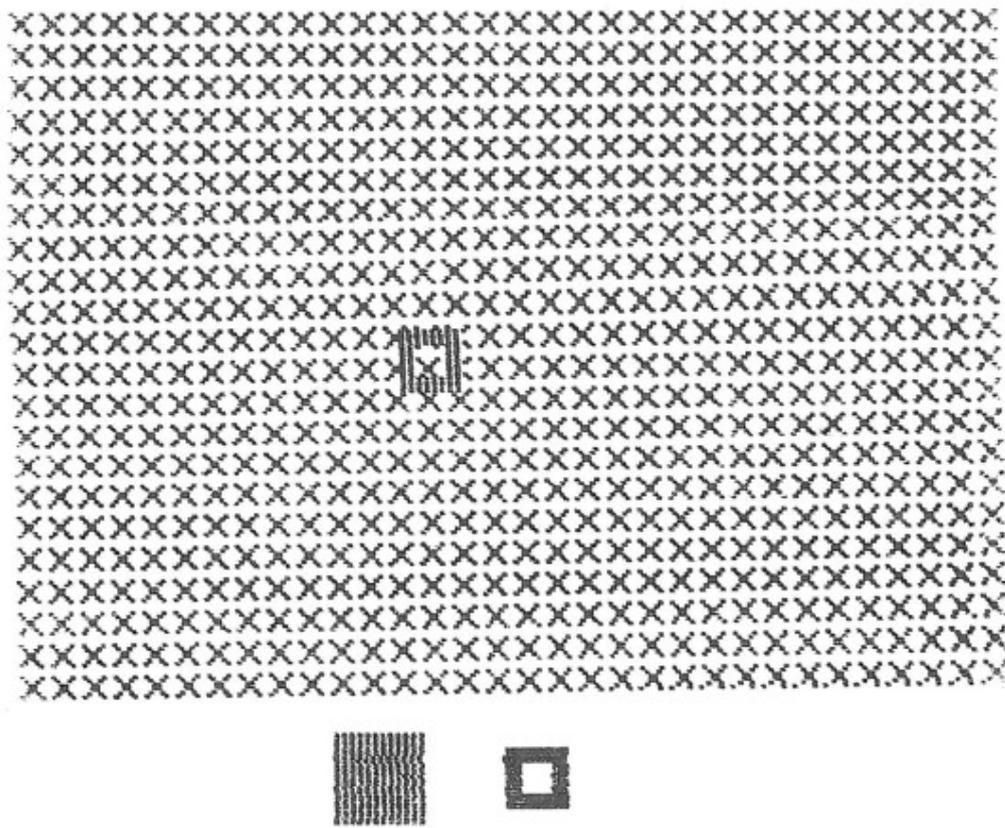
Figure 10.6

Even this by no means exhausts the possibilities of your sprite. By copying all the routines into two different locations and allocating another block of RAM as a second display file copy, you can have *two* sprites on the go.

If you use a copy of the display file with sprite 1 *already printed on it* as the 'clear' background for sprite 2, sprite 2 can be made to pass in front of sprite 1, and vice versa.

You can do a lot more with mattes. By making a matte of a section of background (perhaps using the PaintCODE routine in Appendix A) you could make a sprite pass behind a building, say. Of course, it makes fairly lavish use of memory, but the display file is easily split into thirds, and you could use and store just one particular third.

CHAPTER 11
The Attributes File

In the Spectrum, the display file and the attributes file are so cleverly linked — 'transparent' is the computerspeak word — that you are scarcely aware that they are separate and that they do quite different jobs in entirely different ways. You accept that you can print in various colours, on assorted background tints, with only the briefest directions in the PRINT instructions.

However, even in the simplest operations it does no harm to realise what the two files of data are up to. By separating their functions, you can increase the range of possible operations considerably.

The attributes file is really a display file with a much coarser 'mesh'. It only copes with 768d bytes of information, as opposed to the 6144d bytes in the display file dealing with a total of 49,152d pixels.

These relative figures make it clear why individual pixels cannot be coloured separately in the Spectrum — if each pixel had to have a byte of attributes in tandem, we should have used up 60K of RAM straight off, before typing a line of program. (On the Spectrum, you will never be able to have a red spider walking about on a black web.) Even the Sinclair QL, with its lavish memory, only manages four bits per pixel for a full display.

Nonetheless, the attributes are very cleverly arranged. Looking at the attributes as a separate file which can be manipulated independently of the main display file, you can use it as a decision maker, controlling the appearance (or *dis*appearance) of print items in the main display. It provides, for example, an excellent way of keeping track of what is happening at various places on the the screen without letting these be apparent to the eye — hiding things, in fact.

Here is a BASIC program to illustrate what I mean:

Position Hidden in Attributes

```
 10 DIM l$(32)
 20 LET x=INT (RND*32)
 30 PRINT PAPER 0; INK 7;l$
 40 PRINT AT 0,x; PAPER 0; INK 5;" "
 99 STOP
100 FOR j=0 TO 31: IF PEEK (22528+j)<>7
```

```
THEN PRINT j
110 NEXT j
199 STOP
200 PRINT AT 0,0;
210 FOR j=0 TO 31: PRINT OVER 1; INK 8;
"A"; : NEXT j
```

Lines 10 to 50 set up a black line, with 'x' being a randomly-chosen position along it. There is nothing to pick out 'x' on the visible screen — all the positions are the same. But, unseen to us, in the attributes file for that position there is a difference — 'x' has the INK bits set to 5, whereas all the rest are 7. This difference holds, even though there is nothing to print with the INK.

Entering CONTINUE sets in motion a search for the odd byte and prints out the number at which it is found. Entering CONTINUE for a second time reveals the hidden position by printing out a line of 'A's — since they are all white, except for the hidden one, this last alone shows up in cyan.

This 'hide and seek' routine can be very useful in writing all sorts of games. As an example, here is a program to generate a pack of cards and deal them out on the screen — but invisibly (face down, as it were). The section from 100 on 'turns them over'. It reveals them on a white ground with the pips in the appropriate colours. I leave it to you to devise a way to shuffle the cards before dealing, and to prepare the graphics for the pips and the court cards. You will also have to decide what to do about the two cards for which there is no room on the display.

Deal Cards and Reveal

```
5 LET b=0: LET k=1: LET s=0
10 FOR i=0 TO 4: FOR j=1 TO 30 STEP 3
20 LET x$=CHR$ (s+144): LET b=NOT b: L
ET x=1+12*(k-1)
30 IF k>9 THEN GO TO 200
40 LET q$=x$+"          "+X$+X$+"     "+
X$+"       "+X$+X$+"  "+X$+"          "+X$+"  "
+X$+X$+"  "+X$+"  "+X$+"       "+X$+"  "+X$+X$
+"  "+X$+X$+"  "+X$+"       "+X$+"  "+X$+X$+"  "
+X$+X$+X$+X$+"       "+X$+"  "+X$+X$+"  "+X$+X
$+"  "+X$+X$+"  "+X$+X$+"  "+X$+X$+"  "+X$+X
$+X$+X$+X$+"  "+X$+X$+"  "+X$+X$+"  "+X$+X$
+X$+X$+X$+X$+X$+X$+"  "+X$
50 PRINT PAPER 3; INK 3; BRIGHT b; OVE
R 1;AT i*4+1,j;q$(x TO x+2);AT i*4+2,j;q
```

```
$(x+3 TO x+5);AT i*4+3,j;q$(x+6 TO x+8);
AT i*4+4,j;q$(x+9 TO x+11)
  60 LET k=k+1:  IF k>13 THEN LET k=1:  LE
T s=s+1
  70 NEXT j
  80 LET b=NOT b
  90 NEXT i
 100 PAUSE 0
 105 LET k=1:  LET c=0
 110 FOR i=0 TO 4:  FOR j=1 TO 30 STEP 3
 120 PRINT PAPER 7;  INK (2 AND c);  BRIGH
T 8;  OVER 1;AT i*4+1,j;"    ";AT i*4+2,j;
"    ";AT i*4+3,j;"    ";AT i*4+4,j;"    "
 130 LET k=k+1:  IF k>13 THEN LET k=1:  LE
T c=NOT c
 150 NEXT j:  NEXT i
 199 STOP
 200 IF k=10 THEN LET q$=x$+"   J   J   "
+x$
 210 IF k=11 THEN LET q$=x$+"   Q   Q   "
+x$
 220 IF k=12 THEN LET q$=x$+"   K   K   "
+x$
 230 IF k=13 THEN LET q$=x$+"   A   A   "
+x$
 240 LET x=1:  GO TO 50
```

Notice how you can use BRIGHT (in the attributes) to give an outline to two cards which are side by side, even though they may be of the same colour. BRIGHT is a very useful attribute: it gives you an extra palette of six, possibly seven, new tints (black is unaffected, of course, and the difference to blue is negligible) depending on your TV.

The positions of the pips are held in q$, while x$ is set for one of the suits by the variable 's', which is added to the first UDG position.

Unfortunately, this book cannot be illustrated by a printout of the complete screen — the ZX printer simply ignores all the attributes, so all the careful work of hiding and revealing goes for nothing.

The attributes file is laid out in a perfectly straightforward way between 5800h and 5B00h. The first byte in the file stores the attributes for the top lefthand position on the screen and thereafter it runs through them in order, to the bottom right position, 300h (768d) bytes later (see p.164 of the Spectrum manual).

Because the file is laid out in this simple way (as opposed to the display

file, of which more in the next chapter) it is very easy to plot your position on the screen and POKE directly into the attributes file.

You can equally easily PEEK into it, to check whether a position is occupied or not. This is one way of getting round the fact that SCREEN$ does not give a result with user graphics. If your graphics character is given a distinctive INK or PAPER colour (or BRIGHT, or FLASH), you can check whether it is on a certain square either by PEEKing the address in the file, or by using ATTR, which calculates the address for you.

Finally, this sensible layout to the attributes file makes it very easy to draw pictures directly to it. Here is a program to do this.

Draw with ATTR

```
10 LET x=16: LET y=10
20 PAUSE 0: LET y$=INKEY$
30 IF CODE y$<48 THEN GO TO 50
40 LET c=CODE y$-48
50 IF CODE y$=8 THEN LET x=x-(x>0)
51 IF CODE y$=9 THEN LET x=x+(x<31)
52 IF CODE y$=10 THEN LET y=y+(y<23)
53 IF CODE y$=11 THEN LET y=y-(y>0)
60 IF CODE y$=13 THEN GO TO 100
70 POKE 22528+x+y*32,c*8
80 GO TO 20
```

In line 10, 'x' and 'y' set a start position in the centre of the screen and print a coloured square. The 'x > 0', 'x < 31', etc., are a way of stopping the printed square running off the screen. They are logical functions which are worth 1 if they are true and 0 if they are not. So 1 will be added to x or y, only if they fall between the set limits (as described in Chapter 8). The program will give you the full 24 screen lines to play with.

The program first sets a colour at line 30, after which you must use the cursor keys with CAPS SHIFT to move the square about. If you hold down a key, the repeat operates. I have not included coding for diagonal moves, but there is a similar program in Appendix A (PaintCODE) which illustrates how this can be done.

The result comes out like a bright and satisfying kind of finger painting and is ideal for drawing backgrounds for print displays, or whatever. Remember that, as you have been dealing only with the attributes, ordinary printed characters can be compiled and displayed over the background quite independently, although you will have to include 'PAPER 8; INK 9', to make sure that they don't import their own attributes with them.

Your impressionist masterpiece needs to be stored somehow, if it is not

to be lost as soon as you LIST your programming. You *can* do this with the Spectrum's SAVE …SCREEN$. However, it takes a terribly long time to do the SAVEing (and to LOAD it) and you will be SAVEing all the print display file as well, which you don't really want in the present case.

It is quicker and neater to copy the whole attributes file somewhere else in the RAM, and SAVE it as a block of data, with 'SAVE ATTRfile CODE xxxxx,768'. This takes a fraction of the time of SCREEN$ and puts the copy file somewhere where it can be called up at will during your program.

You can make the copy using a BASIC program like:

Store ATTR File at F000h (61440d)

```
10 FOR j=0 TO 768
20 POKE 61440+j,PEEK (22528+j)
30 NEXT j
```

A machine code program, though, is even simpler. It works in the twinkling of an eye, rather than about 10 seconds, as does the reverse program which will put your design back on the screen whenever you want it.

Store Attributes File at F000h

```
F500  21 00 58      LD    HL,5800
F503  11 00 F0      LD    DE,F000
F506  01 00 03      LD    BC,0300
F50B  C9            RET
```

Restore Attributes File from F000h

```
F50C  21 00 50      LD    HL,5000
F50F  11 00 F0      LD    DE,F000
F512  11 00 03      LD    BC,0300
F515  ED B0         LDIR
F517  C9            RET
```

The two routines (using the addresses I have given) are called by 'USR 62720' and 'USR 62732'.

The 768d bytes used to store the attributes file do not take up a large amount of extra RAM — especially when compared with the 6144d bytes for the print display file. However, they do store a good deal of redundant information for our purposes — we are only interested in the PAPER

colours. It is quite possible to extract the PAPER information and store this in 288d bytes, by itself.

Colour

The method of storing these colour bits leads us into the territory of colour reproduction in general. Photography, printing and computer displays all follow the same principles for colour. This is not altogether surprising, since our own colour vision depends on the presence or absence of three primary colours, which is the principle employed in the colour reproduction systems I've mentioned.

To deal with the general principles first. The Spectrum stores PAPER colours in bits 3, 4 and 5 of each attributes byte (see p.116 of the Spectrum manual). The first of these bits codes for blue, the second for green, and the third for red. Ignoring the fact that the bits have been shifted three positions to the left, the three primary colours are coded by setting bit 1 for blue, bit 2 for red and bit 3 for green, giving them positional values of 1, 2 and 4. Now look at the colours over the numeral keys on the Spectrum. Neat, isn't it?

The other three colours are combinations of two primary colours: they are called 'complementary colours' and are known as cyan (from the name of the dye originally used to produce it in photography), magenta and yellow. White is a combination of all three. These check out on the keyboard, too.

The complementary colours are also known as 'minus' colours, because magenta is '(white) minus green', yellow is 'minus blue' and cyan is 'minus red'.

You may be wondering what all this has to do with the Spectrum but all will shortly be revealed in a blinding flash, because if we peel off all the ATTR bits at position 3 and store them, we have, in fact, stored the blue image. The bits at position 4 will store the green image; and the bits at position 5 will store the red image. The complementary colours will be stored in the appropriate pairs of blocks and white in all three.

To restore the bits to the screen in the correct colours, we can either *add* the primaries on a black screen (in computer terms, set the primary bits in a blank byte), or we can *subtract* the complementary colours from a white screen, which means we reset the complementary pairs of bits in a byte where the bits are all initially set.

All these exercises can be done using the logical functions of the Z80 chip. Some programs to carry them out follow. Most of the hard work lies in getting the bits and bytes into the right order and the right position.

The first routine is a little more complicated than it need be, because I have arranged for the 'PAPER colour' bits, which we collect and store,

to be arranged as 'characters' so that they can be printed out as UDGs. This shows the complete screen compressed into a block of 4 × 3 characters.

The actual testing is done by checking whether bit 3 (the blue bit) of each ATTR byte is set. If it is, we set a bit in the A register and eventually transfer the completed byte in A to a storage position. Meanwhile, we have shifted the whole ATTR file one bit to the right, so that bit 4 becomes bit 3. When we perform the operation again, the red bits will have been stored. At the end of the second round, ATTR will have been shifted again, so that the green bit becomes bit 3.

We restore the whole attributes file at the end of the routine.

3-colour Separations of ATTR

```
F000  11 00 F1       LD    DE,F100      START OF STORAGE ADDRESS
F003  06 03          LD    B,03         COUNT FOR 3 PASSES = BLUE,
                                         RED, GREEN
F005  C5             PUSH  BC
F006  21 00 58       LD    HL,5800      START OF ATTR
F009  0E 03          LD    C,03         VERTICAL — 3 PRINT POS.
F00B  06 08          LD    B,08         VERTICAL — 8 LINES PER PRINT
                                         POS.
F00D  C5             PUSH  BC
F00E  D5             PUSH  DE
F00F  0E 04          LD    C,04         HORIZONTAL — 4 PRINT POS.
F011  06 08          LD    B,08         HORIZONTAL — 8 BITS PER
                                         PRINT POS.
F013  AF             XOR   A
F014  CB 5E          BIT   3,(HL)       TEST BIT 3 ATTR
F016  28 01          JR    Z,F019       JUMP IF NOT SET
F018  37             SCF                SET CARRY IF BIT 3 SET
F019  17             RLA                GET CARRY INTO A
F01A  CB 0E          RRC   (HL)         SHIFT ATTR TO GET BIT 4 INTO
                                         BIT 3 POS.
F01C  23             INC   HL
F01D  10 F5          DJNZ  F014
F01F  12             LD    (DE),A       TRANSFER A TO STORAGE BYTE
F020  C5             PUSH  BC
F021  EB             EX    DE,HL
F022  0E 08          LD    C,08
F024  09             ADD   HL,BC        MOVE STORAGE ADDRESS
F025  EB             EX    DE,HL        8 BYTES = NEXT UDG
F026  C1             POP   BC
F027  0D             DEC   C
F028  20 E7          JR    NZ,F011      REPEAT OPERATION FOR
                                         4 UDG
F02A  D1             POP   DE
F02B  13             INC   DE           ADDRESS NEXT BYTE OF
                                         UDG CHRS
```

```
F02C  C1               POP   BC
F02D  10 DE            DJNZ  F00D          REPEAT ALL OPS FOR 8
                                           BYTES PER CHR
F02F  C5               PUSH  BC        ⎫
F030  EB               EX    DE,HL     ⎪
F031  0E 18            LD    C,18      ⎬  MOVE STORAGE ADDRESS
F033  09               ADD   HL,BC     ⎪  TO NEXT GROUP OF 4 CHRS
F034  C1               POP   BC        ⎭

F035  EB               EX    DE,HL
F036  0D               DEC   C
F037  20 D2            JR    NZ,F00B       REPEAT ALL OPS 3 TIMES
                                           FOR 3 GROUPS OF 4
F039  C1               POP   BC
F03A  10 C9            DJNZ  F005          REPEAT WHOLE 3 TIMES
                                           FOR BLUE, RED, GREEN
F03C  01 00 03         LD    BC,0300   ⎫
F03F  CB 06            RLC   (HL)      ⎪
F041  CB 06            RLC   (HL)      ⎪
F043  CB 06            RLC   (HL)      ⎪
F045  2B               DEC   HL        ⎬  RESTORE SHIFTED ATTR
F046  0B               DEC   BC        ⎪  FILE TO ORIGINAL STATE
F047  78               LD    A,B       ⎪
F048  B1               OR    C         ⎪
F049  20 F4            JR    NZ,F03F   ⎭
F04B  C9               RET
```

The three blocks of characters should now be stored in the RAM, starting at F100h (61696d) for the blue, F160h (61792d) for the red and F1C0h (61888d) for the green. The routine is placed at F000h, which is 61440d.

Here is a BASIC program to execute the machine code routine and then display the three miniature 'colour separation' images. To print out these images, the program POKEs the requisite addresses into the system variable UDG, at 5C7Bh and 5C7Ch (23675d and 23676d). Tack it on to the BASIC 'Draw with ATTR' program.

Generate and Print out Colour Separation Images

```
100 RANDOMIZE USR 61440
110 POKE 23675,0: POKE 23676,241
120 PRINT INK 8; INVERSE 1;"ABCD"'"EFGH
'"'"IJKL"
130 POKE 23675,96
140 PRINT INK 8; INVERSE 1;"ABCD"'"EFGH
'"'"IJKL"
150 POKE 23675,192
```

```
 160 PRINT INK 8; INVERSE 1;"ABCD"''"EFGH
"'"IJKL"
```

The printout in **Figure 11.1**, with letters for the colours, shows how the separations take shape. Each miniature pattern depicts a minus colour. The border, being black, shows up in each of them. If you could make transparencies of these miniatures and view them, one on top of the other, you would get full colour reproduction of the original pattern.
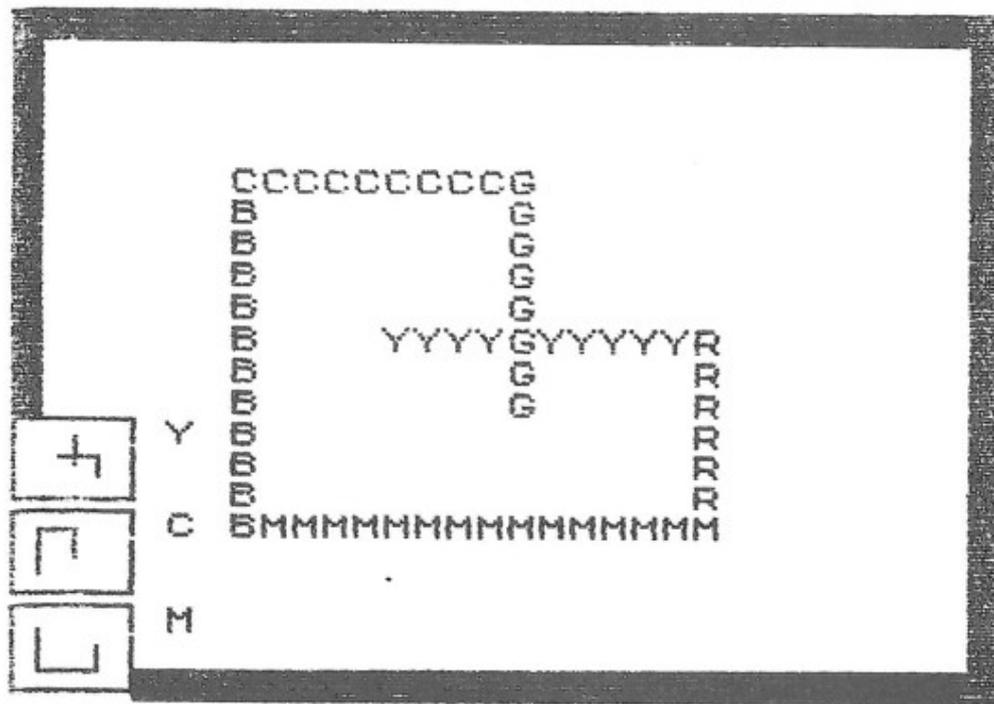

Figure 11.1

In the meantime, let's write a routine to put the attributes together again. Here it is.

Reconstitute in ATTR

F050	DD 21 60 F1	LD	IX,F160	ADDRESS FOR 'RED' IMAGE
F054	21 00 58	LD	HL,5800	ATTR FILE
F057	0E 03	LD	C,03	1
F059	06 08	LD	B,08	2
F05B	C5	PUSH	BC	
F05C	DD E5	PUSH	IX	CONTROL LOOPS
F05E	0E 04	LD	C,04	3
F060	06 08	LD	B,08	4
F062	DD CB A0 06	RLC	(IX-60)	'BLUE' STORAGE ADDRESS
F066	38 02	JR	C,F06A	
F068	CB 9E	RES	3,(HL)	CANCEL 'BLUE' BIT = YELLOW IMAGE

```
F06A  DD CB 00 06   RLC         (IX)        'RED' STORAGE ADDRESS
F06E  38 02         JR          C,F072
F070  CB A6         RES         4,(HL)      CANCEL 'RED' BIT = CYAN
                                            IMAGE
F072  DD CB 60 06   RLC         (IX+60)     'GREEN' STORAGE ADDRESS
F076  38 02         JR          C,F07A
F078  CB AE         RES         5,(HL)      CANCEL 'GREEN' BIT =
                                            MAGENTA IMAGE
F07A  23            INC         HL
F07B  10 E5         DJNZ        F062        LOOP 4
F07D  11 08 00      LD          DE,0008
F080  DD 19         ADD         IX,DE
F082  0D            DEC         C
F083  20 DB         JR          NZ,F060     LOOP 3
F085  DD E1         POP         IX
F087  DD 23         INC         IX
F089  C1            POP         BC
F08A  10 CF         DJNZ        F05B        LOOP 2
F08C  1E 18         LD          E,18
F08E  DD 19         ADD         IX,DE
F090  0D            DEC         C
F091  20 C6         JR          NZ,F059     LOOP 1
F093  C9            RET
```

As you can see, this routine relies heavily on the indexed register IX. This allows us to access the stored colour information in the same way right through. The offsets between equivalent positions in the three colour images always remain the same.

The base address for IX points to the 'green' store. This is the middle one of the three. I have done this because the register can only index forward 128 bytes. Since each colour store contains 96 bytes, it would not be possible to index forward to the third colour from the first. However, you can index *backwards* by 127 bytes as well, so by choosing the middle position you can reach all three store addresses.

Before we finally leave the attributes, I'd like to point out that the techniques we have been talking about could be a very handy method of producing miniature drawings for sprites. Many people find it easier to do the drawings on a full-sized TV screen and then scale them down to sprite size. You could even try and use the colour separations to produce a cycle of three drawings for animation. But I suspect that, by the time you had worked out the colour relationships, you might just as well have designed each one from scratch.

However, if you wanted to try, here is a BASIC program to change the colours of an attributes painting on the screen:

Turn Designs Red or Green

```
  99 REM Turn image red
 100 FOR j=0 TO 768: IF PEEK (22528+j)<>
7*8 THEN POKE 22528+j,16
 110 NEXT j
 190 PAUSE 0
 199 REM Turn image green
 200 FOR j=0 TO 768: IF PEEK (22528+j)<>
7*8 THEN POKE 22528+j,32
 210 NEXT j
```
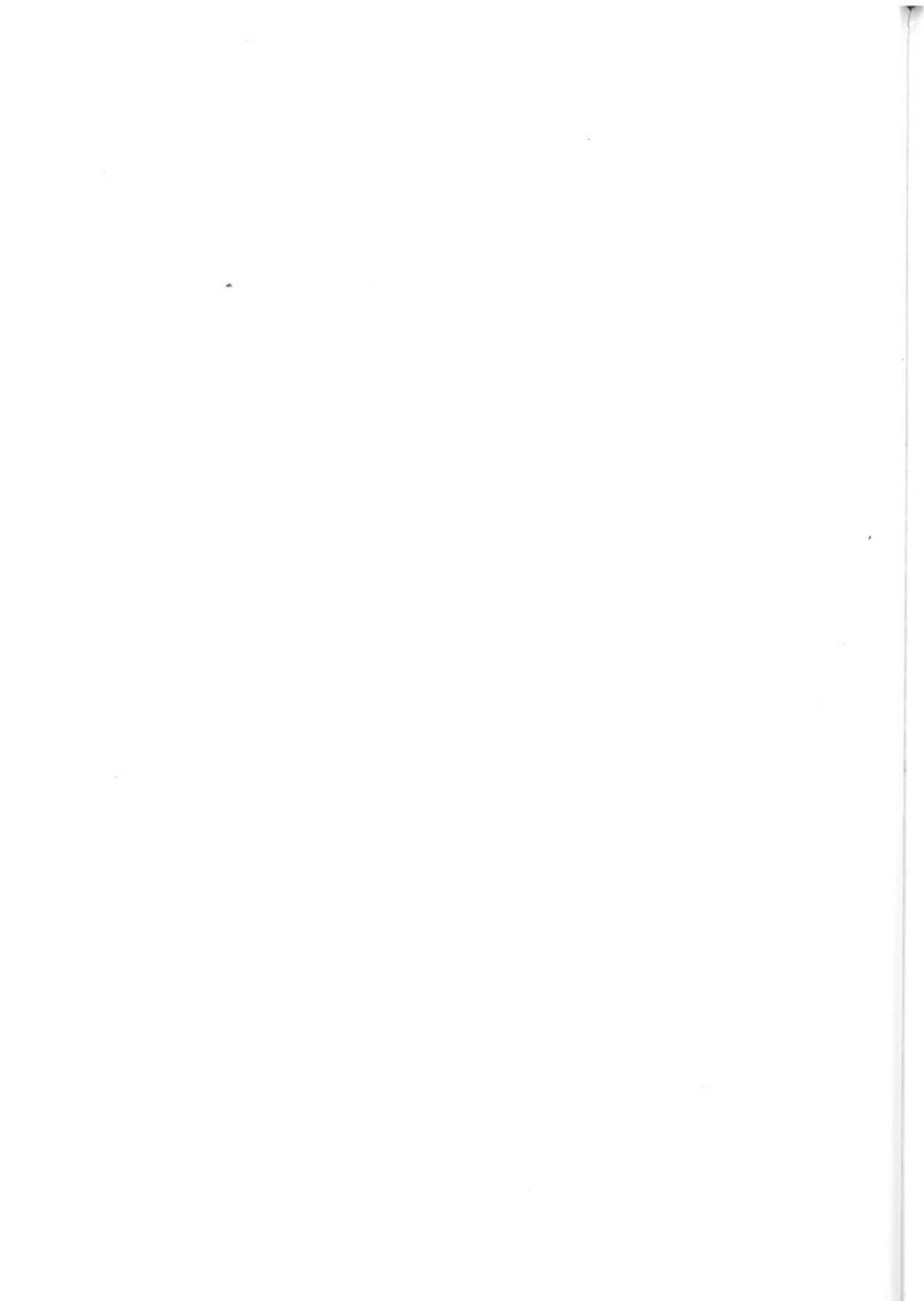
CHAPTER 12
The Display File

While the attributes file is of interest, the main print display file is far more important in the normal use of the Spectrum. At a pinch, you could get by without the attributes at all, but without the main display file the computer would be virtually blind and dumb.

Before you can start manipulating the main Sinclair display file, you have to understand how it works. Alas, as has already become clear, it is nothing like as straightforward as the attributes file. Apart from being very much longer, it is laid out in a notoriously unconventional way (see Chapter 9). You need a clear head to map your way about the screen. No wonder the Spectrum manual remarks primly, 'It is rather curiously laid out, so you probably won't want to PEEK or POKE in it.' Unfortunately, you probably will....

In dealing with sprites, we made a lot of use of the ROM routine PIXEL__AD, in order to pinpoint the screen address once we had the x and y PLOT coordinates. But if you want, for example, to scroll the whole screen, pixel by pixel, this routine is not really the answer.

Bearing in mind the mental model I described in Chapter 9 (with the 'long character lines') let's look again at the make-up of a column of print positions.

In printing a black column or bar down the screen, you are POKEing a series of addresses in the display file with the value FFh (255d). If we start at the top lefthand corner of the screen, the first address is 4000h (16384d). Then we have to add 100h (256d) for the next seven positions, which are the beginning of a line, in a group of long lines. After that, we have to backtrack to position 32 on the first long line to get the next vertical position below the last, on the screen. (Look again at **Figure 9.1** in Chapter 9.)

We then go through the same operation of adding 100h eight times, as we did before, and again find a position 20h along for the next group of lines... up to eight times in all. Then we have to tackle the next group of long lines, starting at the base address plus 800h (2048d)... And then the third.

Actually, in a BASIC program, it's not too bad:

POKE Vertical Bar to Screen

```
10 LET x=16384
20 FOR z=0 TO 2
30 FOR i=0 TO 7
40 FOR j=0 TO 7
50 POKE x+256*j+32*i,255
60 NEXT j: NEXT i
70 LET x=x+2048: NEXT z
```

You can see rather clearly how the loops of 8, 8 and 3 are nested together. The same holds true for the machine code version, although the different sections of the counts are tested by logical ADDing, so that it is not so easy to follow.

Print Vertical Bar

```
E000  21 10 40    LD    HL,4010
E003  06 C0       LD    B,C0
E005  3E FF       LD    A,FF
E007  77          LD    (HL),A
E008  24          INC   H
E009  7C          LD    A,H
E00A  E6 07       AND   07          TEST FOR BOTTOM
                                    OF LONG LINE
E00C  20 0A       JR    NZ,E018
E00E  7D          LD    A,L
E00F  C6 20       ADD   A,20
E011  6F          LD    L,A         GET NEXT POS. ON
                                    LONG LINE
E012  3F          CCF
E013  9F          SBC   A,A
E014  E6 F8       AND   F8          CORRECT FOR EACH
                                    THIRD OF SCREEN
E016  84          ADD   A,H
E017  67          LD    H,A
E018  10 EB       DJNZ  E005
E01A  C9          RET
```

Finding UDGs

Before we go on to what bureaucrats would probably like to call an 'in-depth analysis of the screen situation', the addressing method I have just outlined does offer help in a particular area where the Spectrum fails.

The manual tells us, on p. 101, that SCREEN$ does not recognise UDGs. As bad luck would have it, these are exactly the characters we are most likely to want to look for. However, we now know how to

follow, say, the top line of character bytes through the display file, and we can use this to spot a UDG, if we prepare it properly.

Cast your mind back to the pack of cards we were discussing a few pages ago. I have made some designs for the four suits, which can be entered into the UDGs. They look quite convincing, as you can see from **Figure 12.1**.
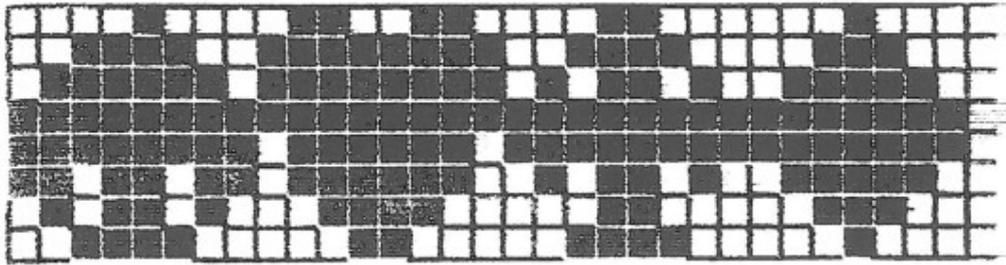

Figure 12.1:   Designs for Card Suits

If you look at the big characters in the grid, you will see that the top line of each suit is different from each of the others, which means that when they are in the display file the top byte at their print position will be identifiable. Here are the values of these bytes:

Spade	08h	8d
Heart	66h	102d
Club	18h	24d
Diamond	10h	16d

You can look for them with the program below, which will put a coloured square on the first occurrence. Incidentally, none of the Sinclair characters use any of these combinations for their first lines — most of them are blank, but © is 3Ch (60d).

Locate Suit

```
   5 PRINT AT 8,5;"♠ ♥ ♣ ♦"
  10 FOR i=0 TO 2: FOR j=0 TO 25
  20 IF PEEK (16384+j+i*2048)=16 THEN GO
TO 40
  30 NEXT j: NEXT i
  40 PRINT AT i*8+INT (j/32),j-32*INT (j
/32); OVER 1; PAPER 4;" "
```

You are more likely, however, to want to identify a suit at a position you already know. In that case, you would use a routine like the following

one, which will add the correct colour to the graphic character at the chosen position.

Check Top Line

```
   1 REM x=COLUMN   y=LINE
1010 LET zone=INT (y/8)
1020 LET y1=y-8*zone
1030 LET p=16384+zone*2048+y1*32+x
1040 PRINT INK 2 AND (PEEK p=16 OR PEEK
p=102);
```

The variable 'zone' identifies which third of the screen we are looking at.

Scrolling

There is one other operation to do with the display file which is not difficult, and that is to scroll the file to the left or to the right. (I'm talking of a pixel scroll rather than a full print position, which is not really a problem, anyway.)

The reason why the pixel scroll is easy is that, as you are not changing the line positions, it does not matter in what order you take them — you can simply start at the top and work down to the bottom.

To do the scroll, you take each byte of the display file in turn (addressed by HL), do a 'rotate (HL)' — either left or right — and move on to the next. The only thing you have to remember is that you must start at the *end*, if you are scrolling to the left and at the *beginning*, if you are scrolling to the right. This is to allow the bits that drop off the end of each character to be picked up and put in the correct position in the next character.

Here is the simplest sort of scroll. You have to provide two loops — ie do the scroll a line at a time — so that you have a chance to wipe out the last carry of the line. If you don't do this, the print will scroll back on to the screen, turning up at some very peculiar places. Try it for yourself, by substituting '00' for 'AF', at F007h.

Scroll Screen to Left

```
F000 21 FF 57    LD    HL,57FF
F003 0E C0       LD    C,C0
F005 06 20       LD    B,20
F007 AF          XOR   A
F008 CB 16       RL    (HL)
F00A 2B          DEC   HL
```

```
F00B  10 FB          DJNZ  F008
F00D  0D             DEC   C
F00E  20 F5          JR    NZ,F005
F010  C9             RET
```

To get it to scroll the other way, change the first line to '21 00 40', change line F008h to 'CB 1E' and the next instruction from '2B' to '23'.

Scrolling is not much fun unless you have something to *scroll on*, as well as scroll off. This means having another display file prepared at another address.

As an example, here's a routine to scroll two screens round and round, like a revolving drum. It could be useful for providing a moving background to something.

Drum Scroll — Right to Left ✔

```
F000  11 FF E7       LD    DE,E7FF     ←
F003  21 FF 57       LD    HL,57FF     ←
F006  F5             PUSH  AF
F007  0E C0          LD    C,C0
F009  F1             POP   AF
F00A  E5             PUSH  HL
F00B  06 20          LD    B,20
F00D  AF             XOR   A
F00E  7E             LD    A,(HL)
F00F  17             RLA               ←
F010  77             LD    (HL),A
F011  2B             DEC   HL          ←
F012  10 FA          DJNZ  F00E
F014  06 20          LD    B,20
F016  1A             LD    A,(DE)
F017  17             RLA               ←
F018  12             LD    (DE),A
F019  1B             DEC   DE          ←
F01A  10 FA          DJNZ  F016
F01C  E3             EX    (SP),HL
F01D  3E 00          LD    A,00
F01F  17             RLA               ←
F020  B6             OR    (HL)
F021  77             LD    (HL),A
F022  E3             EX    (SP),HL
F023  F1             POP   AF
F024  F5             PUSH  AF
F025  0D             DEC   C
```

```
F026  20 E1              JR    NZ,F009
F028  F1                 POP   AF
F029  C9                 RET
```

61488

Transfer Second Display to D000h

```
F030  21 00 40           LD    HL,4000
F033  11 00 D0           LD    DE,D000
F036  01 00 18           LD    BC,1800
F039  ED B0              LDIR
F03B  C9                 RET
```

61504

Scroll One Complete Screen

```
F040  01 00 00           LD    BC,0000
F043  C5                 PUSH  BC
F044  CD 00 F0           CALL  F000
F047  C1                 POP   BC
F048  10 F9              DJNZ  F043
F04A  C9                 RET
```

The main program is really just two versions of the earlier program spliced together. The visible display file is at 4000h and the second one is at D000h. HL holds the address of the first, and DE holds the second. (Remember, we are starting at the end, so the initial addresses are 57FFh and E7FFh.)

There's an interesting little operation at F01C–F024h. At the end of the line, we are left with a dangling carry bit. It has dropped off the last byte and has to be tacked on to the first byte of the line, if the continuous movement is to be preserved. To do this, we PUSH the address of the first byte, early on (at F00Ah), because by the time we get to F01Ch we have a new address in HL, which we want to keep. There's nowhere to PUSH the new address, so we swap it with the first address, still on the top of the stack and get whatever was in the carry into the byte at this address. Then we swap the HL addresses once again and throw away the first one (the address of the first byte) by POPping it into AF and get on with the job.

The arrows in the routine point to the bytes which have to be changed, if you want to change the direction of the scroll. Here they are, tabulated:

	R to L		*L to R*	
F000–02	11 FF E7		11 00 D0	
F003–05	21 FF 57		21 00 40	
F00F	17	RLA	1f	RRA
F011	2B	DEC HL	23	INC HL

F017	17	RLA	1F	RRA
F019	1B	DEC DE	13	INC DE
F01F	17	RLA	1F	RRA

The other two short routines are for use in connection with the main one. The first transfers whatever is on the screen to the second display file, which we create at D000h. The other is a demonstration program which will completely scroll from one screen to the other: the operation takes exactly 256 cycles, from B = 0 to B = 0.

Rearranging the file

Scrolling the screen sideways, as you can see, is a fairly painless operation. However, if we want to scroll up and down, we are tangled again with that nightmare of jumping from line to line. After struggling with it for some time, I have come to the conclusion that the only sensible course is to rewrite the entire display file at another address and to rearrange it in a way which makes it possible to deal with. That is with line 2 coming after line 1, line 3 after line 2, and so on.

In this way, we only have two short routines to deal with all situations — one to rearrange the data in the shape we want it and the other to reconstitute the display file and get it back on the screen.

The routines are very much like the Print Vertical Bar routine at the beginning of this chapter, with the added factor of dealing with a complete line of bytes, rather than a single one. Here is the first:

Rearrange D/File at D000–D800h

```
F000  21 00 40    LD    HL,4000
F003  11 00 D0    LD    DE,D000
F006  0E C0       LD    C,C0

F008  E5          PUSH  HL
F009  06 20       LD    B,20
F00B  7E          LD    A,(HL)
F00C  12          LD    (DE),A
F00D  23          INC   HL
F00E  13          INC   DE
F00F  10 FA       DJNZ  F00B

F011  E1          POP   HL
F012  24          INC   H
F013  7C          LD    A,H
F014  E6 07       AND   07
F016  20 0A       JR    NZ,F022
```

```
F018  7D           LD    A,L
F019  C6 20        ADD   A,20
F01B  6F           LD    L,A
F01C  3F           CCF
F01D  9F           SBC   A,A
F01E  E6 F8        AND   F8
F020  84           ADD   A,H

F021  67           LD    H,A

F022  0D           DEC   C
F023  20 E3        JR    NZ,F008
F025  C9           RET
```

Writing the program to restore the display file is even less arduous: you make a copy of the first program at a new address (I have chosen F030h). If you have a good assembler or editing program, you can probably do this automatically. Then you alter just two instructions: you change 'LD A (HL)', at F03Bh, into 'LD A (DE)' and you change 'LD (DE)A', at F03Ch, into 'LD (HL)A'.

Restore D/File from D000–D800h

```
F030  21 00 40     LD    HL,4000
F033  11 00 D0     LD    DE,D000
F036  0E C0        LD    C,C0

F038  E5           PUSH  HL
F039  06 20        LD    B,20
F03B  1A           LD    A,(DE)
F03C  77           LD    (HL),A
F03D  23           INC   HL
F03E  13           INC   DE
F03F  10 FA        DJNZ  F03B

F041  E1           POP   HL
F042  24           INC   H
F043  7C           LD    A,H
F044  E6 07        AND   07
F046  20 0A        JR    NZ,F052

F048  7D           LD    A,L
F049  C6 20        ADD   A,20
F04B  6F           LD    L,A
F04C  3F           CCF
F04D  9F           SBC   A,A
F04E  E6 F8        AND   F8
```

```
F050  84          ADD   A,H
F051  67          LD    H,A

F052  0D          DEC   C
F053  20  E3      JR    NZ,F038
F055  C9          RET
```

Armed with these two routines, a whole range of screen manipulations becomes possible. Up and down scrolling, pixel by pixel, becomes a simple LDIR instruction, and you can easily scroll sections of the screen (any section, not just thirds), or even windows.

It is equally easy, of course, to scroll from side to side — you could quite well replace the Drum Scroll routine we discussed earlier, but then you would have to have *two* spare display files on the go at the same time, which seems rather lavish for what you are trying to do.

Here is a more exotic diagonal scroll, with a little routine to run it continuously.

Diagonal Screen Scroll

```
F100  21  20  D0    LD    HL,D020
F103  11  00  D0    LD    DE,D000
F106  01  E0  17    LD    BC,17E0
F109  CB  1E        RR    (HL)
F10B  ED  A0        LDI
F10D  EA  09  F1    JP    PE,F109
F110  C3  30  F0    JP    F030
```

Repeat Routine

```
F200  06  A0        LD    B,A0
F202  C5            PUSH  BC
F203  CD  00  F1    CALL  F100
F206  C1            POP   BC
F207  10  F9        DJNZ  F202
F209  C9            RET
```

Do not, as I did, forget to include the 'PUSH BC' and 'POP BC' to preserve the count in the second program — to see the entire listing disappear majestically into the righthand corner, never to be seen again!

Shrinking the screen

These are only small samples of the freedom given by our rearranged display file. There is no problem in shrinking a screen down to quarter size, as shown in **Figure 12.2**.
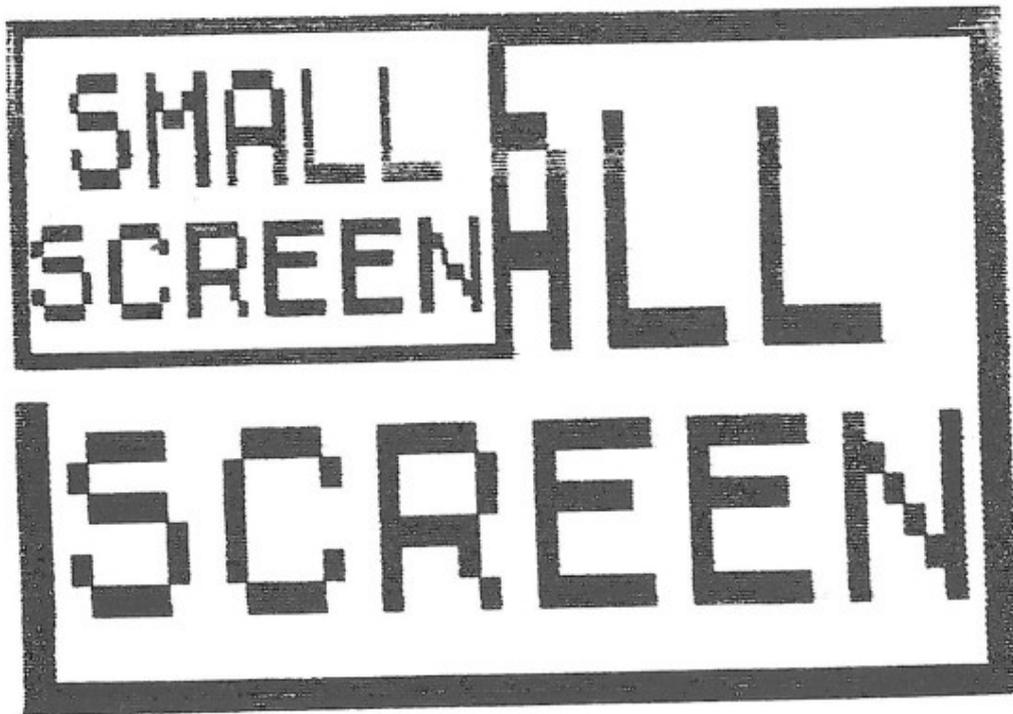
Figure 12.2

The routine reproduced here is a fairly 'coarse mesh' one — it just squeezes together all the righthand nibbles, using the 'rotate left decimal' instruction and skips every alternate line. It will only work with a fairly bold design, such as the one I have used. But it would not be at all difficult to pick up alternate bits from the original file, to give a better resolution, rather as we did to generate the 'four-bit character' font. However, I don't think print would ever be legible with the letters compressed into a 4 × 4 matrix.

Make Quarter-sized Screen

```
F060  11 00 D0    LD    DE,D000
F063  21 00 D0    LD    HL,D000
F066  0E 60       LD    C,60
F068  06 10       LD    B,10

F06A  1A          LD    A,(DE)
F06B  ED 6F       RLD
F06D  13          INC   DE
F06E  1A          LD    A,(DE)
F06F  ED 6F       RLD
F071  13          INC   DE
F072  23          INC   HL
F073  10 F5       DJNZ  F06A

F075  C5          PUSH  BC
F076  01 20 00    LD    BC,0020
```

```
F079  EB            EX      DE , HL
F07A  09            ADD     HL , BC
F07B  EB            EX      DE , HL
F07C  0E  10        LD      C , 10
F07E  09            ADD     HL , BC
F07F  C1            POP     BC
F080  0D            DEC     C
F081  20  E5        JR      NZ , F068

F083  C3  30  F0    JP      F030
```

CHAPTER 13
Inputs and Outputs

I sometimes forget that the Spectrum is not self-contained. But if it were not for a number of input and output devices by which it is served, the computer would be of no practical use.

The standard input devices are the keyboard and the cassette recorder, or Microdrive. Output devices are the television, the recorder (or Microdrive, once again) with the beeper and the printer. Many other I/O devices can be added, of course — joysticks, paper printers and whatnot. But they all require special interfaces, and, in this book, I want to look at methods of using the different devices which make up the *standard* Spectrum, in ways that Sinclair Research did not intend.

I can say straight away that there is nothing much to be done with the recorder and the television, as such (as opposed to the information we send them). Nor is there much to be gained by tinkering with the output to the printer, to say nothing of the danger of tying the computer in a knot, and so I will not be dealing with these devices in this chapter.

The areas that are worth exploring are the output to the border of the television screen, which is normally controlled by the 'BORDER x' command, and to the beeper.

Even in these areas, the scope is limited. You can turn the border into an imitation of a Neapolitan icecream, if that appeals, or make the screen 'explode'. Or you can modify the single note BEEP, to give a range of catcalls and whistles. To make some of these tricks more effective, we can also change the interrupt mode.

Perhaps this should be explained first.

The interrupt mode
In normal operation, the Spectrum hardware 'interrupts' the work of the Z80 microprocessor every 20 microseconds (50 times a second). This means that the Z80 stops whatever it is doing and goes into an interrupt routine, which has to be completed before it can get on with its main task.

This interrupt routine is fixed in the ROM at 0038h, and consists of a quick update on the Spectrum timer, at 'FRAMES', followed by a keyboard scan, to see what keys are depressed.

The Z80 microprocessor is designed to allow these interrupts, to carry out precisely this type of regular operation. It has, in fact, three kinds (or modes) of interrupt, known as IM0, IM1 and IM2. There is also a 'non-maskable', ie unstoppable, interrupt, NMI.

IM0 can't be used by the Spectrum. IM1 is the mode normally used to do the keyboard scan, etc.: it causes a direct jump to the start of the routine at 0038h, the 'RST 38' instruction. The all-powerful NMI has been blocked — apparently deliberately — in the Sinclair ROM. Which leaves only IM2 to consider.

Let's see what IM2 does. When the Z80 has been programmed for the IM2 mode and an interrupt occurs, the Z80 immediately concocts an address. This address is made from a low order byte, taken from the gadget which did the interrupting — the Sinclair ULA — and a high order byte, taken from the contents of the I register, an esoteric register provided by the Z80 for this purpose.

Having put together this address, the Z80 then jumps to it, hoping to find *another* address put there by the programmer. It then proceeds to execute whatever routine it finds at this second address.

This sounds very complicated, but the purpose is to allow the microprocessor to tackle a particular routine dictated by a particular peripheral.

Now, what can we do with that? When an interrupt occurs, the ULA is not programmed to supply any particular byte, so the default byte on the bus should be 'FF'.

This means that the address which IM2 will concoct will be 'xxFF', where 'xx' has to be what IM2 finds in the I register. And this is something *we* can put there.

For complicated hardware reasons, the complete address should not be placed between 4000h and 7FFFh, which — as bad luck would have it — is the entire RAM area for the 16K Spectrum! However, 16K owners should not despair yet.

48K owners are free to choose any xxFF address they like above 8000h and put into it the address of the interrupt routine they have written: ie if your interrupt routine is at F001h, then you could place this address at EFFFh, in the form:

```
EFFF    01
F000    F0
```

So what you have to do is to set the interrupt mode as IM2, load I with (in the present case) 'EF' and provide a suitable routine at F001h. The Spectrum will then carry out this routine every time an interrupt occurs, in the middle of whatever else you have programmed it to do.

If you want the program to continue updating the clock and scanning the keyboard, then you should make it jump to 0038h when your own routine is

finished. Otherwise, you do an 'enable interrupts' ('FB'), followed by RET, or RETI.

An important point to note is that *you must save all the registers* to be used during the interrupt routine, including AF and the alternate registers if used. They must all be PUSHed at the start and then POPped again before the finish.

16K owners may be feeling a bit disheartened at the moment, but there is a dodge which allows them to join in. It is possible to use an address in the ROM. Of course, the ROM can't be modified, but you can look for two adjacent bytes which are at 'xxFF' and 'xxFF+1', and which together hold a viable address in the spare RAM.

This search yields the following results, when looking for addresses in the ROM greater than A000h, at xxFFh locations:

'xxFF' Addresses in ROM (48K)

511	01FF	CE52
2559	09FF	FE69
3071	0BFF	E608
3327	0CFF	CFBF
3583	0DFF	CD17
4351	10FF	CB10
4863	12FF	CD01
5119	13FF	C255
5631	15FF	C9D9
5887	16FF	C970
7423	1CFF	C31B
8447	20FF	CD21
9215	23FF	C181
10751	29FF	E32A
11775	2DFF	D9E5
12543	30FF	EB30
12799	31FF	E128
13823	35FF	DF24
14335	37FF	A10F
14591	38FF	FFFF
14847	39FF	FFFF
15103	3AFF	FFFF
15359	3BFF	FFFF
23551	5BFF	FF00

Another list more suitable for the 16K Spectrum, goes as follows:

'xxFF' Addresses in ROM (16K)

1791	71DD	06FF
4095	0FFF	6D18

5375	14FF	6469
7935	1EFF	67CD
10495	28FF	7E5C
24063	5DFF	6964
24831	60FF	79A2

There is quite a wide choice. Almost the highest practical address for the 48K Spectrum comes at 09FFh, where there is 'FE69h'. 16K owners could try 28FFh, which yields '7E5Ch'.

I say 'almost the highest practical address', because 48K owners have a final, rather jokey choice; they can use FFFFh! Oddly, although FFFFh is the absolute end of the RAM, you don't just fall off into space after that: you simply go round again. So that the address after FFFFh, for practical purposes, is 0000h. And this pair yield quite a useful address for an interrupt routine, 'F300h'.

FFFF	00	(you can change this)
0000	F3	

There are two further notes of caution to be sounded. (After all, we are *not* doing what Sir Clive intended!) First, there is nothing that says that the byte on the input buffer has to be 'FF'. It usually is, but if you have some other gadgets hooked up, it may be different. It could be anything at all. (This snag, luckily, does not arise with the Microdrive.)

Faced with this problem, you would have to provide a complete range of bytes, 257 in all, any pair of which will give the same address. For example, you might choose the byte 'FE'. Paired together, this would provide an address 'FEFE' at which you could put your interrupt routine.

You would have to load all the bytes from, say, FD00h to FE00h with FE. Then you would have to get the byte FD into the I register, after which it wouldn't matter *what* other byte got provided for the vector address, this would always produce FEFE.

Secondly, proceed with extreme caution when using ROM addresses when the Interface 1 is connected. The Spectrum may refer the vector address to the Interface 1 ROM, with hopeless results. Alas, I know of no way round this problem except to disconnect the Interface.

We have now reached the stage when we want to be able to set the interrupt mode and supply the byte in I. The following would be the sort of routine, using a ROM address:

Set Interrupt Mode 2

not work on 128K +2a

```
F000  3E  09        LD    A,09
F002  ED  47        LD    I,A
```

```
F004 ED 5E        IM2
F006 C9           RET
```

We also need a similar piece of machine code, to restore the status quo, once we have finished using our interrupt program.

Restore Interrupt Mode 1

```
F010 3E 3F        LD    A,3F
F012 ED 47        LD    I,A
F014 ED 56        IM1
F016 C9           RET
```

The programs we set up at the interrupt address FE69h can be anything within reason. However, it has to be remembered that the Z80 cannot get on with the main routine while it is attending to the interrupt routine, so the main routine will be slowed down.

The main problem with the interrupt switch, which really limits its use, is the fact that we are tinkering with the wrong thing. You really want to be able to control the *timing* of the interrupt, as much as the interrupt routine: there is a limit to the number of things you want to check regularly 50 times a second. It would be much more useful to have the interrupt under our own control, in order to exploit it fully. But this is not possible, so we should be thankful for what we have.

The television screen

Here is a program to turn the television border into a Neapolitan icecream.

Neapolitan Ice Border

```
FE69 F5           PUSH  AF
FE6A C5           PUSH  BC
FE6B 06 08        LD    B,08
FE6D 0E 00        LD    C,00
FE6F 78           LD    A,B
FE70 3D           DEC   A
FE71 D3 FE        OUT   (FE),A
FE73 00           NOP
FE74 00           NOP
FE75 00           NOP
FE76 0D           DEC   C
FE77 20 F9        JR    NZ,FE72
```

```
FE79  10  F2              DJNZ  FE6D
FE7B  C1                  POP   BC
FE7C  F1                  POP   AF
FE7D  C3  38  00          JP    0038
```

The NOPs are there to cause a delay, so that the colours in the border are evenly spaced out.

Another quite jolly variation is the following, which makes the screen appear to explode, by switching the colours every time a new frame appears:

Flashing Border

```
FE69  F5                  PUSH  AF
FE6A  3A  81  5C          LD    A,(5C81)
FE6D  D3  FE              OUT   (FE),A
FE6F  3D                  DEC   A
FE70  20  02              JR    NZ,FE74
FE72  3E  08              LD    A,08
FE74  32  81  5C          LD    (5C81),A
FE77  F1                  POP   AF
FE78  C3  38  00          JP    0038
```

Probably the most useful application of the changed interrupt is to allow you to provide a moving background, independent of other program activity, in the course of a game (such as the Drum Scroll program described in Chapter 12).

The Spectrum BEEP

There is one other area where some worthwhile input/output techniques can be put into practice, and that is with the beeper.

In normal operation, the beeper will only produce a single note of definite pitch and duration. Using BASIC programming and a FOR ...NEXT loop, you can get a varying succession of notes, but you can't get a smooth slide in pitch — the switch in and out of BASIC breaks the continuity. However, some simple machine code programming makes this possible.

The Spectrum does its BEEPing in a very simple way. The internal speaker is connected to one of the output ports of the Z80 processor (see p.118 of the manual). When the speaker bit (D4) is set, it activates the circuit and a click is produced at the speaker. By arranging that D4 switches on and off some hundreds of times a second, the ear interprets the clicks as a sound of definite pitch.

Clearly, with this system there can be no way, without extra hardware, of modifying the waveform and so changing the characteristics or volume of sound. However, there is one thing we can play with and that is pitch; we can (and do, whenever we set up new values for BEEP) alter the rate of clicks and so change the frequency of the note.

The way in which this program controls the rate of clicks is by setting up a count (about 100 is usually the right range) and outputting a click at the end of the count. Even a count of 100 cycles will occupy less than 10 microseconds, so the succession of clicks will produce a note well within the audible range.

If we arrange to vary the number in the count, increasing it or decreasing it regularly, we get a note that changes pitch apparently continuously, like a penny whistle.

Here is the machine code listing to do this:

Penny Whistle — up or down

```
F000 F3              DI
F001 11 10 00        LD    DE,0010
F004 26 0A           LD    H,0A
F006 3A 48 5C        LD    A,(5C48)
F009 1F              RRA
F00A 1F              RRA
F00B 1F              RRA
F00C 0E FE           LD    C,FE          OUTPUT PORT 254
F00E EE 10           XOR   10
F010 ED 79           OUT   (C),A
F012 43              LD    B,E
F013 10 FE           DJNZ  F013
F015 25              DEC   H
F016 20 F4           JR    NZ,F00C
F018 1C              INC   E
F019 15              DEC   D
F01A 20 E8           JR    NZ,F004
F01C FB              EI
F01D C9              RET
```

There are a couple of interesting points in the listing. In the first place, output port 254 sets the border colour, as well as driving the speaker (see p.160 of the manual). So, in order to preserve this colour, we collect it from the system variable BORDCR, at 5C48h, in line F006h, and then push the bits into the positions we require in the next three instructions. The XOR instruction at F00Dh switches the speaker bit on and off, as described in Chapter 6.

The DI at the start of the routine and the EI at the end are there to prevent the routine being interrupted by the keyboard scan. If this is allowed to occur, the interrupts superimpose their own 50 Hz hum on the note you are producing, spoiling the quality of the sound.

The H, DE and B registers are concerned with controlling the pitch of the note, the span of the slide and the total duration. E governs the pitch —it is the source register from which B is repeatedly loaded, to be used in a DJNZ operation to control the interval between clicks.

By incrementing E, a note will swoop down; decrementing E will make a note slide up.

H controls the number of cycles at a particular frequency, before the next increment or decrement. As a result of this function, H also controls the overall duration of the program.

D governs the number of intervals used; ie the span of the slide.

All the values can be altered experiméntally and will considerably affect the type of sound produced, as might be expected. There is no fixed 'best fit', although the numbers given make a stab at an average.

By POKEing F018h (61464d) alternately with 1Ch (28d) for 'INC E' and 1Dh (29d) for 'DEC E', you can get an up-and-down swoop, something like a wolfwhistle, or a police siren.

There is a second sound effect which relies on changing frequency, which can therefore be produced by simple BEEPing techniques. This routine outputs two different notes at once. (I had originally hoped, when the program was planned, that the result would play a chord, but it doesn't work quite like that. Presumably, to sound a chord, you have to superimpose two separate, complete waveforms, rather than two sets of on/off signals at different frequencies.)

However, the program produces some interesting beat effects, ranging from a sort of rasping twitter to quite a bell-like clang.

Double Note

```
F000  F3              DI
F001  3A 48 5C        LD    A,(5C48)
F004  1F              RRA
F005  1F              RRA
F006  1F              RRA
F007  06 00           LD    B,00
F009  0E FE           LD    C,FE
F00B  25              DEC   H            COUNTER, LOOP 1
F00C  20 06           JR    NZ,F014
F00E  EE 10           XOR   10
F010  ED 79           OUT   (C),A
F012  26 F0           LD    H,F0         SET COUNTER
F014  2D              DEC   L            COUNTER, LOOP 2
```

```
F015 20 F4          JR      NZ,F00B
F017 EE 10          XOR     10
F019 ED 79          OUT     (C),A

F01B 2E ED          LD      L,ED       SET COUNTER
F01D 10 EC          DJNZ    F00B
F01F FB             EI
F020 C9             RET
```

The program uses the same system to generate the sound as before, but this time there are two counters — one for note 1 and the other for note 2. The routine counts down on each of them alternately, and each time one of them reaches zero, it outputs to the speaker, after which the count is initialised again.

The number loaded into the B register, at F006h, controls the number of times the entire program cycles through before stopping, ie the duration of the note. Since only the B register is used, the biggest number it can deal with is 256d, so that the duration is limited. The actual length of the note also depends on the pitch — it will be longer for a deep note than it will be for a high note.

The BASIC program given below runs through a representative selection of note pairs. They vary in effect quite a lot, but the best seem to be when one note is nearly the same as the other, or nearly the same as one of its harmonics.

You could use this as part of an interrupt program, but the effect is somewhat spoiled by the fact that it must be produced in staccato bursts if your main program is to have a chance to run as well.

Run Through Double Notes

```
  100 FOR i=100 TO 250 STEP 50: FOR j=1 T
O 255
  110 POKE 61459,i: POKE 61468,j: RANDOMI
ZE USR 61440
  120 PRINT AT 10,10;i;TAB 15;j
  130 NEXT j: CLS : NEXT i
```

CHAPTER 14
Following a Machine Code Program — Hex/Dec

So far, we have mostly been considering routines which will end up as subroutines in larger-scale programs. But it can be interesting to work through a complete program in machine code and to recognise how it is put together and made accessible to the user.

I have tried to reconstruct in this chapter the thinking that went into making a program to convert decimal into hex and hex into decimal. I store this along with my assembler, so as to be able to convert addresses as required.

The program uses the Spectrum calculator to work out the hex or decimal digits. It's a moot point whether it is better to use the calculator to do the simple arithmetic required, or whether this would be better written into the program. Most simple calculations can be done using the 'ADD', 'SUB' or 'SHIFT' instructions to add, subtract, multiply or divide. For instance, 'times 10' is achieved like this:

```
Number in HL
ADD HL,HL    29    × 2
PUSH HL      E5    Store
ADD HL,HL    29    × 4
ADD HL,HL    29    × 8
POP  BC      C1    (× 2 in BC)
ADD HL,BC    09    × 10
```

To do the same thing using the calculator requires the following:

```
Number in BC
CALL STACK_BC   CD 2B 2D    Number on Calc. stack
RST 28          EF          use Calc.
                A4          stack constant, '10'
                04          multiply
                38          end Calc.
```

The number is now on the top of the calculator stack, ready to be printed,

using 'CALL PRINT_FP (CD E3 2D), the routine which prints out decimal numbers in full, including decimal points or 'E' notation as appropriate.

The calculator routines look a little obscure, because they use the Spectrum shorthand. Once the 'RST 28' instruction is reached in a program, the program no longer interprets the subsequent bytes as normal Z80 instructions, but as cues to call specific ROM routines, which do arithmetical or other tasks.

'A4' stacks the number 10d on the calculator stack, above the previous entries: '04' multiplies the two top entries on the stack together and leaves the answer in place of them: '38' signals the end of the calculation and a return to normal programming. A good machine code primer will give a complete list of these codes or 'literals'.

The 'Hex/Dec' program uses the calculator to work out the values of the numbers input, in either of the formats.

Looking at the program broadly, you can see that there will have to be two main subroutines, one to change hex into dec, and the other to change dec into hex. There will also have to be a master routine, which will switch to the required subroutine on request.

This master routine does not require any calculating — it is just a selection routine, with an input. If it gets 'H', it goes one way; if it gets 'D', it goes the other.

Let's see what that would look like:

Hex/Dec Selection

```
F000 FD CB 01 AE    RES    5, (IY+01)
F004 FD CB 01 6E    BIT    5, (IY+01)
F008 28 FA          JR     Z,F004
F00A 3A 08 5C       LD     A,(5C08)
F00D FE 48          CP     48
F00F 28 06          JR     Z,F017
F011 FE 44          CP     44
F013 28 53          JR     Z,F068
F015 18 E9          JR     F000
```

We have the same 'Wait for a key' listing which was described in Chapter 6, under the Typewriter routine. When the key code is in the A register, it is tested twice, once for 'H' (48h) and once for 'D' (44h). Either of these values gives a jump to a different address. Anything else goes back to the start.

However, left to itself, this input routine will be quite uncommunicative. It will just show a blank screen. We need some kind of message to be

printed up, as a cue for action. So we add the following coding at the beginning:

Hex/Dec Selection with Cue Message

```
F000  3E 02        LD    A,02
F002  CD 01 16     CALL  1601

F005  11 00 F1     LD    DE,F100
F008  01 1E 00     LD    BC,001E
F00B  CD 3C 20     CALL  203C
F00E  FD CB 01 AE  RES   5, (IY+01)
F012  FD CB 01 6E  BIT   5, (IY+01)
F016  28 FA        JR     Z,F012
F018  3A 08 5C     LD    A, (5C08)
F01B  FE 48        CP    48
F01D  28 06        JR     Z,F025
F01F  FE 44        CP    44
F021  28 53        JR     Z,F076
F023  18 E9        JR    F00E
```

The first two instructions set the printing for the upper screen; then we point to a message in DE of length BC and call PR__STRING (203Ch).

The coding for the message has to be entered at the F100h address. It is 30 bytes long (1Eh) and looks like this:

Hex/Dec Message — 1

```
DEFB:-
F100  EE 22 12 01 48 12 00 22
F108  EB 48 45 58 0D EE 22 12
F110  01 44 12 00 22 EB 44 45
F118  43 49 4D 41 4C 0D
```

Printing up the 'printable' characters of this message gives this result:

Hex/Dec Message — 2

```
DEFB:-
F100    .  "   .  .  H   .  .   "
F108    .  H  E  X  .  .   "  .
F110    .  D  .  .   "  .  D  E
F118  C  I  M  A  L  .
```

Part of the text seems legible, but a lot of it seems to be missing. The

'missing' bytes contain Spectrum character codes for items other than letters. 'EEh' is the code for the word 'INPUT', as you will see if you look at the manual (p.183). '12h' codes for 'FLASH' and the following '01h' is interpreted by the PR__STRING routine as 'FLASH 1'. The '12 00', two characters later, gives 'FLASH 0'. 'EB' codes for 'FOR', complete with space after it.

The entire coding corresponds to the BASIC line

```
10 PRINT "INPUT """; FLASH 1;"H"; FLAS
H 0;""" FOR HEX"'"INPUT """; FLASH 1;"D"
; FLASH 0;""" FOR DECIMAL"
```

It appears on the screen as,

INPUT "H" FOR HEX
INPUT "D" FOR DECIMAL

with the 'H' and the 'D' flashing.

You can try out this whole section of the program if you put RETs at F025h and F076h. But, remember, the routine is looking for *capital* 'D' and 'H'. We shall have to write in something to make sure that the CAPS LOCK is on.

From hex into dec

It's now time to consider the two subroutines. 'Hex into Dec' is probably the simpler, so let's look at that first.

Since the top address we are going to need to deal with is limited to FFFFh, the program will never need to use more than four hex digits. We need to organise input in a loop with '× 4' loop control. Each pass of the loop will multiply the existing total by 16d, and then add to it the value of the hex digit just input. We shall arrange to make the initial total 0 so that, after the four passes, the value of our total will be that of the four hex digits and we can print it in decimal form using PRINT__FP.

The first step is to get the value of the digit input. We can use the 'Wait for a key' routine again and get the key code into A. Next we want to make sure that what we have is a valid hex digit. Luckily, there is a little ROM subroutine at 2D1Bh, which will check to see if the input is a numeral between '0' and '9'. If the key code fails this, we have to check whether it lies between 'A' and 'F'. Only if the key code passes all these tests will the program continue. If it *is* OK, we had better print the digit, too.

Hex/Dec — Input Hex Digit

```
E000  FD CB 01 AE   RES   5, (IY+01)
E004  FD CB 01 6E   BIT   5, (IY+01)
E008  28 FA         JR       Z,E004
E00A  3A 08 5C      LD    A,(5C08)
E00D  CD 1B 2D      CALL  2D1B
E010  30 08         JR       NC,E01A
E012  FE 41         CP    41          'A'
E014  38 EA         JR       C,E000
E016  FE 47         CP    47          'F'
E018  30 E6         JR       NC,E000
E01A  F5            PUSH  AF
E01B  D7            RST   10          PRINT
E01C  F1            POP   AF
E01D  C9            RET
```

Now to do the multiplying and adding. We need to have the value '16' ready somewhere in the calculator, to do our multiplying. We also need to have a zero on the calculator stack at the start of operations. Both of these preparations had better be made before we start inputting digits.

The best place for the '16' is in the calculator's memory. It can stay there as long as wanted and be called out on to the stack, by a single literal, each time we need to use it. You get it to the memory by stacking it and then using the literal 'C0'. So our opening gambit is:

Hex/Dec — Value 16d to Calculator Memory

```
E000  3E 10         LD    A,10
E002  CD 28 2D      CALL  2D28
E005  EF            RST   28
E006  C0       stk  mem 0
E007  38       end  calc.
E008  AF            XOR   A
E009  CD 28 2D      CALL  2D28
```

After INPUT, we have a value in A which is either '0' to '9' or 'A' to 'F'. However, the value of key code 'A' is not one more than key code '9' — it is eight more. We have to do a little more adjusting, before we can be sure that we have got the value right. So, continuing with our complete listing, to date, this is:

Hex/Dec — Hex Input, Opening Section

```
E000  3E 10         LD    A,10
```

```
E002 CD 28 2D        CALL  2D28
E005 EF              RST   28
E006 C0        stk  mem  0
E007 38        end  calc.
E008 AF              XOR   A
E009 CD 28 2D        CALL  2D28

E00C FD CB 01 AE  RES   5, (IY+01)
E010 FD CB 01 6E  BIT   5, (IY+01)
E014 28 FA           JR      Z,E010
E016 3A 08 5C     LD    A,(5C08)
E019 CD 1B 2D        CALL  2D1B

E01C 30 08           JR      NC,E026
E01E FE 41           CP    41
E020 38 EA           JR      C,E00C
E022 FE 47           CP    47
E024 30 E6           JR      NC,E00C

E026 F5              PUSH  AF
E027 D7              RST   10
E028 F1              POP   AF

E029 FE 41           CP    41
E02B 38 02           JR      C,E02F
E02D D6 07           SUB   07
```

Now we had better PUSH our value in A, as we are going to do some arithmetic which will corrupt this register. The first thing is to multiply the existing calculator stack value by 16.

```
EF     RST 28
E0     get MEM,0 on stack (this is '16')
04     multiply
38     end Calc.
```

Now we can POP AF again and stack it.

```
F1              POP  AF
CD 26 2D   CALL "STK_DIGIT" +
```

The last call, to 2D26h, is a modified CALL to STK_DIGIT, which stacks the value of a valid ASCII numeral. Since our own offering may not be a numeral (it may be 'A' to 'F') but one we know is valid, we skip

the checking procedure, between 2D22h and 2D25h, which might otherwise reject it.

Now back to the calculator again.

EF	RST 28
0F	add
38	end Calc.

This adds the original value on the stack (multiplied by 16) to the new value just input, and leaves the result as the top item on the stack.

If we arrange to do this four times, we have the value of a four-digit hex number on the stack, which can be printed in decimal with the PRINT__FP routine.

So, for the grand finale, which only needs a printed input cue to complete it:

Hex/Dec — Hex Input Complete

```
E000 3E 10            LD   A,10
E002 CD 28 2D         CALL 2D28          16D TO CALC. MEM 0
E005 EF               RST  28
E006 C0         stk mem 0
E007 38         end calc.
E008 AF               XOR  A
E009 CD 28 2D         CALL 2D28          0 TO CALC. STACK

E00C 06 04            LD   B,04          4 DIGIT COUNT
E00E C5               PUSH BC

E00F FD CB 01 AE RES  5, (IY+01)
E013 FD CB 01 6E BIT  5, (IY+01)         WAIT FOR KEY
E017 28 FA            JR      Z,E013
E019 3A 08 5C         LD   A,(5C08)
E01C CD 1B 2D         CALL 2D1B          CHECK '0'–'9'?

E01F 30 08            JR      NC,E029
E021 FE 41            CP   41
E023 38 EA            JR      C,E00F      CHECK 'A'–'F'?
E025 FE 47            CP   47
E027 30 E6            JR      NC,E00F

E029 F5               PUSH AF            PRINT HEX DIGIT
E02A D7               RST  10
E02B F1               POP  AF

E02C FE 41            CP   41
E02E 38 02            JR      C,E032
```

```
E030  D6 07          SUB   07
E032  F5             PUSH  AF

E033  EF             RST   28
E034  E0       get mem 0              MULTIPLY STACK TOP
E035  04  *    multiply               BY 16D
E036  38       end calc.
E037  F1             POP   AF
E038  CD 26 2D       CALL  2D26       STACK VALUE IN A
E03B  EF             RST   28
E03C  0F       add                    ADD NEW VALUE TO
                                      STACK TOP
E03D  38       end calc.

E03E  C1             POP   BC         COUNT
E03F  10 CD          DJNZ  E00E

E041  3E 06          LD    A,06       PRINT 'TAB 16' (=,)
E043  D7             RST   10
E044  CD E3 2D       CALL  2DE3       PRINT DECIMAL NUMBER
E047  C9             RET
```

From dec to hex

The second subroutine, to go from decimal to hex, follows much the same lines, except that we multiply the total by 10d, rather than 16d, on each pass. Also, we have no ready-made routine for printing out hex digits, so we shall have to write one ourselves.

Each time we extract a hex digit from the number we are working on, we shall produce a value in A which must lie between 0–15d (0–Fh).

Simply by adding 48d (30h), we shall get the codes for the decimal numerals. In the case of the values from 10d to 15d, we have to arrange to add a further 7 to bring it up to the codes for 'A' to 'F'. So the coding will look like this:

Dec/Hex — Print Hex Digit

```
E200  FE 0A          CP    0A
E202  38 02          JR    C,E206
E204  C6 07          ADD   A,07
E208  D7             RST   10
E209  C9             RET
```

We also need to work out how we are going to extract these hex digits from the value of the complete number entered. This turns out to be very easy. Suppose that our value is held in a register pair (it will need to be a pair, as the maximum value we shall be dealing with, FFFFh, needs more than one register to hold it). The value will be in the form 'xxxx', where

each 'x' is a hex digit. So we just need a simple program, to extract each nibble in turn from the register pair and send it off to the printing subroutine which we have just written.

Assuming that the value is in BC, the following would do the job:

Dec/Hex — Print Hex Digits (1)

```
E100  78              LD    A,B
E101  E6 F0           AND   F0
E103  1F              RRA
E104  1F              RRA
E105  1F              RRA
E106  1F              RRA
E107  CD 00 E2        CALL  E200    PRINT ROUTINE
E10A  78              LD    A,B
E10B  E6 0F           AND   0F
E10D  CD 00 E2        CALL  E200    PRINT ROUTINE
E110  79              LD    A,C
E111  E6 F0           AND   F0
E113  1F              RRA
E114  1F              RRA
E115  1F              RRA
E116  1F              RRA
E117  CD 00 E2        CALL  E200    PRINT ROUTINE
E11A  79              LD    A,C
E11B  E6 0F           AND   0F
E11D  CD 00 E2        CALL  E200    PRINT ROUTINE
E120  C9              RET
```

The only possible drawback to this version is that it is not relocatable — it relies on a subroutine CALL, which has to be at a fixed address. We might be able to get rid of this, if we arranged a '× 4' loop and shifted the nibbles into A, rather than masking them with the AND.

Dec/Hex — Print Hex Digits (2)

```
E100  1E 04           LD    E,04
E102  16 04           LD    D,04
E104  AF              XOR   A
E105  CB 11           RL    C
E107  CB 10           RL    B
E109  17              RLA
E10A  15              DEC   D
E10B  20 F8           JR    NZ,E105
E10D  FE 0A           CP    0A
E10F  38 02           JR    C,E113
```

```
E111  C6 07          ADD   A,07
E113  C6 30          ADD   A,30
E115  D7             RST   10
E116  1D             DEC   E
E117  20 E9          JR    NZ,E102
E119  C9  ˙          RET
```

As it turns out, the second version is shorter, as well as being relocatable, although it does use an extra register. On the whole, it seems the better one, so let's adopt it.

Now for the input and value extracting.

Most of the first part of the routine is virtually a carbon copy of the hex/dec one. You do not have to place 10h (16d) in the calculator memory: there is a constant 0Ah (10d) permanently on call among the other constants in the Spectrum system. Also, you no longer have to check the digits to see that they fall between 'A' and 'F' — they can only be ordinary numerals. So the routine, up to the print section, looks like this:

Dec/Hex — Input Decimal, part 1

```
E200  AF              XOR   A
E201  CD 28 2D        CALL  2D28
E204  06 05           LD    B,05          MAX. NO. OF DIGITS
E206  C5              PUSH  BC
E207  FD CB 01 AE     RES   5,  (IY+01)
E20B  FD CB 01 6E     BIT   5,  (IY+01)
E20F  28 FA           JR    Z,E20B
E211  3A 08 5C        LD    A,(5C08)      CHECK '0'–'9'
E214  CD 1B 2D        CALL  2D1B
E217  38 EE           JR    C,E207
E219  F5              PUSH  AF
E21A  D7              RST   10            PRINT DEC. DIGIT
E21B  F1              POP   AF
E21C  F5              PUSH  AF
E21D  EF              RST   28
E21E  A4              constant 10
E21F  04              multiply
E220  38              end calc.
E221  F1              POP   AF
E222  CD 22 2D        CALL  2D22          STK_DIGIT
E225  EF              RST   28
E226  0F              add
E227  38              end calc.
E228  C1              POP   BC
```

```
E229 10 DB          DJNZ E206
E22B CD A2 2D        CALL 2DA2
E22E C9              RET
```

The last CALL, to 2DA2h, is to the ROM FP__TO__BC routine. This
puts the value of the floating point number at the top of the calculator
stack into the BC register. From here, as we have found, it is a simple
matter to print out the value in hex. Before the printout, we once again do
a 'TAB 16;', by PRINTing CHR 06h.

Dec/Hex — Decimal Input, Complete

```
E200 AF              XOR  A
E201 CD 28 2D        CALL 2D28
E204 06 05           LD   B,05
E206 C5              PUSH BC
E207 FD CB 01 AE     RES  5, (IY+01)
E20B FD CB 01 6E     BIT  5, (IY+01)
E20F 28 FA           JR   Z,E20B
E211 3A 08 5C        LD   A,(5C08)
E214 FE 0D           CP   0D
E216 20 03           JR   NZ,E21B
E218 C1             .POP  BC
E219 18 17           JR   E232
E21B CD 1B 2D        CALL 2D1B
E21E 38 E7           JR   C,E207
E220 F5              PUSH AF
E221 D7              RST  10
E222 F1              POP  AF
E223 F5              PUSH AF
E224 EF              RST  28
E225 A4         constant 10
E226 04         multiply
E227 38         end calc.
E228 F1              POP  AF
E229 CD 22 2D        CALL 2D22
E22C EF              RST  28
E22D 0F         add
E22E 38         end calc.
E22F C1              POP  BC
E230 10 D4           DJNZ E206
E232 3E 06           LD   A,06
E234 D7              RST  10
E235 CD A2 2D        CALL 2DA2
E238 1E 04           LD   E,04
```

```
E23A  16 04      LD    D,04
E23C  AF         XOR   A
E23D  CB 11      RL    C
E23F  CB 10      RL    B
E241  17         RLA
E242  15         DEC   D
E243  20 F8      JR    NZ,E23D
E245  FE 0A      CP    0A
E247  38 02      JR    C,E24B
E249  C6 07      ADD   A,07
E24B  C6 30      ADD   A,30
E24D  D7         RST   10
E24E  1D         DEC   E
E24F  20 E9      JR    NZ,E23A
E251  C9         RET
```

This has assembled all the main components of the complete program. We still have to write labels for the two subroutines and initialise with a CAPS LOCK.

Looked at schematically, the program when grouped together has this shape:

Initialise   F000–F011
   Set CAPS
   Open screen channel
Message
Menu   F012–F028
   Wait for key
   Choose 'H'
   Choose 'D'
Hex/Dec message   F029–F031
Set up calculator   F032–F03D
Hex value input & calculate   F03E–F072
End routine   F073–F07A
   Print 'Tab 16' and number
   Jump to 'finalise'
Dec/Hex message   F07B–F083
Set up calculator   F084–F087
Dec value input & calculate   F088–F0B5
Calculate & print hex   F0B6–F0D4
Finalise   F0D5–F0E3
   Wait for key
   Reset CAPS
   Return

Here it is, printed in full.
Hex/Dec

```
F000  FD CB 30 DE   SET   3, (IY+30)
F004  3E 02         LD    A,02
F006  CD 01 16      CALL  1601
F009  11 00 F1      LD    DE,F100
F00C  01 1E 00      LD    BC,001E
F00F  CD 3C 20      CALL  203C
F012  FD CB 01 AE   RES   5, (IY+01)
F016  FD CB 01 6E   BIT   5, (IY+01)
F01A  28 FA         JR    Z,F016
F01C  3A 08 5C      LD    A,(5C08)
F01F  FE 48         CP    48
F021  28 06         JR    Z,F029
F023  FE 44         CP    44
F025  28 53         JR    Z,F07A
F027  18 E9         JR    F012

F029  11 40 F1      LD    DE,F140
F02C  01 26 00      LD    BC,0026
F02F  CD 3C 20      CALL  203C

F032  3E 10         LD    A,10
F034  CD 2B 2D      CALL  2D2B
F037  EF            RST   28
F038  C0        stk mem 0
F039  38        end calc.
F03A  AF            XOR   A
F03B  CD 2B 2D      CALL  2D2B

F03E  06 04         LD    B,04
F040  C5            PUSH  BC
F041  FD CB 01 AE   RES   5, (IY+01)
F045  FD CB 01 6E   BIT   5, (IY+01)
F049  28 FA         JR    Z,F045
F04B  3A 08 5C      LD    A,(5C08)
F04E  CD 1B 2D      CALL  2D1B
F051  30 08         JR    NC,F05B
F053  FE 41         CP    41
F055  38 EA         JR    C,F041
F057  FE 47         CP    47
F059  30 E6         JR    NC,F041
F05B  F5            PUSH  AF
F05C  D7            RST   10
F05D  F1            POP   AF
```

```
F05E  FE 41              CP    41
F060  38 02              JR       C,F064
F062  D6 07              SUB   07
F064  F5                 PUSH  AF
F065  EF                 RST   28
F066  E0    *  get mem 0
F067  04       multiply
F068  38       end calc.
F069  F1                 POP   AF
F06A  CD 26 2D           CALL  2D26
F06D  EF                 RST   28
F06E  0F       add
F06F  38       end calc.
F070  C1                 POP   BC
F071  10 CD              DJNZ  F040

F073  3E 06              LD    A,06
F075  D7                 RST   10
F076  CD E3 2D           CALL  2DE3
F079  18 5A              JR       F0D5
F07B  11 1E F1           LD    DE,F11E
F07E  01 22 00           LD    BC,0022
F081  CD 3C 20           CALL  203C

F084  AF                 XOR   A
F085  CD 28 2D           CALL  2D28

F088  06 05              LD    B,05
F08A  C5                 PUSH  BC
F08B  FD CB 01 AE        RES   5, (IY+01)
F08F  FD CB 01 6E        BIT   5, (IY+01)
F093  28 FA              JR       Z,F08F
F095  3A 08 5C           LD    A,(5C08)
F098  FE 0D              CP    0D
F09A  20 03              JR       NZ,F09F
F09C  C1                 POP   BC
F09D  18 17              JR       F0B6
F09F  CD 1B 2D           CALL  2D1B
F0A2  38 E7              JR       C,F08B
F0A4  F5                 PUSH  AF
F0A5  D7                 RST   10
F0A6  F1                 POP   AF
F0A7  F5                 PUSH  AF
F0A8  EF                 RST   28
F0A9  A4       constant 10
```

```
F0AA  04          multiply
F0AB  38          end calc.
F0AC  F1                    POP    AF
F0AD  CD 22 2D              CALL   2D22
F0B0  EF                    RST    28
F0B1  0F          add
F0B2  38          end calc.
F0B3  C1                    POP    BC
F0B4  10 D4                 DJNZ   F08A
F0B6  3E 06                 LD     A,06
F0B8  D7                    RST    10
F0B9  CD A2 2D              CALL   2DA2
F0BC  1E 04                 LD     E,04
F0BE  16 04                 LD     D,04
F0C0  AF                    XOR    A
F0C1  CB 11                 RL     C
F0C3  CB 10                 RL     B
F0C5  17                    RLA
F0C6  15                    DEC    D
F0C7  20 F8                 JR     NZ,F0C1
F0C9  FE 0A                 CP     0A
F0CB  38 02                 JR     C,F0CF
F0CD  C6 07                 ADD    A,07
F0CF  C6 30                 ADD    A,30
F0D1  D7                    RST    10
F0D2  1D                    DEC    E
F0D3  20 E9                 JR     NZ,F0BE

F0D5  FD CB 01 AE  RES   5, (IY+01)
F0D9  FD CB 01 6E  BIT   5, (IY+01)
F0DD  28 FA                 JR     Z,F0D9
F0DF  FD CB 30 9E  RES   3, (IY+30)
F0E3  C9                    RET
```

The data for the messages are arranged as follows:

```
DEFB: -
F100  EE  22  12  01  48  12  00  22
F108  EB  48  45  58  0D  EE  22  12
F110  01  44  12  00  22  EB  44  45
F118  43  49  4D  41  4C  0D  14  01
F120  44  45  43  49  4D  41  4C  14
F128  00  EE  55  50  CC  35  20  44
F130  49  47  49  54  53  0D  2B  20
F138  22  45  4E  54  45  52  22  0D
```

```
F140  14  01  48  45  58  14  00  EE
F148  34  20  44  49  47  49  54  53
F150  0D  28  57  49  54  48  20  4C
F158  45  41  44  49  4E  47  20  5A
F160  45  52  4F  53  29  0D
```

yielding the following characters:

```
F100   .    "    .    .    H    .    .    "
F108   .    H    E    X    .    .    "    .
F110   .    D    .    .    "    .    D    E
F118   C    I    M    A    L    .    .    .
F120   D    E    C    I    M    A    L    .
F128   .    .    U    P    .    5         D
F130   I    G    I    T    5    .    +
F138   "    E    N    T    E    R    "    .
F140   .    .    H    E    X    .    .    .
F148   4         D    I    G    I    T    5
F150   .    (    W    I    T    H         L
F158   E    A    D    I    N    G         Z
F160   E    R    O    5    )    .
```

When printed out, the messages should look like this:

```
INPUT "H" FOR HEX
INPUT "D" FOR DECIMAL
```

```
HEX INPUT 4 DIGITS
(WITH LEADING ZEROS)
```

```
DECIMAL INPUT UP TO 5 DIGITS
+ "ENTER"
```

The whole program can be used by a machine code CALL to F000h. When used from a BASIC program, you would need 'RANDOMIZE USR 61440'.

APPENDIX A
Machine Code Routines

PaintCODE

This machine code program will INK in any line drawn figure on the Spectrum, so enabling you to produce solid coloured shapes which otherwise might be difficult.

Unfortunately, like most good things in life, it is not perfect. This is a result of the way in which the routine tackles this problem.

The routine paints the figure by looking at the screen a line at a time and joining together any pairs of dots it finds. It does this by setting all the bits in between the pairs of dots. Unfortunately, it sometimes gets in a muddle when it comes across a line with an odd number of dots, when it has no way of telling which dots go with each other: it produces lines where there should be blanks and vice versa.

To eliminate this would require a much extended program which looked at the lines on either side, as well as the one being changed. Life is too short, considering the use the program gets.

PaintCODE

```
F000  21 00 40      LD    HL,4000
F003  1E C0         LD    E,C0
F005  0E 20         LD    C,20
F007  06 08         LD    B,08
F009  7E            LD    A,(HL)
F00A  07            RLCA
F00B  38 11         JR    C,F01E
F00D  10 FB         DJNZ  F00A
F00F  77            LD    (HL),A
F010  23            INC   HL
F011  0D            DEC   C
F012  20 F3         JR    NZ,F007
F014  1D            DEC   E
F015  20 EE         JR    NZ,F005
F017  C9            RET
F018  06 08         LD    B,08
F01A  7E            LD    A,(HL)
```

```
F01B  07          RLCA
F01C  30  08      JR      NC,F026

F01E  10  FB      DJNZ    F01B
F020  23          INC     HL
F021  0D          DEC     C
F022  28 -F0      JR      Z,F014
F024  18  F2      JR      F018
F026  C5          PUSH    BC
F027  E5          PUSH    HL
F028  F5          PUSH    AF
F029  18  06      JR      F031
F02B  06  08      LD      B,08
F02D  7E          LD      A,(HL)
F02E  07          RLCA
F02F  38  08      JR      C,F039
F031  10  FB      DJNZ    F02E
F033  23          INC     HL
F034  0D          DEC     C
F035  20  F4      JR      NZ,F02B
F037  18  16      JR      F04F
F039  F1          POP     AF
F03A  E1          POP     HL
F03B  C1          POP     BC
F03C  18  04      JR      F042
F03E  06  08      LD      B,08
F040  7E          LD      A,(HL)
F041  07          RLCA
F042  CB  C7      SET     0,A
F044  38  11      JR      C,F057
F046  10  F9      DJNZ    F041
F048  77          LD      (HL),A
F049  23          INC     HL
F04A  0D          DEC     C
F04B  20  F1      JR      NZ,F03E
F04D  18  03      JR      F052
F04F  F1          POP     AF
F050  F1          POP     AF
F051  F1          POP     AF
F052  18  C0      JR      F014
F054  07          RLCA
F055  38  04      JR      C,F05B
F057  10  FB      DJNZ    F054
F059  18  B4      JR      F00F
F05B  F5          PUSH    AF
```

```
F05C  C5           PUSH  BC
F05D  18  01        JR    F060
F05F  07            RLCA
F060  10  FD        DJNZ  F05F
F062  77            LD    (HL),A
F063  C1            POP   BC
F064  F1            POP   AF
F065  18  B7        JR    F01E
```

Mapper

In order to draw the shapes which can be filled in with the PaintCODE routine, you might use a program like this one:

Mapper

```
   5 POKE 23660,5: LET a=0
  10 INPUT "x";x;"y";y
  20 PLOT INVERSE a;x,y
  30 PRINT #1;AT 0,12;"3=</  4=<\  9=>/  0=
>\ "
  40 PAUSE 0
  50 LET y$=INKEY$
  60 LET x=x-(y$="3")-(y$="4")-(y$="5")+
(y$="8")+(y$="9")+(y$="0")
  70 LET y=y-(y$="3")+(y$="4")-(y$="6")+
(y$="7")+(y$="9")-(y$="0")
  80 IF y$="1" THEN RANDOMIZE USR (5+PEE
K 23635+256*PEEK 23636)
  90 LET x=x-(x>255)+(x<0): LET y=y-(y>1
75)+(y<0)
 100 GO TO 20
```

It is very similar to the Draw a Sprite program in Chapter 8, but the programming is more compact and it will draw over the whole screen.

PaintCODE has to be put into a line 1 REM statement, as described in Chapter 1. Once again, it is accessed indirectly, using the system variable PROG, so as to get the right address whether or not Interface 1 is in place (see Chapters 1 and 6).

You can also use Mapper to touch out those unwanted lines where PaintCODE has got its sums wrong.

Find z$

You often need to find the address of a BASIC program string in the

course of a machine code program. There are routines to discover the address of the actual string, but they have to be set up for each individual case, and there is no really suitable ROM routine.

There is a lot to be said for making a rule to copy the string under consideration into another string, which can be consistently addressed by the machine code. A convenient string to choose is 'z$'. If you put the chosen string, or element of a string array, into z$ immediately before calling the machine code routine with 'USR xxxxx', z$ will always be the last string in the variables area, before E_LINE. You can then be sure of finding its address by a simple search routine.

Unfortunately, the code which signals the start of a simple string in the variables is the same as the code for the capital version of the letter which names the string — in this case, capital 'Z'. Because there is a chance that this letter might feature in the string, we have to check that there is no mistake. This can be done by checking the next character but one, which will almost certainly not be a printable character if the letter signals a string — unless the string is extremely long (see the diagram on p.168 of the manual).

If, for some reason, the search fails and 'Z' cannot be found, the program will drop into BASIC with an error message.

Find z$

```
F000  3E 5A        LD    A,5A
F002  A7           AND   A
F003  2A 59 5C     LD    HL,(5C59)
F006  ED 4B 4B 5C  LD    BC,(5C4B)
F00A  E5           PUSH  HL
F00B  ED 42        SBC   HL,BC
F00D  44           LD    B,H
F00E  4D           LD    C,L
F00F  0C           INC   C
F010  E1           POP   HL
F011  ED B9        CPDR
F013  28 02        JR    Z,F017
F015  CF           RST   08
F016  01           DEFB  01          ERROR REPORT 2
F017  23           INC   HL
F018  23           INC   HL
F019  23           INC   HL
F01A  3E 1F        LD    A,1F
F01C  BE           CP    (HL)
F01D  2B           DEC   HL
F01E  2B           DEC   HL
F01F  2B           DEC   HL
```

```
F020 3E 5A        LD     A,5A
F022 38 ED        JR     C,F011
F024 23           INC    HL
F025 23           INC    HL
F026 4E           LD     C,(HL)
F027 23           INC    HL
F028 46           LD     B,(HL)
F029 23           INC    HL
F02A C9           RET
```

The address of the first character of z$ is returned in HL and the length of the string in BC.

Pip routine

One of the attractive touches about the Spectrum is the way in which it makes a little click when a key is depressed, like a mechanical typewriter. This is a great help to people like myself, who are not touch typists and cannot watch the screen when they are using the keyboard. It verifies that a key has actually been struck.

The routine below is a straight copy of the method used in the ROM to produce the click, so that you can have the same advantage when inputting machine code routines.

All the prime registers and the IX register are affected by the routine, hence the mass of PUSHes and POPs which surround the three instructions.

This routine can be set up as a subroutine in your machine code programs, and CALLed after every input from the keyboard.

Pip routine

```
F000 DD E5        PUSH IX
F002 E5           PUSH HL
F003 D5           PUSH DE
F004 C5           PUSH BC
F005 11 00 00     LD   DE,0000
F008 21 CB 00     LD   HL,00CB
F00B CD B5 03     CALL 03B5          ROM BEEPER ROUTINE
F00E C1           POP  BC
F00F D1           POP  DE
F010 E1           POP  HL
F011 DD E1        POP  IX
F013 C9           RET
```

Alternatively, if you are using a BASIC program and can spare a couple of UDG characters, you could try the following:

Pip BASIC

```
  10 DATA 17,0,0,33,203,0,205,181,3,201
  20 FOR j=0 TO 9: READ n: POKE USR "A"+
j,n: NEXT j

  99 REM DEMONSTRATION
 100 PAUSE 0: RANDOMIZE USR USR "A": GO
TO 100
```

This allows you to make a click when you are taking an INKEY$ value after a 'PAUSE 0', in place of the normal 'INPUT...'.

I rather like the 'USR USR'. Of course, USR 'A' (or any other UDG letter) is simply an address in the upper RAM. And you are not limited to POKEing it with a sequence of eight positions to make a graphic character.

Since we are dealing with a BASIC program, we don't have to bother about saving and restoring all the registers.

APPENDIX B
Some ROM Subroutines

0010
PRINT__A__1

The address called by 'RST 10' (the opcode D7), which leads to the main printing routine. This must be the most used routine in the Spectrum's book. It will print to the screen in the current print position (upper or lower screen), or send to the printer, according to the channel set, any printable character held in A. It will print the expanded labels corresponding to the appropriate character codes. It will evaluate the control characters in the first 20 positions of the character set. In fact, it works like an Aladdin's lamp for the Spectrum display.

0028
FP__CALC

Another entry point to extended routines, this one is called by 'RST 28' (opcode EF). 'RST 28' accesses the calculator, which is a world of its own, with its own set of rules.

0038
MASK__INT

The maskable interrupt routine. Normally activated every 20 microseconds to update the Spectrum clock and scan the keyboard.

03B5
BEEPER

HL must hold the pitch and DE the duration. Also used to produce the keyboard click.

04C2
SA__BYTES

This is the complete SAVE routine, used to save all bytes. Also used for the header.

0556
LD_BYTES

LOAD routine. IX holds the start address and DE the number of bytes to be LOADed. The carry flag must be set for LOADing: if it is reset, the routine will VERIFY.

0970
SA_CONTRL

SAVE routine. IX holds the start address and DE the number of bytes to be SAVEd. This routine contains the whole normal SAVE routine, including the 'Wait for a key' pause. Entry at 0984h skips the wait for input.

0D6B
CLS

Channel 'S' (No. 2 — upper screen) must be opened before and after CALLing.

0DD9
CL_SET

Calculates and sets DF_CC when S_POSN values are held in BC.

0E44
CL_LINE

This routine will clear a specified number of lines starting from the bottom (line 24). The number of lines to be cleared must be specified in B. '06 18' would clear the whole screen.

0EAC
COPY

The printer channel (No. 3) must be opened first (see below).

1601
CHAN_OPEN

Open channel routine. Used in the form, 'LD A,x   CALL 1601': x = 1 for lower screen, x = 2 for upper screen, x = 3 for printer.

16C5
SET_STK

This routine clears the calculator stack. Used in calculator operations.

203C
PR_STRING

Prints string ($) with start address in DE and length in BC. The routine uses 'RST 10', so control characters, etc., will also be activated.

22AA
PIXEL_AD

Takes POINT coordinate in BC and delivers the display file byte position in HL, also the point position within the byte in A.

2D1B
NUMERIC

Verifies that A holds an ASCII digit, between 0 and 9. Carry flag set, if valid. Used in STK__NUM routine.

2D22
STK__DIGIT

Valid ASCII digit in A passed to calculator stack as floating point number.

2D28
STACK__A

Value in A passed to calculator stack.

2D2B
STACK__BC

Value in BC passed to calculator stack.

2DA2
FP__TO__BC

This is an 'unstacking' routine. Passes the value of the floating point integer at the top of the stack to the BC register.

2DD5
FP__TO__A

Another unstacking routine. Passes the value of the floating point integer at the top of the calculator stack to the A register.

2DE3
PRINT__FP

The floating point number on the top of the calculator stack is printed in decimal (including decimal point) or 'E' notation, as appropriate.

These routines represent only a small number of the total routines stored in the ROM and used in the implementation of the Spectrum BASIC programming. Fuller guides to the Spectrum ROM will give further information on this important subject.
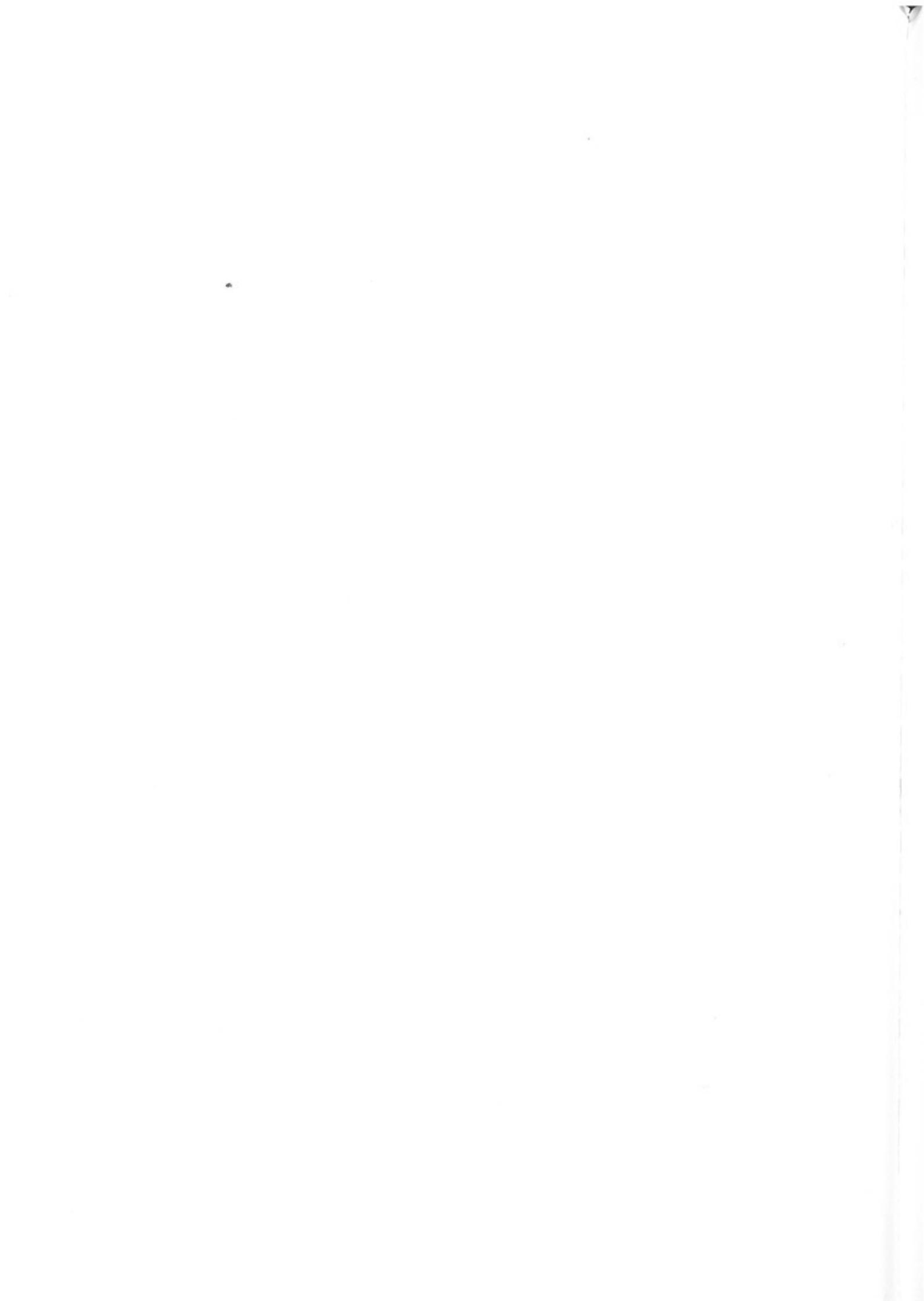
Index

This index was prepared on a ZX Spectrum and printed on a ZX printer. For technical reasons, it has been reset here, but a sample of the original copy is given below. The program uses many of the routines and enhanced characters described in this book. For those interested, further particulars can be obtained from the author, c/o Sunshine Books.

Other titles from Sunshine

SPECTRUM BOOKS

Spectrum Adventures
Tony Bridge & Roy Carnell   ISBN 0 946408 07 6   **£5.95**

ZX Spectrum Astronomy
Maurice Gavin   ISBN 0 946408 24 6   **£6.95**

Spectrum Machine Code Applications
David Laine   ISBN 0 946408 17 3   **£6.95**

The Working Spectrum
David Lawrence   ISBN 0 946408 00 9   **£5.95**

Inside Your Spectrum
Jeff Naylor & Diane Rogers   ISBN 0 946408 35 1   **£6.95**

Master your ZX Microdrive
Andrew Pennell   ISBN 0 946408 19 X   **£6.95**

COMMODORE 64 BOOKS

Graphic Art for the Commodore 64
Boris Allan   ISBN 0 946408 15 7   **£5.95**

DIY Robotics and Sensors on the Commodore Computer
John Billingsley   ISBN 0 946408 30 0   **£6.95**

Artificial Intelligence on the Commodore 64
Keith & Steven Brain   ISBN 0 946408 29 7   **£6.95**

Commodore 64 Adventures
Mike Grace   ISBN 0 946408 11 4   **£5.95**

Business Applications for the Commodore 64
James Hall   ISBN 0 946408 12 2   **£5.95**

Mathematics on the Commodore 64
Czes Kosniowski   ISBN 0 946408 14 9   **£5.95**

Advanced Programming Techniques on the Commodore 64
David Lawrence   ISBN 0 946408 23 8   **£5.95**

There are 49,152 separate dots which go to make up the picture from a ZX Spectrum on the television screen. This book explains how to get every one of them under your control, so that you can produce the displays you want, when you want and where you want.

Moving sprites, transparent sprites, big letters, small letters, scrolls and windows — even the incredible shrinking screen — are dealt with in clear detail, with helpful text illustrations, most of them drawn by the Spectrum itself.

Every chapter contains short BASIC programs, or machine code routines to perform specific tasks, which can be incorporated in a range of programs of your own devising, to make them clearer, more colourful, more exciting.

For anyone interested in machine code, or in ways to squeeze more from a program, this is the book for you.

John Durst is a former film director who decided to confront animation problems on his Spectrum. He is a regular contributor to Popular Computing Weekly.