How to Write ZX Spectrum Games

Version 1.0

Jonathan Cauldwell

Document History

Version 0.1 - February 2006
Version 0.2 - January 2007
Version 0.3 - December 2008
Version 0.4 - July 2009
Version 0.5 - October 2010
Version 0.6 - April 2014
Version 1.0 - June 2015

Copyright

Introduction

So you've read the Z80 documentation, you know how the instructions affect the registers and now you want to put this knowledge to use. Judging by the number of emails I have received asking how to read the keyboard, calculate screen addresses or emit white noise from the beeper it has become clear that there really isn't much in the way of resources for the new Spectrum programmer. This document, I hope, will grow to fill this void in due course. In its present state it is clearly years from completion, but in publishing the few basic chapters that exist to date I hope it will be of help to other programmers.

The ZX Spectrum was launched in April 1982, and by today's standards is a primitive machine. In the United Kingdom and a few other countries it was the most popular games machine of the 1980s, and through the joys of emulation many people are enjoying a nostalgic trip back in time with the games of their childhoods. Others are only now discovering the machine for the first time, and some are even taking up the challenge of writing games for this simple little computer. After all, if you can write a decent machine code game for a 1980s computer there probably isn't much you couldn't write.

Purists will hate this document, but writing a game isn't about writing "perfect" Z80 code - as if there were such a thing. A Spectrum game is a substantial undertaking, and you won't get around to finishing it if you are too obsessed with writing the very best scoring or keyboard reading algorithms. Once you've written a routine that works and doesn't cause problems elsewhere, move on to the next routine. It doesn't matter if it's a little messy or inefficient, because the important part is to get the gameplay right. Nobody in his right mind is going to disassemble your code and pick faults with it.

The chapters in this document have been ordered in a way designed to enable the reader to start writing a simple game as soon as possible. Nothing beats the thrill of writing your first full machine-code game, and I have set out this manual in such a way as to cover the very basic minimum requirements for this in the first few chapters. From there we move on to cover more advanced methods which should enable the reader to improve the quality of games he is capable of writing.

Throughout this document a number of assumptions have been made. For a start, it is assumed that the reader is familiar with most Z80 opcodes and what they do. If not there are plenty of guides around which will explain these far better than I could ever do. Learning machine code instructions isn't difficult, but knowing how to put them together in meaningful ways can be. Familiarity with the load (ld), compare (cp), and conditional jump (jp z / jp c / jp nc) instructions is a good place to start. The rest will fall into place once these are learned.

Tools

These days we have the benefit of more sophisticated hardware, and there is no need to develop software on the machine for which it is intended. There are plenty of adequate cross-assemblers around which will allow Spectrum software to be developed on a PC and the binary file produced can then be imported into an emulator - SPIN is a popular emulator which has support for this feature.

For graphics there's a tool called SevenUp which I use, and can thoroughly recommend. This can convert bitmaps into Spectrum images, and allows the programmer to specify the order in which sprites or other graphics are sorted. Output can be in the form of a binary image, or source code. Another popular program is TommyGun.

Music wise I'd recommend the SoundTracker utility which can be downloaded from the World of Spectrum archives. There's a separate compiler program you'll also need. Bear in mind that these are Spectrum programs, not PC tools and need to be run on an emulator. Beepola is an excellent tool for producing 48K beeper music, and runs on Windows PCs

As editors and cross-compilers go I am not in a position to recommend the best available, because I use an archaic editor and Z80 Macro cross-assembler written in 1985, running in DOS windows. Neither are tools I would recommend to others. If you require advice on which tools might be suitable for you, I suggest you try the World of Spectrum development forum at http://www.worldofspectrum.org/forums/. This friendly community has a wide range of experience and is always willing to help.

Personal Quirks

Over the many years that I have been writing Spectrum software a number of habits have formed which may seem odd. The way I order my coordinates, for example, does not follow the conventions of mathematics. My machine code programs follow the Sinclair BASIC convention of PRINT AT x,y; where x refers to the number of character cells or pixels from the top of the screen and y is the number of characters or pixels from the left edge. If this seems confusing at first I apologise, but it always seemed a more logical way of ordering things and it just stuck with me. Some of my methodology may seem unusual in places, so where you can devise a better way of doing something by all means go with that instead.

One other thing: commenting your code as you go along is important, if not essential. It can be hellishly difficult trying to find a bug in an uncommented routine you wrote only a few weeks ago. It may seem tedious to have to document every subroutine you write, but it will save development time in the long run. In addition, should you wish to re-use a routine in another game at some point in the future, it will be very easy to rip out the required section and adapt it for your next project.

Other than that, just have fun. If you have any suggestions to make or errors to report, please get in touch.

Jonathan Cauldwell, January 2007.

http://www.spanglefish.com/egghead/

Contents

Chapter One - Simple Text and Graphics

Hello World

The first BASIC program that most novice programmers write is usually along these lines:

```
10 PRINT "Hello World"
20 GOTO 10
```

Alright, so the text may differ. Your first effort may have said "Dave is ace" or "Rob woz ere", but let's face it, displaying text and graphics on screen is probably the most important aspect of writing any computer game and - with the exception of pinball or fruit machines - it is practically impossible to conceive a game without a display. With this in mind let us begin this tutorial with some important display routines in the Spectrum ROM.

So how would we go about converting the above BASIC program to machine code? Well, we can PRINT by using the RST 16 instruction - effectively the same as PRINT CHR$ a - but that merely prints the character held in the accumulator to the current channel. To print a string on screen, we need to call two routines - one to open the upper screen for printing (channel 2), then the second to print the string. The routine at ROM address 5633 will open the channel number we pass in the accumulator, and 8252 will print a string beginning at de with length bc to this channel. Once channel 2 is opened, all printing is sent to the upper screen until we call 5633 with another value to send output elsewhere. Other interesting channels are 1 for the lower screen (like PRINT #1 in BASIC, and we can use this to display on the bottom two lines) and 3 for the ZX Printer.

```
       ld a,2              ; upper screen
       call 5633           ; open channel
loop   ld de,string        ; address of string
       ld bc,eostr-string  ; length of string to print
       call 8252           ; print our string
       jp loop             ; repeat until screen is full

string defb '(your name) is cool'
eostr  equ $
```

Running this listing fills the screen with the text until the scroll? prompt is displayed at the bottom. You will note however, that instead of each line of text appearing on a line of its own as in the BASIC listing, the beginning of each string follows directly on from the end of the previous one which is not exactly what we wanted. To acheive this we need to throw a line ourselves using an ASCII control code. One way of doing this would be to load the accumulator with the code for a new line (13), then use RST 16 to print this code. Another more efficient way is to add this ASCII code to the end of our string thus:

```
string defb '(your name) is cool'
       defb 13
eostr  equ $
```

There are a number of ASCII control codes like this which alter the current printing position, colours etc. and experimentation will help you to decide which ones you yourself will find most useful. Here are the main ones I use:

13	NEWLINE	sets print position to the beginning of the next line.
16,c	INK	Sets ink colour to the value of the following byte.
17,c	PAPER	Sets ink colour to the value of the following byte.
22,x,y	AT	Sets print x and y coordinates to the values specified in the following two bytes.

Code 22 is particularly handy for setting the coordinates at which a string or graphic character is to be displayed. This example will display an exclamation mark in the bottom right of the screen:

```
        ld a,2              ; upper screen
        call 5633           ; open channel
        ld de,string        ; address of string
        ld bc,eostr-string  ; length of string to print
        call 8252           ; print our string
        ret

string defb 22,21,31,'!'
eostr  equ $
```

This program goes one step further and animates an asterisk from the bottom to the top of the screen:

```
        ld a,2          ; 2 = upper screen.
        call 5633       ; open channel.
        ld a,21         ; row 21 = bottom of screen.
        ld (xcoord),a   ; set initial x coordinate.
loop    call setxy      ; set up our x/y coords.
        ld a,'*'        ; want an asterisk here.
        rst 16          ; display it.
        call delay      ; want a delay.
        call setxy      ; set up our x/y coords.
        ld a,32         ; ASCII code for space.
        rst 16          ; delete old asterisk.
        ld hl,xcoord    ; vertical position.
        dec (hl)        ; move it up one line.
        ld a,(hl)       ; where is it now?
        cp 255          ; past top of screen yet?
        jr nz,loop      ; no, carry on.
        ret
delay   ld b,10         ; length of delay.
delay0  halt            ; wait for an interrupt.
        djnz delay0     ; loop.
        ret             ; return.
setxy   ld a,22         ; ASCII control code for AT.
        rst 16          ; print it.
        ld a,(xcoord)   ; vertical position.
        rst 16          ; print it.
        ld a,(ycoord)   ; y coordinate.
        rst 16          ; print it.
        ret

xcoord defb 0
ycoord defb 15
```

Printing Simple Graphics

Moving asterisks around the screen is all very fine but for even the simplest game we really need to display graphics. Advanced graphics are discussed in later chapters, for now we will only be using simple Space Invader type graphics, and as any BASIC programmer will tell you, the Spectrum has a very simple mechanism for this - the User Defined Graphic, usually abbreviated to UDG.

The Spectrum's ASCII table contains 21 (19 in 128k mode) user-defined graphics characters, beginning at code 144 and going on up to 164 (162 in 128k mode). In BASIC UDGs are defined by poking data into the UDG area at the top of RAM, but in machine code it makes more sense to change the system variable which points to the memory location at which the UDGs are stored, which is done by changing the two-byte value at address 23675.

We can now modify our moving asterisk program to display a graphic instead with a few changes which are underlined.

```
        ld hl,udgs      ; UDGs.
        ld (23675),hl   ; set up UDG system variable.
        ld a,2          ; 2 = upper screen.
        call 5633       ; open channel.
        ld a,21         ; row 21 = bottom of screen.
        ld (xcoord),a   ; set initial x coordinate.
loop    call setxy      ; set up our x/y coords.
        ld a,144        ; show UDG instead of asterisk.
        rst 16          ; display it.
        call delay      ; want a delay.
        call setxy      ; set up our x/y coords.
        ld a,32         ; ASCII code for space.
        rst 16          ; delete old asterisk.
        call setxy      ; set up our x/y coords.
        ld hl,xcoord    ; vertical position.
        dec (hl)        ; move it up one line.
        ld a,(xcoord)   ; where is it now?
        cp 255          ; past top of screen yet?
        jr nz,loop      ; no, carry on.
        ret
delay   ld b,10         ; length of delay.
delay0  halt            ; wait for an interrupt.
        djnz delay0     ; loop.
        ret             ; return.
setxy   ld a,22         ; ASCII control code for AT.
        rst 16          ; print it.
        ld a,(xcoord)   ; vertical position.
        rst 16          ; print it.
        ld a,(ycoord)   ; y coordinate.
        rst 16          ; print it.
        ret
xcoord  defb 0
ycoord  defb 15
udgs    defb 60,126,219,153
        defb 255,255,219,219
```

Of course, there's no reason why you couldn't use more than the 21 UDGs if you wished. Simply set up a number of banks of them in memory and point to each one as you need it.

Alternatively, you could redefine the character set instead. This gives a larger range of ASCII characters from 32 (SPACE) to 127 (the copyright symbol). You could even mix text and graphics, redefining the letters and numbers of your font to the style of your choice, then using up the symbols and lowercase letters for aliens, zombies or whatever your game requires. To point to another set we subtract 256 from the address at which the font is placed and place this in the two byte system

variable at address 23606. The default Sinclair font for example is located at ROM address 15616, so the system variable at address 23606 points to 15360 when the Spectrum is first switched on.

This code copies the Sinclair ROM font to RAM making it "bolder" as it goes, then sets the system variable to point to it:

```
       ld hl,15616          ; ROM font.
       ld de,60000          ; address of our font.
       ld bc,768            ; 96 chars * 8 rows to alter.
font1  ld a,(hl)            ; get bitmap.
       rlca                 ; rotate it left.
       or (hl)              ; combine 2 images.
       ld (de),a            ; write to new font.
       inc hl               ; next byte of old.
       inc de               ; next byte of new.
       dec bc               ; decrement counter.
       ld a,b               ; high byte.
       or c                 ; combine with low byte.
       jr nz,font1          ; repeat until bc=zero.
       ld hl,60000-256      ; font minus 32*8.
       ld (23606),hl        ; point to new font.
       ret
```

Displaying Numbers

For most games it is better to define the player's score as a string of ASCII digits, although that does mean more work in the scoring routines and makes high score tables a real pain in the backside for an inexperienced assembly language programmer. We will cover this in a later chapter, but for now we'll use some handy ROM routines to print numbers for us.

There are two ways of printing a number on the screen, the first of which is to make use of the same routine that the ROM uses to print Sinclair BASIC line numbers. For this we simply load the bc register pair with the number we wish to print, then call 6683:

```
       ld bc,(score)
       call 6683
```

However, since BASIC line numbers can go only as high as 9999, this has the disadvantage of only being capable of displaying a four digit number. Once the player's score reaches 10000 other ASCII characters are displayed in place of numbers. Fortunately, there is another method which goes much higher. Instead of calling the line number display routine we can call the routine to place the contents of the bc registers on the calculator stack, then another routine which displays the number at the top of this stack. Don't worry about what the calculator stack is and what its function is because it's of little use to an arcade games programmer, but where we can make use of it we will. Just remember that the following three lines will display a number from 0 to 65535 inclusive:

```
       ld bc,(score)
       call 11563           ; stack number in bc.
       call 11747           ; display top of calc. stack.
```

Changing Colours

To set the permanent ink, paper, brightness and flash levels we can write directly to the system variable at 23693, then clear the screen with a call to the ROM:

```
; We want a yellow screen.
```

```
ld a,49             ; blue ink (1) on yellow paper (6*8).
ld (23693),a        ; set our screen colours.
call 3503           ; clear the screen.
```

The quickest and simplest way to set the border colour is to write to port 254. The 3 least significant bits of the byte we send determine the colour, so to set the border to red:

```
ld a,2              ; 2 is the code for red.
out (254),a         ; write to port 254.
```

Port 254 also drives the speaker and Mic socket in bits 3 and 4. However, the border effect will only last until your next call to the beeper sound routine in the ROM (more on that later), so a more permanent solution is required. To do this, we simply need to load the accumulator with the colour required and call the ROM routine at 8859. This will change the colour and set the BORDCR system variable (located at address 23624) accordingly. To set a permanent red border we can do this:

```
ld a,2              ; 2 is the code for red.
call 8859           ; set border colour.
```

Chapter Two - Keyboard and Joystick Control

One Key at a Time

Providing that you haven't disabled or otherwise meddled with the Spectrum's default interrupt mode the ROM will automatically read the keyboard and update several system variables located at memory location 23552 fifty times per second. The simplest way to check for a keypress is to first load address 23560 with a null value, then interrogate this location until it changes, the result being the ASCII value of the key pressed. This is most useful for those "press any key to continue" situations, for choosing items from a menu and for keyboard input such as high score name entry routines. Such a routine might look like this:

```
       ld hl,23560          ; LAST K system variable.
       ld (hl),0            ; put null value there.
loop   ld a,(hl)            ; new value of LAST K.
       cp 0                 ; is it still zero?
       jr z,loop            ; yes, so no key pressed.
       ret                  ; key was pressed.
```

Multiple Keypresses

Single keypresses are seldom any use for fast action arcade games however, for this we need to detect more than one simultaneous keypress and this is where things get a little trickier. Instead of reading memory addresses we have to read one of eight ports, each of which corresponds to a row of five keys. Of course, most Spectrum models appear to have far more keys than this so where did they all go? Well actually, they don't. The original Spectrum keyboard layout consisted of just forty keys, arranged in eight groupings or rows of five. In order to access some of the functions it was necessary to press certain combinations of keys together - for example to delete the combination required was CAPS SHIFT and 0 together. Sinclair added these extra keys when the Spectrum Plus came onto the scene in 1985, and they work by simulating the combinations of keypresses required for the original rubber keyed models.

The original keyboard layout was separated into these groupings:

```
Port  Keys

32766 B, N, M, Symbol Shift, Space
49150 H, J, K, L, Enter
57342 Y, U, I, O, P
61438 6, 7, 8, 9, 0
63486 5, 4, 3, 2, 1
64510 T, R, E, W, Q
65022 G, F, D, S, A
65278 V, C, X, Z, Caps Shift
```

To discover which keys are being pressed we read the appropriate port number, each key in the row being allocated one of the lower five bits d0-d4 (values 1,2,4,8 and 16) where d0 represents the outside key, d4 the innermost. Curiously, each bit is high where it is not pressed, low where it is - the opposite of what you might expect.

To read a row of five keys we simply load the port number into the bc register pair, then perform the instruction `in a,(c)`. As we only need the lowest value bits we can ignore the bits we dont want either with an `and 31` or by rotating the bits out of the accumulator into the carry flag using five `rra:call c,(address)` instructions.

If this is difficult to understand consider the following example:

```
        ld bc,63486         ; keyboard row 1-5/joystick port 2.
        in a,(c)            ; see what keys are pressed.
        rra                 ; outermost bit = key 1.
        push af             ; remember the value.
        call nc,mpl         ; it's being pressed, move left.
        pop af              ; restore accumulator.
        rra                 ; next bit along (value 2) = key 2.
        push af             ; remember the value.
        call nc,mpr         ; being pressed, so move right.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 4) = key 3.
        push af             ; remember the value.
        call nc,mpd         ; being pressed, so move down.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 8) reads key 4.
        call nc,mpu         ; it's being pressed, move up.
```

Joysticks

Sinclair joystick ports 1 and 2 were simply mapped to each of the rows of number keys and you can easily prove this by going into the BASIC editor and using the joystick to type numbers. Port 1 (Interface 2) was mapped to the keys 6,7,8,9 and 0, Port 2 (Interface 1) to keys 1,2,3,4 and 5. To detect joystick input we simply read the port in the same way as reading the keyboard. Sinclair joysticks use ports 63486 (Interface 1/port 2), and 61438 (Interface 2/port 1), bits d0-d4 will give a 0 for pressed, 1 for not pressed.

The popular Kempston joystick format is not mapped to the keyboard and can be read by using port 31 instead. This means we can use a simple `in a,(31)`. Again, bit values d0-d4 are used although this time the bit settings are as you might expect, with a bit set high if the joystick is being applied in a particular direction. The resulting bit values will be 1 for pressed, 0 for not pressed.

```
; Example joystick control routine.

joycon ld bc,31             ; Kempston joystick port.
       in a,(c)             ; read input.
       and 2                ; check "left" bit.
       call nz,joyl         ; move left.
       in a,(c)             ; read input.
       and 1                ; test "right" bit.
       call nz,joyr         ; move right.
       in a,(c)             ; read input.
       and 8                ; check "up" bit.
       call nz,joyu         ; move up.
       in a,(c)             ; read input.
       and 4                ; check "down" bit.
       call nz,joyd         ; move down.
       in a,(c)             ; read input.
       and 16               ; try the fire bit.
       call nz,fire         ; fire pressed.
```

A Simple Game

We can now go one step further and, putting into practice what we have already covered, write the main control section for a basic game. This will form the basis of a simple Centipede variant we will be developing over the next few chapters. We haven't covered everything needed for such a game yet but we can make a start with a small control loop which allows the player to manipulate a small gun base around the screen. Be warned, this program has no exit to BASIC so make sure you've saved a copy of your source code before running it.

```
; We want a black screen.

       ld a,71             ; white ink (7) on black paper (0),
                           ; bright (64).
       ld (23693),a        ; set our screen colours.
       xor a               ; quick way to load accumulator with zero.
       call 8859           ; set permanent border colours.

; Set up the graphics.

       ld hl,blocks        ; address of user-defined graphics data.
       ld (23675),hl       ; make UDGs point to it.

; Okay, let's start the game.

       call 3503           ; ROM routine - clears screen, opens chan 2.

; Initialise coordinates.

       ld hl,21+15*256     ; load hl pair with starting coords.
       ld (plx),hl         ; set player coords.

       call basexy         ; set the x and y positions of the player.
       call splayr         ; show player base symbol.

; This is the main loop.

mloop  equ $

; Delete the player.

       call basexy         ; set the x and y positions of the player.
       call wspace         ; display space over player.

; Now we've deleted the player we can move him before redisplaying him
; at his new coordinates.

       ld bc,63486         ; keyboard row 1-5/joystick port 2.
       in a,(c)            ; see what keys are pressed.
       rra                 ; outermost bit = key 1.
       push af             ; remember the value.
       call nc,mpl         ; it's being pressed, move left.
       pop af              ; restore accumulator.
       rra                 ; next bit along (value 2) = key 2.
       push af             ; remember the value.
       call nc,mpr         ; being pressed, so move right.
       pop af              ; restore accumulator.
       rra                 ; next bit (value 4) = key 3.
       push af             ; remember the value.
       call nc,mpd         ; being pressed, so move down.
       pop af              ; restore accumulator.
       rra                 ; next bit (value 8) reads key 4.
       call nc,mpu         ; it's being pressed, move up.


; Now he's moved we can redisplay the player.

       call basexy         ; set the x and y positions of the player.
       call splayr         ; show player.
```

```
       halt                ; delay.

; Jump back to beginning of main loop.

       jp mloop

; Move player left.

mpl    ld hl,ply           ; remember, y is the horizontal coord!
       ld a,(hl)           ; what's the current value?
       and a               ; is it zero?
       ret z               ; yes - we can't go any further left.
       dec (hl)            ; subtract 1 from y coordinate.
       ret

; Move player right.

mpr    ld hl,ply           ; remember, y is the horizontal coord!
       ld a,(hl)           ; what's the current value?
       cp 31               ; is it at the right edge (31)?
       ret z               ; yes - we can't go any further left.
       inc (hl)            ; add 1 to y coordinate.
       ret

; Move player up.

mpu    ld hl,plx           ; remember, x is the vertical coord!
       ld a,(hl)           ; what's the current value?
       cp 4                ; is it at upper limit (4)?
       ret z               ; yes - we can go no further then.
       dec (hl)            ; subtract 1 from x coordinate.
       ret

; Move player down.

mpd    ld hl,plx           ; remember, x is the vertical coord!
       ld a,(hl)           ; what's the current value?
       cp 21               ; is it already at the bottom (21)?
       ret z               ; yes - we can't go down any more.
       inc (hl)            ; add 1 to x coordinate.
       ret

; Set up the x and y coordinates for the player's gunbase position,
; this routine is called prior to display and deletion of gunbase.

basexy ld a,22             ; AT code.
       rst 16
       ld a,(plx)          ; player vertical coord.
       rst 16              ; set vertical position of player.
       ld a,(ply)          ; player's horizontal position.
       rst 16              ; set the horizontal coord.
       ret

; Show player at current print position.

splayr ld a,69             ; cyan ink (5) on black paper (0),
                           ; bright (64).
       ld (23695),a        ; set our temporary screen colours.
       ld a,144            ; ASCII code for User Defined Graphic 'A'.
       rst 16              ; draw player.
       ret

wspace ld a,71             ; white ink (7) on black paper (0),
                           ; bright (64).
       ld (23695),a        ; set our temporary screen colours.
       ld a,32             ; SPACE character.
       rst 16              ; display space.
       ret

plx    defb 0              ; player's x coordinate.
ply    defb 0              ; player's y coordinate.

; UDG graphics.
```

```
blocks defb 16,16,56,56,124,124,254,254    ; player base.
```

Fast, isn't it? In fact, we've slowed the loop down with a halt instruction but it still runs at a speedy 50 frames per second, which is probably a little too fast. Don't worry, as we add more features to the code it will begin to slow down. If you are feeling confident you might like to try adapting the above program to work with a Kempston joystick. It isn't difficult, and merely requires changing port 63486 to port 31, and replacing the four subsequent `call nc,(address)` to `call c,(address)` (The bits are reversed, remember?)

Redefineable keys are a little more tricky. As you are probably aware, the original Spectrum keyboard was divided into 8 rows of 5 keys each, and by reading the port associated with a particular row of keys, then testing bits d0-d4 we can tell if a particular key is being pressed. If you were to replace ld bc,31 in the code snippet above with ld bc,49150 you could test for the row of keys H to Enter - though that doesn't make for a convenient redefine keys routine. Thankfully, there is another way of going about it.

We can establish the port required for each row of keys using the formula in the Spectrum manual. Where n is the row number 0-7 the port address will be 254+256*(255-2^n). There's a ROM routine at address 654 which does a lot of the hard work for us by returning the number of the key pressed in the e register, in the range 0-39. 0-7 correspond to the innermost key of each row in turn (that's B, H, Y, 6, 5, T, G and V), 8-15 to the next key along in each row up to 39 for the outermost key on the last row - CAPS SHIFT. The shift key status, just for the record, is also returned in d. If no key is pressed then e returns 255.

The ROM routine can only return a single key number which is no good for detecting more than one keypress at a time. To determine whether or not a specific key is being pressed at any time we need to convert the number back into a port and bit, then read that port and check the individual bit for ourselves. There's a very handy routine I use for the job, and it's the only routine in my games which I didn't write myself. Credit for that must go to Stephen Jones, a programmer who used to write excellent articles for the Spectrum Discovery Club many years ago. To use his routine, load the accumulator with the number of the key you wish to test, call ktest, then check the carry flag. If it's set the key is not being pressed, if there's no carry then the key is being pressed. If that's too confusing and seems like the wrong way round, put a ccf instruction just before the ret.

```
; Mr. Jones' keyboard test routine.

ktest  ld c,a              ; key to test in c.
       and 7               ; mask bits d0-d2 for row.
       inc a               ; in range 1-8.
       ld b,a              ; place in b.
       srl c               ; divide c by 8,
       srl c               ; to find position within row.
       srl c
       ld a,5              ; only 5 keys per row.
       sub c               ; subtract position.
       ld c,a              ; put in c.
       ld a,254            ; high byte of port to read.
ktest0 rrca                ; rotate into position.
       djnz ktest0         ; repeat until we've found relevant row.
       in a,(254)          ; read port (a=high, 254=low).
ktest1 rra                 ; rotate bit out of result.
       dec c               ; loop counter.
       jp nz,ktest1        ; repeat until bit for position in carry.
       ret
```

Chapter Three - Loudspeaker Sound Effects

The Loudspeaker

There are two ways of generating sound and music on the ZX Spectrum, the best and most complicated of which is via the AY38912 sound chip in the 128K models. This method is described in detail in a later chapter, but for now we will concern ourselves with the 48K loudspeaker. Simple it may be, but this method does have its uses especially for short sharp sound effects during games.

Beep

First of all we need to know how to produce a beep of a certain pitch and duration, and the Sinclair ROM has a fairly accessible routine to do the job for us at address 949, all that is required is to pass the parameters for pitch in the HL register pair and duration in DE, call 949 and we get an appropriate "beep".

Alas, the way in which we work out the parameters required is a little tricky as it needs a little calculation. We need to know the Hertz value for the frequency of note to emit, essentially just the number of times the loudspeaker needs to be toggled each second to produce the desired pitch. A suitable table is located below:

Middle C	261.63
C sharp	277.18
D	293.66
D sharp	311.13
E	329.63
F	349.23
F sharp	369.99
G	392.00
G sharp	415.30
A	440.00
A sharp	466.16
B	493.88

For each octave higher, simply double the frequency, to go an octave lower halve it. For example, to produce a note C one octave higher than middle C we take the value for Middle C - 261.63, and double it to 523.26.

Once the frequency is established we multiply it by the number of seconds required and pass this to the ROM routine in the DE register pair as the duration - so to play the note at middle C for one tenth of a second the duration required would be 261.63 * 0.1 = 26. The pitch is worked out by first dividing the 437500 by the frequency, subtracting 30.125 and passing the result in the HL registers. For middle C this would mean a value of 437500 / 261.63 - 30.125 = 1642.

In other words:

DE = Duration = Frequency * Seconds
HL = Pitch = 437500 / Frequency - 30.125

So to play note G sharp one octave above that of middle C for one quarter of one second:

```
; Frequency of G sharp in octave of middle C = 415.30
; Frequency of G sharp one octave higher = 830.60
; Duration = 830.6 / 4 = 207.65
; Pitch = 437500 / 830.6 - 30.125 = 496.6

       ld hl,497          ; pitch.
       ld de,208          ; duration.
       call 949           ; ROM beeper routine.
       ret
```

Of course, this routine isn't just useful for musical notes - we can use it for a variety of effects as well, one of my favourites being a simple pitch bend routine:

```
       ld hl,500          ; starting pitch.
       ld b,250           ; length of pitch bend.
loop   push bc
       push hl            ; store pitch.
       ld de,1            ; very short duration.
       call 949           ; ROM beeper routine.
       pop hl             ; restore pitch.
       inc hl             ; pitch going up.
       pop bc
       djnz loop          ; repeat.
       ret
```

Have a play with the above routine - by fiddling with it it's pretty easy to adjust the pitch up and down, and to change the starting frequency and pitch bend and length producing a number of interesting effects. One word of warning though - Don't go too crazy with your pitch or duration values or the beeper routine will get stuck and you won't be able to regain control of your Spectrum without resetting it.

White Noise

When using the loudspeaker we don't even have to stick with the routines in the ROM, it is easy enough to write our own sound effects routines, especially if we want to generate white noise for crashes and bangs. White noise is usually a lot more fun to play with.

To generate white noise all we need is a quick and simple random number generator (a Fibonacci sequence might work, but I'd recommend stepping a pointer through the first 8K of ROM and fetching the byte at each location to get a reasonably random 8-bit number). Then write this value to port 254. Remember this port also controls the border colour so if you don't want a striped multicolour border effect we need to mask off the border bits with AND 248 and add the number for the border colour we want (1 for blue, 2 for red etc.) before performing an OUT (254) instruction. When we've done this we need to put in a small delay loop (short for high pitch, long for lower pitch) and repeat the process a few hundred times. This will give us a nice "crash" effect.

This routine is based on a sound effect from Egghead 3:

```
noise  ld e,250           ; repeat 250 times.
       ld hl,0            ; start pointer in ROM.
noise2 push de
       ld b,32            ; length of step.
noise0 push bc
       ld a,(hl)          ; next "random" number.
       inc hl             ; pointer.
       and 248            ; we want a black border.
       out (254),a        ; write to speaker.
```

```
        ld a,e              ; as e gets smaller...
        cpl                 ; ...we increase the delay.
noise1  dec a               ; decrement loop counter.
        jr nz,noise1        ; delay loop.
        pop bc
        djnz noise0         ; next step.
        pop de
        ld a,e
        sub 24              ; size of step.
        cp 30               ; end of range.
        ret z
        ret c
        ld e,a
        cpl
noise3  ld b,40             ; silent period.
noise4  djnz noise4
        dec a
        jr nz,noise3
        jr noise2
```

Chapter Four - Random Numbers

Generating random numbers in machine code can be a tricky problem for a novice programmer.

First of all, let's get one thing straight. There is no such thing as a random number generator. The CPU merely follows instructions and has no mind of its own, it cannot simply pluck a number out of thin air based on a whim. Instead, it needs to follow a formula which will produce an unpredictable sequence of numbers which do not *appear* to follow any sort of pattern, and therefore give the impression of randomness. All we can do is return a false - or *pseudo* - random number.

One method of obtaining a pseudo-random number would be to use the Fibonacci sequence, however the easiest and quickest method of generating a pseudo-random 8-bit number on the Spectrum is by stepping a pointer through the ROM, and examining the contents of the byte at each location in turn. There is one small drawback to this method - the Sinclair ROM contains a very uniform and non-random area towards the end which is best avoided. By limiting the pointer to, say, the first 8K of ROM we still have a sequence of 8192 "random" numbers, more than enough for most games. In fact, every game I have ever written with a random number generator uses this method, or a very similar one:

```
; Simple pseudo-random number generator.
; Steps a pointer through the ROM (held in seed), returning
; the contents of the byte at that location.

random ld hl,(seed)        ; Pointer
       ld a,h
       and 31              ; keep it within first 8k of ROM.
       ld h,a
       ld a,(hl)           ; Get "random" number from location.
       inc hl              ; Increment pointer.
       ld (seed),hl
       ret
seed   defw 0
```

Let's put our new random number generator to use in our Centipede game. Every Centipede game needs mushrooms - lots of them - scattered randomly across the play area, and we can now call the random routine to supply coordinates for each mushroom as we display them. The bits underlined are those we need to add.

```
; We want a black screen.

       ld a,71             ; white ink (7) on black paper (0),
                           ; bright (64).
       ld (23693),a        ; set our screen colours.
       xor a               ; quick way to load accumulator with zero.
       call 8859           ; set permanent border colours.

; Set up the graphics.

       ld hl,blocks        ; address of user-defined graphics data.
       ld (23675),hl       ; make UDGs point to it.

; Okay, let's start the game.

       call 3503           ; ROM routine - clears screen, opens chan 2.

; Initialise coordinates.

       ld hl,21+15*256     ; load hl pair with starting coords.
       ld (plx),hl         ; set player coords.
```

```
        call basexy         ; set the x and y positions of the player.
        call splayr         ; show player base symbol.

; Now we want to fill the play area with mushrooms.

        ld a,68             ; green ink (4) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary colours.
        ld b,50             ; start with a few.
mushlp ld a,22              ; control code for AT character.
        rst 16
        call random         ; get a 'random' number.
        and 15              ; want vertical in range 0 to 15.
        rst 16
        call random         ; want another pseudo-random number.
        and 31              ; want horizontal in range 0 to 31.
        rst 16
        ld a,145            ; UDG 'B' is the mushroom graphic.
        rst 16              ; put mushroom on screen.
        djnz mushlp         ; loop back until all mushrooms displayed.


; This is the main loop.

mloop  equ $

; Delete the player.

        call basexy         ; set the x and y positions of the player.
        call wspace         ; display space over player.

; Now we've deleted the player we can move him before redisplaying him
; at his new coordinates.

        ld bc,63486         ; keyboard row 1-5/joystick port 2.
        in a,(c)            ; see what keys are pressed.
        rra                 ; outermost bit = key 1.
        push af             ; remember the value.
        call nc,mpl         ; it's being pressed, move left.
        pop af              ; restore accumulator.
        rra                 ; next bit along (value 2) = key 2.
        push af             ; remember the value.
        call nc,mpr         ; being pressed, so move right.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 4) = key 3.
        push af             ; remember the value.
        call nc,mpd         ; being pressed, so move down.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 8) reads key 4.
        call nc,mpu         ; it's being pressed, move up.


; Now he's moved we can redisplay the player.

        call basexy         ; set the x and y positions of the player.
        call splayr         ; show player.

        halt                ; delay.

; Jump back to beginning of main loop.

        jp mloop

; Move player left.

mpl     ld hl,ply           ; remember, y is the horizontal coord!
        ld a,(hl)           ; what's the current value?
        and a               ; is it zero?
        ret z               ; yes - we can't go any further left.
        dec (hl)            ; subtract 1 from y coordinate.
        ret
```

```
; Move player right.

mpr     ld hl,ply           ; remember, y is the horizontal coord!
        ld a,(hl)           ; what's the current value?
        cp 31               ; is it at the right edge (31)?
        ret z               ; yes - we can't go any further left.
        inc (hl)            ; add 1 to y coordinate.
        ret

; Move player up.

mpu     ld hl,plx           ; remember, x is the vertical coord!
        ld a,(hl)           ; what's the current value?
        cp 4                ; is it at upper limit (4)?
        ret z               ; yes - we can go no further then.
        dec (hl)            ; subtract 1 from x coordinate.
        ret

; Move player down.

mpd     ld hl,plx           ; remember, x is the vertical coord!
        ld a,(hl)           ; what's the current value?
        cp 21               ; is it already at the bottom (21)?
        ret z               ; yes - we can't go down any more.
        inc (hl)            ; add 1 to x coordinate.
        ret

; Set up the x and y coordinates for the player's gunbase position,
; this routine is called prior to display and deletion of gunbase.

basexy ld a,22              ; AT code.
        rst 16
        ld a,(plx)          ; player vertical coord.
        rst 16              ; set vertical position of player.
        ld a,(ply)          ; player's horizontal position.
        rst 16              ; set the horizontal coord.
        ret

; Show player at current print position.

splayr ld a,69              ; cyan ink (5) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary screen colours.
        ld a,144            ; ASCII code for User Defined Graphic 'A'.
        rst 16              ; draw player.
        ret

wspace ld a,71              ; white ink (7) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary screen colours.
        ld a,32             ; SPACE character.
        rst 16              ; display space.
        ret

; Simple pseudo-random number generator.
; Steps a pointer through the ROM (held in seed), returning
; the contents of the byte at that location.

random ld hl,(seed)         ; Pointer
        ld a,h
        and 31              ; keep it within first 8k of ROM.
        ld h,a
        ld a,(hl)           ; Get "random" number from location.
        inc hl              ; Increment pointer.
        ld (seed),hl
        ret
seed    defw 0

plx     defb 0              ; player's x coordinate.
ply     defb 0              ; player's y coordinate.
```

```
; UDG graphics.

blocks defb 16,16,56,56,124,124,254,254    ; player base.
       defb 24,126,255,255,60,60,60,60     ; mushroom.
```

Once run this listing looks more like a Centipede game than it did before, but there's a major problem. The mushrooms are distributed in a random fashion around the screen, but the player can move straight through them. Some form of collision detection is required to prevent this happening, and we shall cover this in the next chapter.

Chapter Five - Simple Background Collision Detection

Finding Attributes

Anyone who ever spent time programming in Sinclair BASIC may well remember the ATTR function. This was a way to detect the colour attributes of any particular character cell on the screen, and though tricky for the BASIC programmer to grasp, could be very handy for simple collision detection. The method was so useful in fact that it its machine language equivalent was employed by a number of commercial games, and it is of great use to the novice Spectrum programmer.

There are two ways to find the colour attribute settings for a particular character cell on the Spectrum. A quick look through the Spectrum's ROM disassembly reveals a routine at address 9603 which will do the job for us, or we can calculate the memory address ourselves.

The simplest way to find an attribute value is to use a couple of ROM routines:

```
       ld bc,(ballx)       ; put x and y in bc register pair.
       call 9603           ; call ROM to put attribute (c,b) on stack.
       call 11733          ; put attributes in accumulator.
```

However, it is much faster to do the calculation ourselves. It is also useful to calculate an attribute's *address*, and not just its value, in case we want to write to it as well.

Calculating Attribute Addresses

Unlike the Spectrum's awkward pixel layout, colour cells, located at addresses 22528 to 23295 inclusive, are arranged sequentially in RAM as one would expect. In other words, the screen's top 32 attribute cells are located at addresses 22528 to 22559 going left to right, the second row of colour cells from 22560 to 22591 and so on. To find the address of a colour cell at print position (x,y) we therefore need only to multiply x by 32, add y, then add 22528 to the result. By then examining the contents of this address we can find out the colours displayed at a particular position, and act accordingly. The following example calculates the address of an attribute at character position (b,c) and returns it in the hl register pair.

```
; Calculate address of attribute for character at (b, c).

atadd  ld a,b              ; x position.
       rrca                ; multiply by 32.
       rrca
       rrca
       ld l,a              ; store away in l.
       and 3               ; mask bits for high byte.
       add a,88            ; 88*256=22528, start of attributes.
       ld h,a              ; high byte done.
       ld a,l              ; get x*32 again.
       and 224             ; mask low byte.
       ld l,a              ; put in l.
       ld a,c              ; get y displacement.
       add a,l             ; add to low byte.
       ld l,a              ; hl=address of attributes.
       ld a,(hl)           ; return attribute in a.
       ret
```

Interrogating the contents of the byte at hl will give the attribute's value, while writing to the memory location at hl will change the colour of the square.

To make sense of the result we have to know that each attribute is made up of 8 bits which are arranged in this manner:

d0-d2	ink colour 0-7,	0=black, 1=blue, 2=red, 3=magenta, 4=green, 5=cyan, 6=yellow, 7=white
d3-d5	paper colour 0-7,	0=black, 1=blue, 2=red, 3=magenta, 4=green, 5=cyan, 6=yellow, 7=white
d6	bright,	0=dull, 1=bright
d7	flash,	0=stable, 1=flashing

The test for green paper for example, might involve

```
       and 56              ; mask away all but paper bits.
       cp 32               ; is it green(4) * 8?
       jr z,green          ; yes, do green thing.
```

while checking for yellow ink could be done like this

```
       and 7               ; only want bits pertaining to ink.
       cp 6                ; is it yellow (6)?
       jr z,yellow         ; yes, do yellow wotsit.
```

Applying what we Have Learned to the Game

We can now add an attribute collision check to our Centipede game. As before, the new sections are underlined.

```
; We want a black screen.

       ld a,71             ; white ink (7) on black paper (0),
                           ; bright (64).
       ld (23693),a        ; set our screen colours.
       xor a               ; quick way to load accumulator with zero.
       call 8859           ; set permanent border colours.

; Set up the graphics.

       ld hl,blocks        ; address of user-defined graphics data.
       ld (23675),hl       ; make UDGs point to it.

; Okay, let's start the game.

       call 3503           ; ROM routine - clears screen, opens chan 2.

; Initialise coordinates.

       ld hl,21+15*256     ; load hl pair with starting coords.
       ld (plx),hl         ; set player coords.

       call basexy         ; set the x and y positions of the player.
       call splayr         ; show player base symbol.

; Now we want to fill the play area with mushrooms.

       ld a,68             ; green ink (4) on black paper (0),
                           ; bright (64).
       ld (23695),a        ; set our temporary colours.
       ld b,50             ; start with a few.
mushlp ld a,22             ; control code for AT character.
       rst 16
       call random         ; get a 'random' number.
```

```
        and 15              ; want vertical in range 0 to 15.
        rst 16
        call random         ; want another pseudo-random number.
        and 31              ; want horizontal in range 0 to 31.
        rst 16
        ld a,145            ; UDG 'B' is the mushroom graphic.
        rst 16              ; put mushroom on screen.
        djnz mushlp         ; loop back until all mushrooms displayed.


; This is the main loop.

mloop   equ $

; Delete the player.

        call basexy         ; set the x and y positions of the player.
        call wspace         ; display space over player.

; Now we've deleted the player we can move him before redisplaying him
; at his new coordinates.

        ld bc,63486         ; keyboard row 1-5/joystick port 2.
        in a,(c)            ; see what keys are pressed.
        rra                 ; outermost bit = key 1.
        push af             ; remember the value.
        call nc,mpl         ; it's being pressed, move left.
        pop af              ; restore accumulator.
        rra                 ; next bit along (value 2) = key 2.
        push af             ; remember the value.
        call nc,mpr         ; being pressed, so move right.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 4) = key 3.
        push af             ; remember the value.
        call nc,mpd         ; being pressed, so move down.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 8) reads key 4.
        call nc,mpu         ; it's being pressed, move up.


; Now he's moved we can redisplay the player.

        call basexy         ; set the x and y positions of the player.
        call splayr         ; show player.

        halt                ; delay.

; Jump back to beginning of main loop.

        jp mloop

; Move player left.

mpl     ld hl,ply           ; remember, y is the horizontal coord!
        ld a,(hl)           ; what's the current value?
        and a               ; is it zero?
        ret z               ; yes - we can't go any further left.

; now check that there isn't a mushroom in the way.

        ld bc,(plx)         ; current coords.
        dec b               ; look 1 square to the left.
        call atadd          ; get address of attribute at this position.
        cp 68               ; mushrooms are bright (64) + green (4).
        ret z               ; there's a mushroom - we can't move there.

        dec (hl)            ; subtract 1 from y coordinate.
        ret

; Move player right.

mpr     ld hl,ply           ; remember, y is the horizontal coord!
        ld a,(hl)           ; what's the current value?
```

```
       cp 31               ; is it at the right edge (31)?
       ret z               ; yes - we can't go any further left.

; now check that there isn't a mushroom in the way.

       ld bc,(plx)         ; current coords.
       inc b               ; look 1 square to the right.
       call atadd          ; get address of attribute at this position.
       cp 68               ; mushrooms are bright (64) + green (4).
       ret z               ; there's a mushroom - we can't move there.

       inc (hl)            ; add 1 to y coordinate.
       ret

; Move player up.

mpu    ld hl,plx           ; remember, x is the vertical coord!
       ld a,(hl)           ; what's the current value?
       cp 4                ; is it at upper limit (4)?
       ret z               ; yes - we can go no further then.

; now check that there isn't a mushroom in the way.

       ld bc,(plx)         ; current coords.
       dec c               ; look 1 square up.
       call atadd          ; get address of attribute at this position.
       cp 68               ; mushrooms are bright (64) + green (4).
       ret z               ; there's a mushroom - we can't move there.

       dec (hl)            ; subtract 1 from x coordinate.
       ret

; Move player down.

mpd    ld hl,plx           ; remember, x is the vertical coord!
       ld a,(hl)           ; what's the current value?
       cp 21               ; is it already at the bottom (21)?
       ret z               ; yes - we can't go down any more.

; now check that there isn't a mushroom in the way.

       ld bc,(plx)         ; current coords.
       inc c               ; look 1 square down.
       call atadd          ; get address of attribute at this position.
       cp 68               ; mushrooms are bright (64) + green (4).
       ret z               ; there's a mushroom - we can't move there.

       inc (hl)            ; add 1 to x coordinate.
       ret

; Set up the x and y coordinates for the player's gunbase position,
; this routine is called prior to display and deletion of gunbase.

basexy ld a,22             ; AT code.
       rst 16
       ld a,(plx)          ; player vertical coord.
       rst 16              ; set vertical position of player.
       ld a,(ply)          ; player's horizontal position.
       rst 16              ; set the horizontal coord.
       ret

; Show player at current print position.

splayr ld a,69             ; cyan ink (5) on black paper (0),
                           ; bright (64).
       ld (23695),a        ; set our temporary screen colours.
       ld a,144            ; ASCII code for User Defined Graphic 'A'.
       rst 16              ; draw player.
       ret

wspace ld a,71             ; white ink (7) on black paper (0),
                           ; bright (64).
       ld (23695),a        ; set our temporary screen colours.
```

```
       ld a,32             ; SPACE character.
       rst 16              ; display space.
       ret

; Simple pseudo-random number generator.
; Steps a pointer through the ROM (held in seed), returning
; the contents of the byte at that location.

random ld hl,(seed)        ; Pointer
       ld a,h
       and 31              ; keep it within first 8k of ROM.
       ld h,a
       ld a,(hl)           ; Get "random" number from location.
       inc hl              ; Increment pointer.
       ld (seed),hl
       ret
seed   defw 0


; Calculate address of attribute for character at (dispx, dispy).

atadd  ld a,c              ; vertical coordinate.
       rrca                ; multiply by 32.
       rrca                ; Shifting right with carry 3 times is
       rrca                ; quicker than shifting left 5 times.
       ld e,a
       and 3
       add a,88            ; 88x256=address of attributes.
       ld d,a
       ld a,e
       and 224
       ld e,a
       ld a,b              ; horizontal position.
       add a,e
       ld e,a              ; de=address of attributes.
       ld a,(de)           ; return with attribute in accumulator.
       ret

plx    defb 0              ; player's x coordinate.
ply    defb 0              ; player's y coordinate.

; UDG graphics.

blocks defb 16,16,56,56,124,124,254,254    ; player base.
       defb 24,126,255,255,60,60,60,60     ; mushroom.
```

Chapter Six - Tables

Aliens Don't Come One at a Time

Let us say, for the sake of example, we were writing a Space Invaders game featuring eleven columns, each containing five rows of invaders. It would be impractical to write the code for each of the fifty-five aliens in turn, so we need to set up a table. In Sinclair BASIC we might go about this by defining three arrays of fifty-five elements - one for the invaders' x coordinates, one for y coordinates, plus a third status byte. We could do something similar in assembler by setting up three tables of fifty-five bytes each in memory, then adding the number for each alien to the start of each table to access the individual element. Unfortunately, that would be slow and cumbersome.

A far better method is to group the three data elements for each invader into a structure, and then have fifty-five of these structures in a table. We can then point hl to the address of each invader, and know that hl points to the status byte, hl plus one points to the x coordinate, and hl plus two points to the y coordinate. The code to display an alien might look something like this

```
        ld hl,aliens        ; alien data structures.
        ld b,55             ; number of aliens.
loop0   call show           ; show this alien.
        djnz loop0          ; repeat for all aliens.
        ret
show    ld a,(hl)           ; fetch alien status.
        cp 255              ; is alien switched off?
        jr z,next           ; yes, so don't display him.
        push hl             ; store alien address on the stack.
        inc hl              ; point to x coord.
        ld d,(hl)           ; get coord.
        inc hl              ; point to y coord.
        ld e,(hl)           ; get coord.
        call disply         ; display alien at (d,e).
        pop hl              ; retrieve alien address from the stack.
next    ld de,3             ; size of each alien table entry.
        add hl,de           ; point to next alien.
        ret                 ; leave hl pointing to next one.
```

Using the Index Registers

The drawback with this routine is that we have to be very careful where hl is pointing to all the time, so it might be an idea to store hl in a two-byte temporary memory location before calling `show`, then restoring it afterwards, adding three at the end of the main loop, then performing the `djnz` instruction. If we were writing for the Nintendo GameBoy with its cut-down Z80 this would probably represent our best option. On machines with more advanced processors such as the Spectrum and CPC464 we can use the index registers, ix, to simplify our code a little. Because the ix register pair allows us to displace our indirect addressing, we can point ix to the beginning of an alien's data structure and access all elements within it without the need to change ix again. Using ix our alien display routine might look like this

```
        ld ix,aliens        ; alien data structures.
        ld b,55             ; number of aliens.
loop0   call show           ; show this alien.
        ld de,3             ; size of each alien table entry.
        add ix,de           ; point to next alien.
        djnz loop0          ; repeat for all aliens.
        ret
show    ld a,(ix)           ; fetch alien status.
        cp 255              ; is alien switched off?
        ret z               ; yes, so don't display him.
        ld d,(ix+1)         ; get coord.
        ld e,(ix+2)         ; get coord.
```

```
       jp disply           ; display alien at (d,e).
```

Using ix means we only ever need to point to the *beginning* of an alien's data structure, so ix will always return the status for the current invader, ix+1 the x coordinate, and so on. This method enables the programmer to use complex data structures for his aliens of up to 128 bytes long, without getting confused as to which bit of the structure our registers are pointing at any given time as with the hl example earlier. Unfortunately, using ix is a little slower than hl, so we shouldn't use it for the more intensive processing tasks such as manipulating graphics.

Let us apply this method to our Centipede game. Firstly, we need to decide how many segments are needed, and what data to store about each segment. In our game the segments will need to move left or right until they hit a mushroom, then move down and go back the other way. So it seems we will need a flag to indicate the particular direction a segment is travelling in, plus an x or y coordinate. Our flag can also be used to indicate that a particular segment has been destroyed. With this in mind we can set up a data structure of three bytes:

```
centf  defb 0              ; flag, 0=left, 1=right, 255=dead.
centx  defb 0              ; segment x coordinate.
centy  defb 0              ; segment y coordinate.
```

If we choose to have ten segments in our centipede, we need to reserve a table space of thirty bytes. Each segment needs to be initialised at the beginning, then deleted, moved and redisplayed during the game.

Initialising our segments is probably the simplest task, so we can use a simple loop incrementing the hl register pair for each byte before setting it. Something like this will usually do the trick:

```
       ld b,10             ; number of segments to initialise.
       ld hl,segmnt        ; segment table.
segint ld (hl),1           ; start off moving right.
       inc hl
       ld (hl),0           ; start at top.
       inc hl
       ld (hl),b           ; use B register as y coordinate.
       inc hl
       djnz segint         ; repeat until all initialised.
```

Processing and displaying each segment is going to be slightly more complicated, so for that we will use the ix registers. We need to write a simple algorithm which manipulates a single segment left or right until it hits a mushroom, then moves down and switches direction. We'll call this routine `proseg` (for "process segment"), and set up a loop which points to each segment in turn and calls `proseg`. Providing we get the movement algorithm correct we should then see a centipede snaking its way through the mushrooms. Applying this to our code is straightforward - we check the flag byte for each segment (ix) to see which way the segment is moving, increment or decrement the horizontal coordinate (ix+2) accordingly, then check the attribute at that character cell. If it's green and black we increment the vertical coordinate (ix+1), and switch the direction flag (ix).

Okay, there are one or two other things to consider, such as hitting the sides or bottom of the screen, but that's just a case of checking the segment's coordinates and switching direction or moving to the top of the screen when we need to. The segments also need to be deleted from their old positions prior to being moved, the redisplayed at their new positions, but we have already covered the steps required to perform those tasks.

Our new code looks like this:

```
; We want a black screen.

        ld a,71             ; white ink (7) on black paper (0),
                            ; bright (64).
        ld (23693),a        ; set our screen colours.
        xor a               ; quick way to load accumulator with zero.
        call 8859           ; set permanent border colours.

; Set up the graphics.

        ld hl,blocks        ; address of user-defined graphics data.
        ld (23675),hl       ; make UDGs point to it.

; Okay, let's start the game.

        call 3503           ; ROM routine - clears screen, opens chan 2.

; Initialise coordinates.

        ld hl,21+15*256     ; load hl pair with starting coords.
        ld (plx),hl         ; set player coords.

        ld b,10             ; number of segments to initialise.
        ld hl,segmnt        ; segment table.
segint  ld (hl),1           ; start off moving right.
        inc hl
        ld (hl),0           ; start at top.
        inc hl
        ld (hl),b           ; use B register as y coordinate.
        inc hl
        djnz segint         ; repeat until all initialised.

        call basexy         ; set the x and y positions of the player.
        call splayr         ; show player base symbol.

; Now we want to fill the play area with mushrooms.

        ld a,68             ; green ink (4) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary colours.
        ld b,50             ; start with a few.
mushlp  ld a,22             ; control code for AT character.
        rst 16
        call random         ; get a 'random' number.
        and 15              ; want vertical in range 0 to 15.
        rst 16
        call random         ; want another pseudo-random number.
        and 31              ; want horizontal in range 0 to 31.
        rst 16
        ld a,145            ; UDG 'B' is the mushroom graphic.
        rst 16              ; put mushroom on screen.
        djnz mushlp         ; loop back until all mushrooms displayed.


; This is the main loop.

mloop   equ $

; Delete the player.

        call basexy         ; set the x and y positions of the player.
        call wspace         ; display space over player.

; Now we've deleted the player we can move him before redisplaying him
; at his new coordinates.

        ld bc,63486         ; keyboard row 1-5/joystick port 2.
        in a,(c)            ; see what keys are pressed.
        rra                 ; outermost bit = key 1.
        push af             ; remember the value.
        call nc,mpl         ; it's being pressed, move left.
        pop af              ; restore accumulator.
        rra                 ; next bit along (value 2) = key 2.
```

```
        push af             ; remember the value.
        call nc,mpr         ; being pressed, so move right.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 4) = key 3.
        push af             ; remember the value.
        call nc,mpd         ; being pressed, so move down.
        pop af              ; restore accumulator.
        rra                 ; next bit (value 8) reads key 4.
        call nc,mpu         ; it's being pressed, move up.


; Now he's moved we can redisplay the player.

        call basexy         ; set the x and y positions of the player.
        call splayr         ; show player.

; Now for the centipede segments.

        ld ix,segmnt        ; table of segment data.
        ld b,10             ; number of segments in table.
censeg push bc
        ld a,(ix)           ; is segment switched on?
        inc a               ; 255=off, increments to zero.
        call nz,proseg      ; it's active, process segment.
        pop bc
        ld de,3             ; 3 bytes per segment.
        add ix,de           ; get next segment in ix registers.
        djnz censeg         ; repeat for all segments.

        halt                ; delay.

; Jump back to beginning of main loop.

        jp mloop

; Move player left.

mpl     ld hl,ply           ; remember, y is the horizontal coord!
        ld a,(hl)           ; what's the current value?
        and a               ; is it zero?
        ret z               ; yes - we can't go any further left.

; now check that there isn't a mushroom in the way.

        ld bc,(plx)         ; current coords.
        dec b               ; look 1 square to the left.
        call atadd          ; get address of attribute at this position.
        cp 68               ; mushrooms are bright (64) + green (4).
        ret z               ; there's a mushroom - we can't move there.

        dec (hl)            ; subtract 1 from y coordinate.
        ret

; Move player right.

mpr     ld hl,ply           ; remember, y is the horizontal coord!
        ld a,(hl)           ; what's the current value?
        cp 31               ; is it at the right edge (31)?
        ret z               ; yes - we can't go any further left.

; now check that there isn't a mushroom in the way.

        ld bc,(plx)         ; current coords.
        inc b               ; look 1 square to the right.
        call atadd          ; get address of attribute at this position.
        cp 68               ; mushrooms are bright (64) + green (4).
        ret z               ; there's a mushroom - we can't move there.

        inc (hl)            ; add 1 to y coordinate.
        ret

; Move player up.
```

```
mpu     ld hl,plx           ; remember, x is the vertical coord!
        ld a,(hl)           ; what's the current value?
        cp 4                ; is it at upper limit (4)?
        ret z               ; yes - we can go no further then.

; now check that there isn't a mushroom in the way.

        ld bc,(plx)         ; current coords.
        dec c               ; look 1 square up.
        call atadd          ; get address of attribute at this position.
        cp 68               ; mushrooms are bright (64) + green (4).
        ret z               ; there's a mushroom - we can't move there.

        dec (hl)            ; subtract 1 from x coordinate.
        ret

; Move player down.

mpd     ld hl,plx           ; remember, x is the vertical coord!
        ld a,(hl)           ; what's the current value?
        cp 21               ; is it already at the bottom (21)?
        ret z               ; yes - we can't go down any more.

; now check that there isn't a mushroom in the way.

        ld bc,(plx)         ; current coords.
        inc c               ; look 1 square down.
        call atadd          ; get address of attribute at this position.
        cp 68               ; mushrooms are bright (64) + green (4).
        ret z               ; there's a mushroom - we can't move there.

        inc (hl)            ; add 1 to x coordinate.
        ret

; Set up the x and y coordinates for the player's gunbase position,
; this routine is called prior to display and deletion of gunbase.

basexy  ld a,22             ; AT code.
        rst 16
        ld a,(plx)          ; player vertical coord.
        rst 16              ; set vertical position of player.
        ld a,(ply)          ; player's horizontal position.
        rst 16              ; set the horizontal coord.
        ret

; Show player at current print position.

splayr  ld a,69             ; cyan ink (5) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary screen colours.
        ld a,144            ; ASCII code for User Defined Graphic 'A'.
        rst 16              ; draw player.
        ret

wspace  ld a,71             ; white ink (7) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary screen colours.
        ld a,32             ; SPACE character.
        rst 16              ; display space.
        ret

segxy   ld a,22             ; ASCII code for AT character.
        rst 16              ; display AT code.
        ld a,(ix+1)         ; get segment x coordinate.
        rst 16              ; display coordinate code.
        ld a,(ix+2)         ; get segment y coordinate.
        rst 16              ; display coordinate code.
        ret

proseg  ld a,(ix)           ; check if segment was switched off
        inc a               ; by collision detection routine.
        ret z               ; it was, so this segment is now dead.
        call segxy          ; set up segment coordinates.
```

```
       call wspace          ; display a space, white ink on black.
       call segmov          ; move segment.
       ld a,(ix)            ; check if segment was switched off
       inc a                ; by collision detection routine.
       ret z                ; it was, so this segment is now dead.
       call segxy           ; set up segment coordinates.
       ld a,2               ; attribute code 2 = red segment.
       ld (23695),a         ; set temporary attributes.
       ld a,146             ; UDG 'C' to display segment.
       rst 16
       ret
segmov ld a,(ix+1)          ; x coord.
       ld c,a               ; GP x area.
       ld a,(ix+2)          ; y coord.
       ld b,a               ; GP y area.
       ld a,(ix)            ; status flag.
       and a                ; is the segment heading left?
       jr z,segml           ; going left - jump to that bit of code.

; so segment is going right then!

segmr  ld a,(ix+2)          ; y coord.
       cp 31                ; already at right edge of screen?
       jr z,segmd           ; yes - move segment down.
       inc a                ; look right.
       ld b,a               ; set up GP y coord.
       call atadd           ; find attribute address.
       cp 68                ; mushrooms are bright (64) + green (4).
       jr z,segmd           ; mushroom to right, move down instead.
       inc (ix+2)           ; no obstacles, so move right.
       ret

; so segment is going left then!

segml  ld a,(ix+2)          ; y coord.
       and a                ; already at left edge of screen?
       jr z,segmd           ; yes - move segment down.
       dec a                ; look right.
       ld b,a               ; set up GP y coord.
       call atadd           ; find attribute address at (dispx,dispy).
       cp 68                ; mushrooms are bright (64) + green (4).
       jr z,segmd           ; mushroom to left, move down instead.
       dec (ix+2)           ; no obstacles, so move left.
       ret

; so segment is going down then!

segmd  ld a,(ix)            ; segment direction.
       xor 1                ; reverse it.
       ld (ix),a            ; store new direction.
       ld a,(ix+1)          ; y coord.
       cp 21                ; already at bottom of screen?
       jr z,segmt           ; yes - move segment to the top.

; At this point we're moving down regardless of any mushrooms that
; may block the segment's path.  Anything in the segment's way will
; be obliterated.

       inc (ix+1)           ; haven't reached the bottom, move down.
       ret

; moving segment to the top of the screen.

segmt  xor a                ; same as ld a,0 but saves 1 byte.
       ld (ix+1),a          ; new x coordinate = top of screen.
       ret

; Simple pseudo-random number generator.
; Steps a pointer through the ROM (held in seed), returning
; the contents of the byte at that location.

random ld hl,(seed)         ; Pointer
       ld a,h
```

```
        and 31              ; keep it within first 8k of ROM.
        ld h,a
        ld a,(hl)           ; Get "random" number from location.
        inc hl              ; Increment pointer.
        ld (seed),hl
        ret
seed    defw 0


; Calculate address of attribute for character at (dispx, dispy).

atadd   ld a,c              ; vertical coordinate.
        rrca                ; multiply by 32.
        rrca                ; Shifting right with carry 3 times is
        rrca                ; quicker than shifting left 5 times.
        ld e,a
        and 3
        add a,88            ; 88x256=address of attributes.
        ld d,a
        ld a,e
        and 224
        ld e,a
        ld a,b              ; horizontal position.
        add a,e
        ld e,a              ; de=address of attributes.
        ld a,(de)           ; return with attribute in accumulator.
        ret

plx     defb 0              ; player's x coordinate.
ply     defb 0              ; player's y coordinate.

; UDG graphics.

blocks  defb 16,16,56,56,124,124,254,254    ; player base.
        defb 24,126,255,255,60,60,60,60     ; mushroom.
        defb 24,126,126,255,255,126,126,24  ; segment.


; Table of segments.
; Format: 3 bytes per entry, 10 segments.
; byte 1: 255=segment off, 0=left, 1=right.
; byte 2 = x (vertical) coordinate.
; byte 3 = y (horizontal) coordinate.

segmnt  defb 0,0,0          ; segment 1.
        defb 0,0,0          ; segment 2.
        defb 0,0,0          ; segment 3.
        defb 0,0,0          ; segment 4.
        defb 0,0,0          ; segment 5.
        defb 0,0,0          ; segment 6.
        defb 0,0,0          ; segment 7.
        defb 0,0,0          ; segment 8.
        defb 0,0,0          ; segment 9.
        defb 0,0,0          ; segment 10.
```

Chapter Seven - Basic Alien Collision Detection

Coordinate Checking

Coordinate checking should be self-explanatory to most programmers, but is included here for the sake of completeness. It is also the next step in the development of our Centipede game.

The simplest type of collision detection would be something like this, used to detect if two UDGs have collided:

```
ld a,(playx)        ; player's x coordinate.
cp (ix+1)           ; compare with alien x.
ret nz              ; not the same, no collision.
ld a,(playy)        ; player's y coordinate.
cp (ix+2)           ; compare with alien y.
ret nz              ; not the same, no collision.
jp collis           ; we have a collision.
```

Okay, so that's pretty simple but most games don't use single-cell character graphics. What if the aliens are four character cells wide by two high, and the player's character is three squares high by three wide? We need to check if any part of the alien has collided with any part of the player, so we need to check that the coordinates are within a certain range. If the alien is less than two squares above the player, or less than 3 below him then the vertical coordinates match. If the alien is also less than four squares to the left of the player, and less than three squares to the right then the horizontal position also matches and we have a collision.

Let's write some code to do this. We can start by taking the player's vertical coordinate:

```
ld a,(playx)        ; player's x coordinate.
```

Then subtract the alien's vertical position:

```
sub (ix+1)          ; subtract alien x.
```

Next, subtract one from the player's *height*, and add it.

```
add a,2             ; player is 3 high, so add 3 - 1 = 2.
```

If the alien is within range the result will be *less than the combined height* of the player and alien, so we perform the check:

```
cp 5                ; combined heights are 3 + 2 = 5.
ret nc              ; not within vertical range.
```

Similarly, we can follow this with the code for the horizontal check:

```
ld a,(playy)        ; player's y coordinate.
sub (ix+2)          ; subtract alien y.
add a,2             ; player is 3 wide, so add 3 - 1 = 2.
cp 7                ; combined widths are 3 + 4 = 7.
ret nc              ; not within horizontal range.
jp collis           ; we have a collision.
```

Of course, this method doesn't just work for character-based graphics, it works perfectly well with sprites too, but more of those later. It's time to finish our game with some collision detection. As our graphics are all single-character UDGs we don't need anything fancy, a quick x=x and y=y check are all we need.

```
numseg equ 8                ; number of centipede segments.

; We want a black screen.

       ld a,71              ; white ink (7) on black paper (0),
                            ; bright (64).
       ld (23693),a         ; set our screen colours.
       xor a                ; quick way to load accumulator with zero.
       call 8859            ; set permanent border colours.

; Set up the graphics.

       ld hl,blocks         ; address of user-defined graphics data.
       ld (23675),hl        ; make UDGs point to it.

; Okay, let's start the game.

       call 3503            ; ROM routine - clears screen, opens chan 2.

       xor a                ; zeroise accumulator.
       ld (dead),a          ; set flag to say player is alive.

; Initialise coordinates.

       ld hl,21+15*256      ; load hl pair with starting coords.
       ld (plx),hl          ; set player coords.
       ld hl,255+255*256    ; player's bullets default.
       ld (pbx),hl          ; set bullet coords.

       ld b,10              ; number of segments to initialise.
       ld hl,segmnt         ; segment table.
segint ld (hl),1            ; start off moving right.
       inc hl
       ld (hl),0            ; start at top.
       inc hl
       ld (hl),b            ; use B register as y coordinate.
       inc hl
       djnz segint          ; repeat until all initialised.

       call basexy          ; set the x and y positions of the player.
       call splayr          ; show player base symbol.

; Now we want to fill the play area with mushrooms.

       ld a,68              ; green ink (4) on black paper (0),
                            ; bright (64).
       ld (23695),a         ; set our temporary colours.
       ld b,50              ; start with a few.
mushlp ld a,22              ; control code for AT character.
       rst 16
       call random          ; get a 'random' number.
       and 15               ; want vertical in range 0 to 15.
       rst 16
       call random          ; want another pseudo-random number.
       and 31               ; want horizontal in range 0 to 31.
       rst 16
       ld a,145             ; UDG 'B' is the mushroom graphic.
       rst 16               ; put mushroom on screen.
       djnz mushlp          ; loop back until all mushrooms displayed.


; This is the main loop.

mloop  equ $

; Delete the player.

       call basexy          ; set the x and y positions of the player.
       call wspace          ; display space over player.

; Now we've deleted the player we can move him before redisplaying him
```

```
; at his new coordinates.

       ld bc,63486         ; keyboard row 1-5/joystick port 2.
       in a,(c)            ; see what keys are pressed.
       rra                 ; outermost bit = key 1.
       push af             ; remember the value.
       call nc,mpl         ; it's being pressed, move left.
       pop af              ; restore accumulator.
       rra                 ; next bit along (value 2) = key 2.
       push af             ; remember the value.
       call nc,mpr         ; being pressed, so move right.
       pop af              ; restore accumulator.
       rra                 ; next bit (value 4) = key 3.
       push af             ; remember the value.
       call nc,mpd         ; being pressed, so move down.
       pop af              ; restore accumulator.
       rra                 ; next bit (value 8) reads key 4.
       push af             ; remember the value.
       call nc,mpu         ; it's being pressed, move up.
       pop af              ; restore accumulator.
       rra                 ; last bit (value 16) reads key 5.
       call nc,fire        ; it's being pressed, move up.

; Now he's moved we can redisplay the player.

       call basexy         ; set the x and y positions of the player.
       call splayr         ; show player.

; Now for the bullet.  First let's check to see if it's hit anything.

       call bchk           ; check bullet position.

       call dbull          ; delete bullets.
       call moveb          ; move bullets.
       call bchk           ; check new position of bullets.
       call pbull          ; print bullets at new position.

; Now for the centipede segments.

       ld ix,segmnt        ; table of segment data.
       ld b,10             ; number of segments in table.
censeg push bc
       ld a,(ix)           ; is segment switched on?
       inc a               ; 255=off, increments to zero.
       call nz,proseg      ; it's active, process segment.
       pop bc
       ld de,3             ; 3 bytes per segment.
       add ix,de           ; get next segment in ix registers.
       djnz censeg         ; repeat for all segments.

       halt                ; delay.

       ld a,(dead)         ; was the player killed by a segment?
       and a
       ret nz              ; player killed - lose a life.

; Jump back to beginning of main loop.

       jp mloop

; Move player left.

mpl    ld hl,ply           ; remember, y is the horizontal coord!
       ld a,(hl)           ; what's the current value?
       and a               ; is it zero?
       ret z               ; yes - we can't go any further left.

; now check that there isn't a mushroom in the way.

       ld bc,(plx)         ; current coords.
       dec b               ; look 1 square to the left.
       call atadd          ; get address of attribute at this position.
       cp 68               ; mushrooms are bright (64) + green (4).
```

```
       ret z               ; there's a mushroom - we can't move there.

       dec (hl)            ; subtract 1 from y coordinate.
       ret

; Move player right.

mpr    ld hl,ply           ; remember, y is the horizontal coord!
       ld a,(hl)           ; what's the current value?
       cp 31               ; is it at the right edge (31)?
       ret z               ; yes - we can't go any further left.

; now check that there isn't a mushroom in the way.

       ld bc,(plx)         ; current coords.
       inc b               ; look 1 square to the right.
       call atadd          ; get address of attribute at this position.
       cp 68               ; mushrooms are bright (64) + green (4).
       ret z               ; there's a mushroom - we can't move there.

       inc (hl)            ; add 1 to y coordinate.
       ret

; Move player up.

mpu    ld hl,plx           ; remember, x is the vertical coord!
       ld a,(hl)           ; what's the current value?
       cp 4                ; is it at upper limit (4)?
       ret z               ; yes - we can go no further then.

; now check that there isn't a mushroom in the way.

       ld bc,(plx)         ; current coords.
       dec c               ; look 1 square up.
       call atadd          ; get address of attribute at this position.
       cp 68               ; mushrooms are bright (64) + green (4).
       ret z               ; there's a mushroom - we can't move there.

       dec (hl)            ; subtract 1 from x coordinate.
       ret

; Move player down.

mpd    ld hl,plx           ; remember, x is the vertical coord!
       ld a,(hl)           ; what's the current value?
       cp 21               ; is it already at the bottom (21)?
       ret z               ; yes - we can't go down any more.

; now check that there isn't a mushroom in the way.

       ld bc,(plx)         ; current coords.
       inc c               ; look 1 square down.
       call atadd          ; get address of attribute at this position.
       cp 68               ; mushrooms are bright (64) + green (4).
       ret z               ; there's a mushroom - we can't move there.

       inc (hl)            ; add 1 to x coordinate.
       ret

; Fire a missile.

fire   ld a,(pbx)          ; bullet vertical coord.
       inc a               ; 255 is default value, increments to zero.
       ret nz              ; bullet on screen, can't fire again.
       ld hl,(plx)         ; player coordinates.
       dec l               ; 1 square higher up.
       ld (pbx),hl         ; set bullet coords.
       ret

bchk   ld a,(pbx)          ; bullet vertical.
       inc a               ; is it at 255 (default)?
       ret z               ; yes, no bullet on screen.
       ld bc,(pbx)         ; get coords.
```

```
        call atadd          ; find attribute here.
        cp 68               ; mushrooms are bright (64) + green (4).
        jr z,hmush          ; hit a mushroom!
        ret

hmush   ld a,22             ; AT control code.
        rst 16
        ld a,(pbx)          ; bullet vertical.
        rst 16
        ld a,(pby)          ; bullet horizontal.
        rst 16
        call wspace         ; set INK colour to white.
kilbul  ld a,255            ; x coord of 255 = switch bullet off.
        ld (pbx),a          ; destroy bullet.
        ret

; Move the bullet up the screen 1 character position at a time.

moveb   ld a,(pbx)          ; bullet vertical.
        inc a               ; is it at 255 (default)?
        ret z               ; yes, no bullet on screen.
        sub 2               ; 1 row up.
        ld (pbx),a
        ret

; Set up the x and y coordinates for the player's gunbase position,
; this routine is called prior to display and deletion of gunbase.

basexy  ld a,22             ; AT code.
        rst 16
        ld a,(plx)          ; player vertical coord.
        rst 16              ; set vertical position of player.
        ld a,(ply)          ; player's horizontal position.
        rst 16              ; set the horizontal coord.
        ret

; Show player at current print position.

splayr  ld a,69             ; cyan ink (5) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary screen colours.
        ld a,144            ; ASCII code for User Defined Graphic 'A'.
        rst 16              ; draw player.
        ret

pbull   ld a,(pbx)          ; bullet vertical.
        inc a               ; is it at 255 (default)?
        ret z               ; yes, no bullet on screen.
        call bullxy
        ld a,16             ; INK control char.
        rst 16
        ld a,6              ; 6 = yellow.
        rst 16
        ld a,147            ; UDG 'D' is used for player bullets.
        rst 16
        ret

dbull   ld a,(pbx)          ; bullet vertical.
        inc a               ; is it at 255 (default)?
        ret z               ; yes, no bullet on screen.
        call bullxy         ; set up bullet coordinates.
wspace  ld a,71             ; white ink (7) on black paper (0),
                            ; bright (64).
        ld (23695),a        ; set our temporary screen colours.
        ld a,32             ; SPACE character.
        rst 16              ; display space.
        ret

; Set up the x and y coordinates for the player's bullet position,
; this routine is called prior to display and deletion of bullets.

bullxy  ld a,22             ; AT code.
        rst 16
```

```
        ld a,(pbx)          ; player bullet vertical coord.
        rst 16              ; set vertical position of player.
        ld a,(pby)          ; bullet's horizontal position.
        rst 16              ; set the horizontal coord.
        ret

segxy   ld a,22             ; ASCII code for AT character.
        rst 16              ; display AT code.
        ld a,(ix+1)         ; get segment x coordinate.
        rst 16              ; display coordinate code.
        ld a,(ix+2)         ; get segment y coordinate.
        rst 16              ; display coordinate code.
        ret

proseg  call segcol         ; segment collision detection.
        ld a,(ix)           ; check if segment was switched off
        inc a               ; by collision detection routine.
        ret z               ; it was, so this segment is now dead.
        call segxy          ; set up segment coordinates.
        call wspace         ; display a space, white ink on black.
        call segmov         ; move segment.
        call segcol         ; new segment position collision check.
        ld a,(ix)           ; check if segment was switched off
        inc a               ; by collision detection routine.
        ret z               ; it was, so this segment is now dead.
        call segxy          ; set up segment coordinates.
        ld a,2              ; attribute code 2 = red segment.
        ld (23695),a        ; set temporary attributes.
        ld a,146            ; UDG 'C' to display segment.
        rst 16
        ret
segmov  ld a,(ix+1)         ; x coord.
        ld c,a              ; GP x area.
        ld a,(ix+2)         ; y coord.
        ld b,a              ; GP y area.
        ld a,(ix)           ; status flag.
        and a               ; is the segment heading left?
        jr z,segml          ; going left - jump to that bit of code.

; so segment is going right then!

segmr   ld a,(ix+2)         ; y coord.
        cp 31               ; already at right edge of screen?
        jr z,segmd          ; yes - move segment down.
        inc a               ; look right.
        ld b,a              ; set up GP y coord.
        call atadd          ; find attribute address.
        cp 68               ; mushrooms are bright (64) + green (4).
        jr z,segmd          ; mushroom to right, move down instead.
        inc (ix+2)          ; no obstacles, so move right.
        ret

; so segment is going left then!

segml   ld a,(ix+2)         ; y coord.
        and a               ; already at left edge of screen?
        jr z,segmd          ; yes - move segment down.
        dec a               ; look right.
        ld b,a              ; set up GP y coord.
        call atadd          ; find attribute address at (dispx,dispy).
        cp 68               ; mushrooms are bright (64) + green (4).
        jr z,segmd          ; mushroom to left, move down instead.
        dec (ix+2)          ; no obstacles, so move left.
        ret

; so segment is going down then!

segmd   ld a,(ix)           ; segment direction.
        xor 1               ; reverse it.
        ld (ix),a           ; store new direction.
        ld a,(ix+1)         ; y coord.
        cp 21               ; already at bottom of screen?
        jr z,segmt          ; yes - move segment to the top.
```

```
; At this point we're moving down regardless of any mushrooms that
; may block the segment's path.  Anything in the segment's way will
; be obliterated.

       inc (ix+1)          ; haven't reached the bottom, move down.
       ret

; moving segment to the top of the screen.

segmt  xor a               ; same as ld a,0 but saves 1 byte.
       ld (ix+1),a         ; new x coordinate = top of screen.
       ret

; Segment collision detection.
; Checks for collisions with player and player's bullets.

segcol ld a,(ply)          ; bullet y position.
       cp (ix+2)           ; is it identical to segment y coord?
       jr nz,bulcol        ; y coords differ, try bullet instead.
       ld a,(plx)          ; player x coord.
       cp (ix+1)           ; same as segment?
       jr nz,bulcol        ; x coords differ, try bullet instead.

; So we have a collision with the player.

killpl ld (dead),a         ; set flag to say that player is now dead.
       ret

; Let's check for a collision with the player's bullet.

bulcol ld a,(pbx)          ; bullet x coords.
       inc a               ; at default value?
       ret z               ; yes, no bullet to check for.
       cp (ix+1)           ; is bullet x coord same as segment x coord?
       ret nz              ; no, so no collision.
       ld a,(pby)          ; bullet y position.
       cp (ix+2)           ; is it identical to segment y coord?
       ret nz              ; no - no collision this time.

; So we have a collision with the player's bullet.

       call dbull          ; delete bullet.
       ld a,22             ; AT code.
       rst 16
       ld a,(pbx)          ; player bullet vertical coord.
       inc a               ; 1 line down.
       rst 16              ; set vertical position of mushroom.
       ld a,(pby)          ; bullet's horizontal position.
       rst 16              ; set the horizontal coord.
       ld a,16             ; ASCII code for INK control.
       rst 16
       ld a,4              ; 4 = colour green.
       rst 16              ; we want all mushrooms in this colour!
       ld a,145            ; UDG 'B' is the mushroom graphic.
       rst 16              ; put mushroom on screen.
       call kilbul         ; kill the bullet.
       ld (ix),a           ; kill the segment.
       ld hl,numseg        ; number of segments.
       dec (hl)            ; decrement it.
       ret

; Simple pseudo-random number generator.
; Steps a pointer through the ROM (held in seed), returning
; the contents of the byte at that location.

random ld hl,(seed)        ; Pointer
       ld a,h
       and 31              ; keep it within first 8k of ROM.
       ld h,a
       ld a,(hl)           ; Get "random" number from location.
       inc hl              ; Increment pointer.
       ld (seed),hl
```

```
       ret
seed   defw 0


; Calculate address of attribute for character at (dispx, dispy).

atadd  ld a,c              ; vertical coordinate.
       rrca                ; multiply by 32.
       rrca                ; Shifting right with carry 3 times is
       rrca                ; quicker than shifting left 5 times.
       ld e,a
       and 3
       add a,88            ; 88x256=address of attributes.
       ld d,a
       ld a,e
       and 224
       ld e,a
       ld a,b              ; horizontal position.
       add a,e
       ld e,a              ; de=address of attributes.
       ld a,(de)           ; return with attribute in accumulator.
       ret

plx    defb 0              ; player's x coordinate.
ply    defb 0              ; player's y coordinate.
pbx    defb 255            ; player's bullet coordinates.
pby    defb 255
dead   defb 0              ; flag - player dead when non-zero.

; UDG graphics.

blocks defb 16,16,56,56,124,124,254,254    ; player base.
       defb 24,126,255,255,60,60,60,60     ; mushroom.
       defb 24,126,126,255,255,126,126,24  ; segment.
       defb 0,102,102,102,102,102,102,0    ; player bullet.


; Table of segments.
; Format: 3 bytes per entry, 10 segments.
; byte 1: 255=segment off, 0=left, 1=right.
; byte 2 = x (vertical) coordinate.
; byte 3 = y (horizontal) coordinate.

segmnt defb 0,0,0          ; segment 1.
       defb 0,0,0          ; segment 2.
       defb 0,0,0          ; segment 3.
       defb 0,0,0          ; segment 4.
       defb 0,0,0          ; segment 5.
       defb 0,0,0          ; segment 6.
       defb 0,0,0          ; segment 7.
       defb 0,0,0          ; segment 8.
       defb 0,0,0          ; segment 9.
       defb 0,0,0          ; segment 10.
```

But wait, why are there *two* tests for collision detection instead of one? Well, imagine the player's gunbase is one character cell to the left of a centipede segment. The player is moving right and the segment is moving left. In the next frame the segment would move into the cell occupied by the player, and the player would move into the position occupied by the segment in the previous frame - player and centipede segment would move straight through each other and a single collision detection check would fail to pick this up. By checking for a collision after the player moves, and then again after the centipede segments have moved we can avoid the problem.

Collisions Between Sprites

Fair enough, most Spectrum games use sprites rather than UDGs so in the next chapter we shall see

how sprites may be drawn. For collision detection, the same principle of coordinate checking can be used to detect collisions between sprites. Subtract the first sprite's coordinates from those of the second, examine the difference and if it's within the size range of the two sprites combined we have a collision on that axis. A simple collision check for two 16x16 pixel sprites might look something like this:

```
; Check (l, h) for collision with (c, b), strict enforcement.

colx16 ld a,l              ; x coord.
       sub c               ; subtract x.
       add a,15            ; add maximum distance.
       cp 31               ; within x range?
       ret nc              ; no - they've missed.
       ld a,h              ; y coord.
       sub b               ; subtract y.
       add a,15            ; add maximum distance.
       cp 31               ; within y range?
       ret                 ; carry flag set if there's a collision.
```

There is a drawback with this method. If your sprites don't entirely fill their 16x16 pixel boundaries then the collision detection will appear to be too strict, and collisions will happen when sprites are close together but not actually touching. A slightly less sensitive check would involve clipping the corners of the sprites into a more octagonal shape, particularly if your sprites have rounded corners. The routine below works by adding the x and y coordinate differences and checking that they are below a certain limit. For a collision between two 16x16 sprites the maximum coordinate distances are 15 pixels for each axis, so by checking that the x and y differences are 25 or less we are effectively shaving a 5x5x5 pixel triangle from each corner.

```
; Check (l, h) for collision with (c, b), cutting corners.

colc16 ld a,l              ; x coord.
       sub c               ; subtract x.
       jr nc,colc1a        ; result is positive.
       neg                 ; make negative positive.
colc1a cp 16               ; within x range?
       ret nc              ; no - they've missed.
       ld e,a              ; store difference.

       ld a,h              ; y coord.
       sub b               ; subtract y.
       jr nc,colc1b        ; result is positive.
       neg                 ; make negative positive.
colc1b cp 16               ; within y range?
       ret nc              ; no - they've missed.

       add a,e             ; add x difference.
       cp 26               ; only 5 corner pixels touching?
       ret                 ; carry set if there's a collision.
```

Chapter Eight - Sprites

Converting Pixel Positions to Screen Addresses

UDGs and character graphics are all very fine and dandy, but the better games usually use sprites and there are no handy ROM routines to help us here because Sir Clive didn't design the Spectrum as a games machine. Sooner or later a games programmer has to confront the thorny issue of the Spectrum's awkward screen layout. It's a tricky business converting x and y pixel coordinates into a screen address but there are a couple of methods we might employ to do this.

Using a Screen Address Look-up Table

The first method is to use a pre-calculated address table containing the screen address for each of the Spectrum's 192 lines such as this, or a similar variation:

```
        xor a               ; clear carry flag and accumulator.
        ld d,a              ; empty de high byte.
        ld a,(xcoord)       ; x position.
        rla                 ; shift left to multiply by 2.
        ld e,a              ; place this in low byte of de pair.
        rl d                ; shift top bit into de high byte.
        ld hl,addtab        ; table of screen addresses.
        add hl,de           ; point to table entry.
        ld e,(hl)           ; low byte of screen address.
        inc hl              ; point to high byte.
        ld d,(hl)           ; high byte of screen address.
        ld a,(ycoord)       ; horizontal position.
        rra                 ; divide by two.
        rra                 ; and again for four.
        rra                 ; shift again to divide by eight.
        and 31              ; mask away rubbish shifted into rightmost bits.
        add a,e             ; add to address for start of line.
        ld e,a              ; new value of e register.
        ret                 ; return with screen address in de.

.
.

addtab defw 16384
       defw 16640
       defw 16896

.
.
```

On the plus side this is very fast, but it does mean having to store each of the 192 line addresses in a table, taking up 384 bytes which might be better employed elsewhere.

Calculating Screen Addresses

The second method involves calculating the address ourselves and doesn't require an address look-up table. In doing this we need to consider three things: Which third of the screen the point is in, the character line to which it is closest, and the pixel line upon which it falls within that cell. Judicious use of the and operand will help us to decide all three. It is a complicated business however, so please bear with me as I endeavour to explain how it works.

We can establish which of the three screen segments a point is situated in by taking the vertical coordinate and masking away the six least significant bits to leave a value of 0, 64 or 128 each of the segments being 64 pixels apart. As the high bytes of the 3 screen segment addresses are 64, 72 and

80 - a difference of 8 going from one segment to another - we take this masked value and divide it by 8 to give us a value of 0, 8 or 16. We then add 64 to give us the high byte of the screen segment.

Each segment is divided into 8 character cell positions which are 32 bytes apart, so to find that aspect of our address we take the vertical coordinate and mask away the two most significant bits we used to determine the segment along with the three least significant bits which determine the pixel position. The instruction and 56 will do nicely. This gives us the character cell position as a multiple of 8, and as the character lines are 32 bytes apart we multiply this by 4 and place our number in the low byte of the screen address.

Finally, character cells are further divided into pixel lines 256 bytes apart, so we again take our vertical coordinate, mask away everything except the bits which determine the line using and 7, and add the result to the high byte. That will give us our vertical screen address. From there we take our horizontal coordinate, divide it by 8 and add it to our address.

Here is a routine which returns a screen address for (xcoord, ycoord) in the de register pair. It could easily be modified to return the address in the hl or bc registers if desired.

```
scadd  ld a,(xcoord)       ; fetch vertical coordinate.
       ld e,a              ; store that in e.

; Find line within cell.

       and 7               ; line 0-7 within character square.
       add a,64            ; 64 * 256 = 16384 = start of screen display.
       ld d,a              ; line * 256.

; Find which third of the screen we're in.

       ld a,e              ; restore the vertical.
       and 192             ; segment 0, 1 or 2 multiplied by 64.
       rrca                ; divide this by 8.
       rrca
       rrca                ; segment 0-2 multiplied by 8.
       add a,d             ; add to d give segment start address.
       ld d,a

; Find character cell within segment.

       ld a,e              ; 8 character squares per segment.
       rlca                ; divide x by 8 and multiply by 32,
       rlca                ; net calculation: multiply by 4.
       and 224             ; mask off bits we don't want.
       ld e,a              ; vertical coordinate calculation done.

; Add the horizontal element.

       ld a,(ycoord)       ; y coordinate.
       rrca                ; only need to divide by 8.
       rrca
       rrca
       and 31              ; squares 0 - 31 across screen.
       add a,e             ; add to total so far.
       ld e,a              ; de = address of screen.
       ret
```

Shifting

Once the address has been established we need to consider how our graphics are shifted into position. The three lowest bit positions of the horizontal coordinate indicate how many pixel shifts are needed. A slow way to plot a pixel would be to call the scadd routine above, perform an and 7

on the horizontal coordinate, then right shift a pixel from zero to seven times depending on the result before dumping it to the screen.

A shifter sprite routine works in the same way. The graphic image is taken from memory one line at a time, shifted into position and then placed on the screen before moving to the next line down and repeating the process. We could write a sprite routine which calculated the screen address for every line drawn, and indeed the first sprite routine I ever wrote worked in such a way. A simpler method is to determine whether we're moving within a character cell, crossing character cell boundaries, or crossing a segment boundary with a couple of `and` instructions and to increment or decrement the screen address accordingly. Put simply, `and 63` will return zero if the new vertical position is crossing a segment, `and 7` will return zero if it is crossing a character cell boundary and anything else means the new line is within the same character cell as the previous line.

This is a shifter sprite routine which makes use of the earlier scadd routine. To use it simply set up the coordinates in dispx and dispy, point the bc register pair at the sprite graphic, and `call sprite`.

```
sprit7 xor 7               ; complement last 3 bits.
       inc a               ; add one for luck!
sprit3 rl d                ; rotate left...
       rl c                ; ...into middle byte...
       rl e                ; ...and finally into left character cell.
       dec a               ; count shifts we've done.
       jr nz,sprit3        ; return until all shifts complete.

; Line of sprite image is now in e + c + d, we need it in form c + d + e.

       ld a,e              ; left edge of image is currently in e.
       ld e,d              ; put right edge there instead.
       ld d,c              ; middle bit goes in d.
       ld c,a              ; and the left edge back into c.
       jr sprit0           ; we've done the switch so transfer to screen.

sprite ld a,(dispx)        ; draws sprite (hl).
       ld (tmp1),a         ; store vertical.
       call scadd          ; calculate screen address.
       ld a,16             ; height of sprite in pixels.
sprit1 ex af,af'           ; store loop counter.
       push de             ; store screen address.
       ld c,(hl)           ; first sprite graphic.
       inc hl              ; increment poiinter to sprite data.
       ld d,(hl)           ; next bit of sprite image.
       inc hl              ; point to next row of sprite data.
       ld (tmp0),hl        ; store in tmp0 for later.
       ld e,0              ; blank right byte for now.
       ld a,b              ; b holds y position.
       and 7               ; how are we straddling character cells?
       jr z,sprit0         ; we're not straddling them, don't bother shifting.
       cp 5                ; 5 or more right shifts needed?
       jr nc,sprit7        ; yes, shift from left as it's quicker.
       and a               ; oops, carry flag is set so clear it.
sprit2 rr c                ; rotate left byte right...
       rr d                ; ...through middle byte...
       rr e                ; ...into right byte.
       dec a               ; one less shift to do.
       jr nz,sprit2        ; return until all shifts complete.
sprit0 pop hl              ; pop screen address from stack.
       ld a,(hl)           ; what's there already.
       xor c               ; merge in image data.
       ld (hl),a           ; place onto screen.
       inc l               ; next character cell to right please.
       ld a,(hl)           ; what's there already.
       xor d               ; merge with middle bit of image.
       ld (hl),a           ; put back onto screen.
       inc hl              ; next bit of screen area.
       ld a,(hl)           ; what's already there.
```

```
       xor e              ; right edge of sprite image data.
       ld (hl),a          ; plonk it on screen.
       ld a,(tmp1)        ; temporary vertical coordinate.
       inc a              ; next line down.
       ld (tmp1),a        ; store new position.
       and 63             ; are we moving to next third of screen?
       jr z,sprit4        ; yes so find next segment.
       and 7              ; moving into character cell below?
       jr z,sprit5        ; yes, find next row.
       dec hl             ; left 2 bytes.
       dec l              ; not straddling 256-byte boundary here.
       inc h              ; next row of this character cell.
sprit6 ex de,hl           ; screen address in de.
       ld hl,(tmp0)       ; restore graphic address.
       ex af,af'          ; restore loop counter.
       dec a              ; decrement it.
       jp nz,sprit1       ; not reached bottom of sprite yet to repeat.
       ret                ; job done.
sprit4 ld de,30           ; next segment is 30 bytes on.
       add hl,de          ; add to screen address.
       jp sprit6          ; repeat.
sprit5 ld de,63774        ; minus 1762.
       add hl,de          ; subtract 1762 from physical screen address.
       jp sprit6          ; rejoin loop.
```

As you can see, this routine utilises the xor instruction to merge the sprite onto the screen, which works in the same way that PRINT OVER 1 does in Sinclair BASIC. The sprite is merged with any graphics already present on screen which can look messy. To delete a sprite we just display it again and the image magically vanishes.

If we wanted to draw a sprite on top of something that is already on the screen we would need some extra routines, similar to the one above. One would be required to store the graphics on screen in a buffer so that that portion of the screen could be re-drawn when the sprite is deleted. The next routine would apply a sprite mask to remove the pixels around and behind the sprite using and or or, then the sprite could finally be applied over the top. Another routine would be needed to restore the relevant portion of screen to its former state should the sprite be deleted. However, this would take a lot of CPU time to achieve so my advice would be not to bother unless your game uses something called *double buffering* - otherwise known as the back screen technique, or you're using a pre-shifted sprites, which we shall discuss shortly.

Another method you may wish to consider involves making sprites appear to pass behind background objects, a trick you may have seen in Haunted House or Egghead in Space. While this method is handy for reducing colour clash it requires a sizeable chunk of memory. In both games a 6K dummy mask screen was located at address 24576, and each byte of sprite data was anded with the data on the dummy screen before being xored onto the physical screen located at address 16384. Because the physical screen and the dummy mask screen were exactly 8K apart it was possible to flip between them by toggling bit 5 of the h register. To do this for the sprite routine above our sprit0 routine might look like this:

```
sprit0 pop hl             ; pop screen address from stack.
       set 5,h            ; address of dummy screen.
       ld a,(hl)          ; what's there already.
       and c              ; mask away parts behind the object.
       res 5,h            ; address of physical screen.
       xor (hl)           ; merge in image data.
       ld (hl),a          ; place onto screen.
       inc l              ; next character cell to right please.
       set 5,h            ; address of dummy screen.
       ld a,(hl)          ; what's there already.
       and d              ; mask with middle bit of image.
       res 5,h            ; address of physical screen.
       xor (hl)           ; merge in image data.
```

```
        ld (hl),a           ; put back onto screen.
        inc hl              ; next bit of screen area.
        set 5,h             ; address of dummy screen.
        ld a,(hl)           ; what's already there.
        and e               ; mask right edge of sprite image data.
        res 5,h             ; address of physical screen.
        xor (hl)            ; merge in image data.
        ld (hl),a           ; plonk it on screen.
        ld a,(tmp1)         ; temporary vertical coordinate.
```

Here is another method to work out the address of the next line down. As before, we calculate it from our current address. Doing so is always quite tricky because of the way the screen is structured. Most of the time our next line is 256 bytes on from the preceding one, but every time we cross a character cell boundary we need to subtract 1760, and when moving from one third of the screen to the next, we need to add 32. This little snippet calculates the address of the next line in the hl register pair without corrupting de.

```
; Line drawn, now work out next target address.

        inc h               ; increment pixel.
        ld a,h              ; get pixel address.
        and 7               ; straddling character position?
        ret nz              ; no, we're on next line already.
        ld a,h              ; get pixel address.
        sub 8               ; subtract 8 for start of segment.
        ld h,a              ; new high byte of address.
        ld a,l              ; get low byte of address.
        add a,32            ; one line down.
        ld l,a              ; new low byte.
        ret nc              ; not reached next segment yet.
        ld a,h              ; address high.
        add a,8             ; add 8 to next segment.
        ld h,a              ; new high byte.
```

Pre-shifted Sprites

A shifter sprite routine has one major drawback: its lack of speed. Shifting all that graphic data into position takes time, and if your game needs a lot of sprites bouncing around the screen, you should consider using pre-shifted sprites instead. This requires up to eight separate copies of the sprite image, one in each of the shifted pixel positions. Most commonly, sprites are moved in 2-pixel jumps and so 4 copies of the sprite are held. It is then simply a matter of calculating which sprite image to use based on the horizontal alignment of the sprite, calculating the screen address, and copying the sprite image to the screen. While this method is much faster it is fantastically expensive in memory terms. A shifter sprite routine requires 32 bytes for an unmasked 16x16 pixel sprite, a pre-shifted sprite with 4 positions requires 128 bytes for the same image and for 8 positions a whopping 256! Writing a Spectrum game is always a compromise between speed and available memory.

You may not necessarily want the same sprite image in each pre-shifted position. For example, by changing the position of a sprite's legs in each of the pre-shifted positions a sprite can be animated to appear as if it is walking from left to right as it moves across the screen. Remember to match the character's legs to the number of pixels it is moved each frame. If you are moving a sprite 2 pixels each frame it is important to make the legs move 2 pixels between frames. Less than this will make the sprite appear as if it is skating on ice, any more and it will appear to be struggling for grip. I'll let you into a little secret here: believe it or not, this can actually affect the way a game feels so getting your animation right is important.

The Byte Scan Method

If you are displaying sprites directly on the screen and wish to display more than a couple, you may soon encounter a problem. Deleting, moving and redisplaying takes time to achieve, and while this is happening the TV scan line is on the move. If the scan line reaches the sprite while it is being displayed or deleted, you may experience flicker. One way to avoid this is to use the byte scan method, or a variation of it.

The byte scan method involves drawing the new sprite while simultaneously deleting the old one, byte for byte, or line for line. If the scan line ever catches up with your sprite, the only bit of flicker you will see is on the byte or line being drawn at the time. So it doesn't really matter how many sprites you have on screen, or where they are, because they never flicker. The down side is that this can be tricky to implement, involves keeping two copies of each sprite's coordinates, image, frame and all relevant info, and requires a lot of registers. Fortunately, the Z80A has a very useful instruction for flipping between two banks of registers very quickly: EXX.

Once we have called the routines to work out our old sprite's graphic data in de, the screen address in hl and maybe a mask in bc, we would then use the exx instruction to flip them into the alternate register set, then call the routines again to put the new sprite data pointer, screen address and mask in the hl, de and bc registers. It is then a case of drawing each line at a time, calculating the address of the next screen line, flipping the registers with exx, drawing the line of the new sprite, calculating the next screen address, then flipping the registers back again. Repeated in a loop, this process will replace a sprite without ever deleting it completely from the screen.

Don't forget, you will still need to have a simple sprite routine which draws a sprite without deleting an old one, or deletes a sprite without redisplaying a new one. Otherwise, you will not be able to bring new sprites on to the screen or remove them when they are no longer required.

Chapter Nine - Background Graphics

Displaying Blocks

Let us say that we want to write a single screen maze game. We need to display the walls around which the player's sprite is to be manipulated, and the best way to do this is to create a table of blocks which are transferred to the screen sequentially. As we step through the table we find the address of the block graphic, calculate the screen address and dump the character to the screen.

We will start with the character display routine. Unlike a sprite routine we need to deal with character positions, and luckily it is easier to calculate a screen address for a character position than it is for individual pixels.

There are 24 vertical and 32 horizontal character cell positions on the Spectrum screen, so our coordinates will be between (0,0) and (23,31). Rows 0-7 fall in the first segment, 8-15 in the middle section and positions 16-23 in the third portioin of the screen. As luck would have it, the high byte of the screen address for each segment increases by 8 from one segment to the next, so by taking the vertical cell number and performing an `and 24` we immediately get the displacement to the start of relevant segment's screen address right there. Add 64 for the start of the Spectrum's screen and we have the high byte of our address. We then need to find the correct character cell within each segment, so we again take the vertical coordinate, and this time use `and 7` to determine which of the seven rows we're trying to find. We multiply this by the character width of the screen - 32 - and add the horizontal cell number to find the low byte of the screen address. A suitable example is below:

```
; Return character cell address of block at (b, c).

chadd  ld a,b              ; vertical position.
       and 24              ; which segment, 0, 1 or 2?
       add a,64            ; 64*256 = 16384, Spectrum's screen memory.
       ld d,a              ; this is our high byte.
       ld a,b              ; what was that vertical position again?
       and 7               ; which row within segment?
       rrca                ; multiply row by 32.
       rrca
       rrca
       ld e,a              ; low byte.
       ld a,c              ; add on y coordinate.
       add a,e             ; mix with low byte.
       ld e,a              ; address of screen position in de.
       ret
```

Once we have our screen address it is a straightforward process to dump the character onto the screen. As long as we are not crossing character cell boundaries the next screen line will always fall 256 bytes after its predecessor, so we increment the high byte of the address to find the next line.

```
; Display character hl at (b, c).

char   call chadd          ; find screen address for char.
       ld b,8              ; number of pixels high.
char0  ld a,(hl)           ; source graphic.
       ld (de),a           ; transfer to screen.
       inc hl              ; next piece of data.
       inc d               ; next pixel line.
       djnz char0          ; repeat
       ret
```

As for colouring our block, we covered that in the chapter on simple attribute collision detection. The atadd routine will give us the address of an attribute cell at character cell (b, c).

Lastly, we need to decide which block to display at each cell position. Say we need 3 types of block for our game - we might use block type 0 for a space, 1 for a wall and 2 for a key. We would arrange the graphics and attributes for each block in separate tables in the same order:

```
blocks equ $

; block 0 = space character.

       defb 0,0,0,0,0,0,0,0

; block 1 = wall.

       defb 1,1,1,255,16,16,16,255

; block 2 = key.

       defb 6,9,9,14,16,32,80,32

attrs  equ $

; block 0 = space.

       defb 71

; block 1 = wall.

       defb 22

; block 2 = key.

       defb 70
```

As we step through our table of up to 24 rows and 32 columns of maze blocks we load the block number into the accumulator, and call the fblock and fattr routines below to obtain the source graphic and attribute addresses.

```
; Find cell graphic.

fblock rlca                ; multiply block number by eight.
       rlca
       rlca
       ld e,a              ; displacement to graphic address.
       ld d,0              ; no high byte.
       ld hl,blocks        ; address of character blocks.
       add hl,de           ; point to block.
       ret

; Find cell attribute.

fattr  ld e,a              ; displacement to attribute address.
       ld d,0              ; no high byte.
       ld hl,attrs         ; address of block attributes.
       add hl,de           ; point to attribute.
       ret
```

Using this method means our maze data requires one byte of RAM for every character cell. For a playing area of 32 cells wide and 16 blocks high this would mean each screen occupying 512 bytes of memory. That would be fine for a 20-screen platformer like Manic Miner, but if you want a hundred screens or more you should consider using bigger blocks so that less are required for each screen. By using character cell blocks which are 16 x 16 pixels instead of 8 x 8 in our example, each screen table would require only 128 bytes meaning more could be squeezed into the Spectrum's memory.

Chapter Ten - Scores and High Scores

More Scoring Routines

Up until now we have gotten away with an unsophisticated scoring routine. Our score is held as a 16-bit number stored in a register pair, and to display it we have made use of the Sinclair ROM's line number print/display routine. There are two main drawbacks to this method, firstly we are limited to numbers 0-9999, and secondly it looks awful.

We could convert a 16-bit number to ASCII ourselves like this:

```
; Show number passed in hl, right-justified.

shwnum ld a,48 (or 32)     ; leading zeroes (or spaces).
       ld de,10000         ; ten thousands column.
       call shwdg          ; show digit.
       ld de,1000          ; thousands column.
       call shwdg          ; show digit.
       ld de,100           ; hundreds column.
       call shwdg          ; show digit.
       ld de,10            ; tens column.
       call shwdg          ; show digit.
       or 16               ; last digit is always shown.
       ld de,1             ; units column.
shwdg  and 48              ; clear carry, clear digit.
shwdg1 sbc hl,de           ; subtract from column.
       jr c,shwdg0         ; nothing to show.
       or 16               ; something to show, make it a digit.
       inc a               ; increment digit.
       jr shwdg1           ; repeat until column is zero.
shwdg0 add hl,de           ; restore total.
       push af
       rst 16              ; show character.
       pop af
       ret
```

This method works well, though we're still limited to a five-digit score of no more than 65535. For a more professional-looking affair complete with any number of leading zeroes we need to hold the score as a string of ASCII digits.

I have used the same scoring technique for something like 15 years now, it isn't terribly sophisticated but it's good enough to do what we need. This method uses one ASCII character per digit, which makes it easy to display. Incidentally, this routine is taken from the shoot-em-up More Tea, Vicar?

```
score  defb '000000'
uscor  ld a,(hl)           ; current value of digit.
       add a,b             ; add points to this digit.
       ld (hl),a           ; place new digit back in string.
       cp 58               ; more than ASCII value '9'?
       ret c               ; no - relax.
       sub 10              ; subtract 10.
       ld (hl),a           ; put new character back in string.
uscor0 dec hl              ; previous character in string.
       inc (hl)            ; up this by one.
       ld a,(hl)           ; what's the new value?
       cp 58               ; gone past ASCII nine?
       ret c               ; no, scoring done.
       sub 10              ; down by ten.
       ld (hl),a           ; put it back
       jp uscor0           ; go round again.
```

To use this we point hl at the digit we would like to increase, place the amount we want to add in the

b register, then call uscor. For example, to add 250 to the score requires 6 lines:

```
; Add 250 to the score.

       ld hl,score+3       ; point to hundreds column.
       ld b,2              ; 2 hundreds = 200.
       call uscor          ; increment the score.
       ld hl,score+4       ; point to tens column.
       ld b,5              ; 5 tens = 50.
       call uscor          ; up the score.
```

Simple, but it does the job. Pedants would no doubt point out that this could be done using BCD, and that all the opcodes for this are in the Z80 instruction set.

High Score Tables

High Score routines are not especially easy to write for a beginner, but once you have written one it can be re-used again and again. The basic principle is that we start at the bottom of the table and work our way up until we find a score that is greater than, or equal to, the player's score. We then shift all the data in the table below that point down by one position, and copy our player's name and score into the table at that point.

We can set the hl or ix register pair to the first digit of the bottom score in the table and work our way comparing each digit to the corresponding one in the player's score. If the digit in the player's score is higher we move up a position, if it is lower we stop there and copy the player's score into the table one place below. If both digits are the same we move to the next digit and repeat the check until the digits are different or we have checked all the digits in the score. If the scores are identical we place the player's entry in the table one place below. This is repeated until a score in the table is higher than the player's score, or we reach the top of the table.

When first initialising a high score table it may be tempting to place your own name at the top with a score that is very difficult to beat. Try to resist this temptation. High score tables are for the player to judge his own performance, and there is no point in frustrating the player by making it difficult to reach the top position.

Chapter Eleven - Enemy Movement

So we have our playfield, and can allow the player to manipulate a sprite around it, but what we now need are some enemy sprites for the player to avoid. A new programmer could struggle here, but it really is far simpler than it first appears.

Patrolling Enemies

The easiest type of enemy to program is that with a fixed algorithm to follow, or a predetermined patrol route. We covered one such technique in the Centipede game earlier. Another very simple example is that found in games such as Jetset Willy, where a sprite travels in a single direction until it reaches the end of its patrol, then switches direction and heads back to its starting point, before changing direction again and starting the cycle again. As you might imagine, these routines are incredibly easy to write.

Firstly we set up our alien structure table with minimum and maximum coordinate positions and the present direction. It's generally a good idea to comment these tables so we'll do that too.

```
; Alien data table, 6 bytes per alien.
; ix     = graphic type, ie chicken/Amoebatron etc.
; ix + 1 = direction, 0=up, 1=right, 2=down, 3=left.
; ix + 2 = current x coordinate.
; ix + 3 = current y coordinate.
; ix + 4 = minimum x or y coord, depends on direction.
; ix + 5 = maximum x or y coord, depends on direction.

altab  defb 0,0,0,0,0,0
       defb 0,0,0,0,0,0
       defb 0,0,0,0,0,0
```

then to manipulate our sprite we might write something like this

```
       ld a,(ix+1)         ; alien movement direction.
       rra                 ; rotate low bit into carry.
       jr nc,movav         ; no carry = 0 or 2, must be vertical.

; direction is 1 or 3 so it's horizontal.

       rra                 ; rotate next bit into carry for test.
       jr nc,movar         ; direction 1 = move alien right.

; Move alien left.

moval  ld a,(ix+3)         ; get y coordinate.
       sub 2               ; move left.
       ld (ix+3),a
       cp (ix+4)           ; reached mimimum yet?
       jr z,movax          ; yes - change direction.
       jr c,movax          ; oops, gone past it.
       ret

; Move alien right.

movar  ld a,(ix+3)         ; get y coordinate.
       add a,2             ; move right.
       ld (ix+3),a
       cp (ix+5)           ; reached maximum yet?
       jr nc,movax         ; yes - change direction.
       ret
```

```
; Move alien vertically.

movav  rra                  ; test direction.
       jr c,movad           ; direction 2 is down.

; Move alien up.

movau  ld a,(ix+2)          ; get x coordinate.
       sub 2                ; move up.
       ld (ix+2),a
       cp (ix+4)            ; reached mimimum yet?
       jr z,movax           ; yes - change direction.
       ret

; Move alien down.

movad  ld a,(ix+2)          ; get x coordinate.
       add a,2              ; move down.
       ld (ix+2),a          ; new coordinate.
       cp (ix+5)            ; reached maximum yet?
       jr nc,movax          ; yes - change direction.
       ret

; Change alien direction.

movax  ld a,(ix+1)          ; direction flag.
       xor 2                ; switch direction, either
                            ; horizontally or vertically.
       ld (ix+1),a          ; set new direction.
       ret
```

If we wanted to go further we might introduce an extra flag to our table, ix+6, to control the speed of the sprite, and only move it, say, every other frame if the flag is set. While simple to write and easy on memory usage, this sort of movement is rather basic and predictable and of limited use. For more complicated patrolling enemies, for example the alien attack waves in a shoot-em-up, we need tables of coordinates and while the code is again easy to write, coordinate tables quickly chew up memory especially if both x and y coordinates are stored. To access such a table we need two bytes per sprite which act as a pointer to the coordinate table.

A typical section of code could look like this:

```
       ld l,(ix+2)          ; pointer low byte, little endian.
       ld h,(ix+3)          ; pointer high byte.
       ld c,(hl)            ; put x coordinate in c.
       inc hl               ; point to y coord.
       ld b,(hl)            ; put y coordinate in b.
       inc hl               ; point to next position.
       ld (ix+2),l          ; next pointer low byte.
       ld (ix+3),h          ; next pointer high byte.
```

The slightly more complicated example below demonstrates an 8-ship attack wave using a table of vertical coordinates. The horizontal position of each sprite moves left at a constant rate of 2 pixels per frame so there's no need to bother storing it. It uses the shifter sprite routine from chapter 8 so the sprites are a little flickery, but that's not important here.

```
mloop  halt                 ; wait for TV beam.
       ld ix,entab          ; point to odd spaceships.
       call mship           ; move spaceships.
       halt
       ld ix,entab+4        ; point to even spaceships.
       call mship           ; move even spaceships.
       call gwave           ; generate fresh waves.
       jp mloop             ; back to start of loop.
```

```
; Move enemy spaceships.

mship  ld b,4              ; number to process.
mship0 push bc             ; store count.
       ld a,(ix)           ; get pointer low.
       ld l,a              ; put into l.
       ld h,(ix+1)         ; get high byte.
       or h                ; check pointer is set up.
       and a               ; is it?
       call nz,mship1      ; yes, process it then.
       ld de,8             ; skip to next-but-one entry.
       add ix,de           ; point to next enemy.
       pop bc              ; restore count.
       djnz mship0         ; repeat for all enemies.
       ret

mship1 push hl             ; store pointer to coordinate.
       call dship          ; delete this ship.
       pop hl              ; restore coordinate.
       ld a,(hl)           ; fetch next coordinate.
       inc hl              ; move pointer on.
       ld (ix),l           ; new pointer low byte.
       ld (ix+1),h         ; pointer high byte.
       ld (ix+2),a         ; set x coordinate.
       ld a,(ix+3)         ; fetch horizontal position.
       sub 2               ; move left 2 pixels.
       ld (ix+3),a         ; set new position.
       cp 240              ; reached the edge of the screen yet?
       jp c,dship          ; not at the moment, display at new position.
       xor a               ; zeroise accumulator.
       ld (ix),a           ; clear low byte of pointer.
       ld (ix+1),a         ; clear high byte of pointer.
       ld hl,numenm        ; number of enemies on screen.
       dec (hl)            ; one less with which to cope.
       ret

gwave  ld hl,shipc         ; ship counter.
       dec (hl)            ; one less.
       ld a,(hl)           ; check new value.
       cp 128              ; waiting for next attack?
       jr z,gwave2         ; attack is imminent so set it up.
       ret nc              ; yes.
       and 7               ; time to generate a new ship?
       ret nz              ; not yet it isn't.
       ld ix,entab         ; enemy table.
       ld de,4             ; size of each entry.
       ld b,8              ; number to check.
gwave0 ld a,(ix)           ; low byte of pointer.
       ld h,(ix+1)         ; high byte.
       or h                ; are they zero?
       jr z,gwave1         ; yes, this entry is empty.
       add ix,de           ; point to next ship slot.
       djnz gwave0         ; repeat until we find one.
       ret
gwave2 ld hl,wavnum        ; present wave number.
       ld a,(hl)           ; fetch current setting.
       inc a               ; next one along.
       and 3               ; start again after 4th wave.
       ld (hl),a           ; write new setting.
       ret
gwave1 ld hl,numenm        ; number of enemies on screen.
       inc (hl)            ; one more to deal with.
       ld a,(wavnum)       ; wave number.
       ld hl,wavlst        ; wave data pointers.
       rlca                ; multiple of 2.
       rlca                ; multiple of 4.
       ld e,a              ; displacement in e.
       ld d,0              ; no high byte.
       add hl,de           ; find wave address.
       ld a,(shipc)        ; ship counter.
       and 8               ; odd or even attack?
       rrca                ; make multiple of 2 accordingly.
```

```
        rrca
        ld e,a              ; displacement in e.
        ld d,0              ; no high byte.
        add hl,de           ; point to first or second half of attack.
        ld e,(hl)           ; low byte of attack pointer.
        inc hl              ; second byte.
        ld d,(hl)           ; high byte of attack pointer.
        ld (ix),e           ; low byte of pointer.
        ld (ix+1),d         ; high byte.
        ld a,(de)           ; fetch first coordinate.
        ld (ix+2),a         ; set x.
        ld (ix+3),240       ; start at right edge of screen.

; Display enemy ships.

dship   ld hl,shipg         ; sprite address.
        ld b,(ix+3)         ; y coordinate.
        ld c,(ix+2)         ; x coordinate.
        ld (xcoord),bc      ; set up sprite routine coords.
        jp sprite           ; call sprite routine.
shipc   defb 128            ; plane counter.
numenm  defb 0              ; number of enemies.


; Attack wave coordinates.
; Only the vertical coordinate is stored as the ships all move left
; 2 pixels every frame.

coord0  defb 40,40,40,40,40,40,40,40
        defb 40,40,40,40,40,40,40,40
        defb 42,44,46,48,50,52,54,56
        defb 58,60,62,64,66,68,70,72
        defb 72,72,72,72,72,72,72,72
        defb 72,72,72,72,72,72,72,72
        defb 70,68,66,64,62,60,58,56
        defb 54,52,50,48,46,44,42,40
        defb 40,40,40,40,40,40,40,40
        defb 40,40,40,40,40,40,40,40
        defb 38,36,34,32,30,28,26,24
        defb 22,20,18,16,14,12,10,8
        defb 6,4,2,0,2,4,6,8
        defb 10,12,14,16,18,20,22,24
        defb 26,28,30,32,34,36,38,40
coord1  defb 136,136,136,136,136,136,136,136
        defb 136,136,136,136,136,136,136,136
        defb 134,132,130,128,126,124,122,120
        defb 118,116,114,112,110,108,106,104
        defb 104,104,104,104,104,104,104,104
        defb 104,104,104,104,104,104,104,104
        defb 106,108,110,112,114,116,118,120
        defb 122,124,126,128,130,132,134,136
        defb 136,136,136,136,136,136,136,136
        defb 136,136,136,136,136,136,136,136
        defb 138,140,142,144,146,148,150,152
        defb 154,156,158,160,162,164,166,168
        defb 170,172,174,176,174,172,170,168
        defb 166,164,162,160,158,156,154,152
        defb 150,148,146,144,142,140,138,136


; List of attack waves.

wavlst  defw coord0,coord0,coord1,coord1
        defw coord1,coord0,coord0,coord1

wavnum  defb 0              ; current wave pointer.


; Spaceship sprite.

shipg   defb 248,252,48,24,24,48,12,96,24,48,31,243,127,247,255,247
        defb 255,247,127,247,31,243,24,48,12,96,24,48,48,24,248,252
```

```
sprit7 xor 7               ; complement last 3 bits.
       inc a               ; add one for luck!
sprit3 rl d                ; rotate left...
       rl c                ; ...into middle byte...
       rl e                ; ...and finally into left character cell.
       dec a               ; count shifts we've done.
       jr nz,sprit3        ; return until all shifts complete.

; Line of sprite image is now in e + c + d, we need it in form c + d + e.

       ld a,e              ; left edge of image is currently in e.
       ld e,d              ; put right edge there instead.
       ld d,c              ; middle bit goes in d.
       ld c,a              ; and the left edge back into c.
       jr sprit0           ; we've done the switch so transfer to screen.

sprite ld a,(xcoord)       ; draws sprite (hl).
       ld (tmp1),a         ; store vertical.
       call scadd          ; calculate screen address.
       ld a,16             ; height of sprite in pixels.
sprit1 ex af,af'           ; store loop counter.
       push de             ; store screen address.
       ld c,(hl)           ; first sprite graphic.
       inc hl              ; increment poiinter to sprite data.
       ld d,(hl)           ; next bit of sprite image.
       inc hl              ; point to next row of sprite data.
       ld (tmp0),hl        ; store in tmp0 for later.
       ld e,0              ; blank right byte for now.
       ld a,b              ; b holds y position.
       and 7               ; how are we straddling character cells?
       jr z,sprit0         ; we're not straddling them, don't bother shifting.
       cp 5                ; 5 or more right shifts needed?
       jr nc,sprit7        ; yes, shift from left as it's quicker.
       and a               ; oops, carry flag is set so clear it.
sprit2 rr c                ; rotate left byte right...
       rr d                ; ...through middle byte...
       rr e                ; ...into right byte.
       dec a               ; one less shift to do.
       jr nz,sprit2        ; return until all shifts complete.
sprit0 pop hl              ; pop screen address from stack.
       ld a,(hl)           ; what's there already.
       xor c               ; merge in image data.
       ld (hl),a           ; place onto screen.
       inc l               ; next character cell to right please.
       ld a,(hl)           ; what's there already.
       xor d               ; merge with middle bit of image.
       ld (hl),a           ; put back onto screen.
       inc hl              ; next bit of screen area.
       ld a,(hl)           ; what's already there.
       xor e               ; right edge of sprite image data.
       ld (hl),a           ; plonk it on screen.
       ld a,(tmp1)         ; temporary vertical coordinate.
       inc a               ; next line down.
       ld (tmp1),a         ; store new position.
       and 63              ; are we moving to next third of screen?
       jr z,sprit4         ; yes so find next segment.
       and 7               ; moving into character cell below?
       jr z,sprit5         ; yes, find next row.
       dec hl              ; left 2 bytes.
       dec l               ; not straddling 256-byte boundary here.
       inc h               ; next row of this character cell.
sprit6 ex de,hl            ; screen address in de.
       ld hl,(tmp0)        ; restore graphic address.
       ex af,af'           ; restore loop counter.
       dec a               ; decrement it.
       jp nz,sprit1        ; not reached bottom of sprite yet to repeat.
       ret                 ; job done.
sprit4 ld de,30            ; next segment is 30 bytes on.
       add hl,de           ; add to screen address.
       jp sprit6           ; repeat.
sprit5 ld de,63774         ; minus 1762.
```

```
       add hl,de           ; subtract 1762 from physical screen address.
       jp sprit6           ; rejoin loop.
scadd  ld a,(xcoord)       ; fetch vertical coordinate.
       ld e,a              ; store that in e.

; Find line within cell.

       and 7               ; line 0-7 within character square.
       add a,64            ; 64 * 256 = 16384 = start of screen display.
       ld d,a              ; line * 256.

; Find which third of the screen we're in.

       ld a,e              ; restore the vertical.
       and 192             ; segment 0, 1 or 2 multiplied by 64.
       rrca                ; divide this by 8.
       rrca
       rrca                ; segment 0-2 multiplied by 8.
       add a,d             ; add to d give segment start address.
       ld d,a

; Find character cell within segment.

       ld a,e              ; 8 character squares per segment.
       rlca                ; divide x by 8 and multiply by 32,
       rlca                ; net calculation: multiply by 4.
       and 224             ; mask off bits we don't want.
       ld e,a              ; vertical coordinate calculation done.

; Add the horizontal element.

       ld a,(ycoord)       ; y coordinate.
       rrca                ; only need to divide by 8.
       rrca
       rrca
       and 31              ; squares 0 - 31 across screen.
       add a,e             ; add to total so far.
       ld e,a              ; de = address of screen.
       ret

xcoord defb 0              ; display coord.
ycoord defb 0              ; display coord.
tmp0   defw 0              ; workspace.
tmp1   defb 0              ; temporary vertical position.

; Enemy spaceship table, 8 entries x 4 bytes each.

entab  defb 0,0,0,0
       defb 0,0,0,0
       defb 0,0,0,0
       defb 0,0,0,0
       defb 0,0,0,0
       defb 0,0,0,0
       defb 0,0,0,0
       defb 0,0,0,0
```

Intelligent Aliens

So far we have dealt with predictable drones, but what if we want to give the player the illusion that enemy sprites are thinking for themselves? One way we could start to do this would be to give them an entirely random decision making process.

Here is the source code for Turbomania, a game originally written for the 1K coding competition in 2005. It's very simple, but incorporates purely random movement. Enemy cars travel in a direction until they can no longer move, then select another direction at random. Additionally, a car may change direction at random even if it can continue in its present direction. It's very primitive of

course, just.take a look at the mcar routine and you'll see exactly what I mean.

```
       org 24576

; Constants.

YELLOW equ 49              ; dull yellow attribute.
YELLOB equ YELLOW + 64     ; bright yellow attribute.

; Main game code.

; Clear the screen to give green around the edges.

       ld hl,23693         ; system variable for attributes.
       ld (hl),36          ; want green background.

waitk  ld a,(23560)        ; read keyboard.
       cp 32               ; is SPACE pressed?
       jr nz,waitk         ; no, wait.
       call nexlev         ; play the game.
       jr waitk            ; SPACE to restart game.


; Clear down level data.

nexlev call 3503           ; clear the screen.
       ld hl,rmdat         ; room data.
       ld de,rmdat+1
       ld (hl),1           ; set up a shadow block.
       ld bc,16            ; length of room minus first byte.
       ldir                ; copy to rest of first row.
       ld bc,160           ; length of room minus first row.
       ld (hl),b           ; clear first byte.
       ldir                ; clear room data.

; Set up the default blocks.

       ld c,15             ; last block position.
popbl0 ld b,9              ; last row.
popbl1 call filblk         ; fill the block.
       dec b               ; one column up.
       jr z,popbl2         ; done column, move on.
       dec b               ; and again.
       jr popbl1
popbl2 dec c               ; move on row.
       jr z,popbl3         ; done column, move on.
       dec c               ; next row.
       jr popbl0

; Now draw the bits unique to this level.

popbl3 ld b,7              ; number of blocks to insert.
popbl5 push bc             ; store counter.
       call random         ; get a random number.
       and 6               ; even numbers in range 0-6 please.
       add a,2             ; make it 2-8.
       ld b,a              ; that's the column.
popbl4 call random         ; another number.
       and 14              ; even numbers 0-12 wanted.
       cp 14               ; higher than we want?
       jr nc,popbl4        ; yes, try again.
       inc a               ; place it in range 1-13.
       ld c,a              ; that's the row.
       call filblk         ; fill block.
popbl6 call random         ; another random number.
       and 14              ; only want 0-8.
       cp 9                ; above number we want?
       jr nc,popbl6        ; try again.
       inc a               ; make it 1-9.
       ld b,a              ; vertical coordinate.
```

```
        call random         ; get horizontal block.
        and 14              ; even, 0-14.
        ld c,a              ; y position.
        call filblk         ; fill in that square.
        pop bc              ; restore count.
        djnz popbl5         ; one less to do.

        xor a               ; zero.
        ld hl,playi         ; player's intended direction.
        ld (hl),a           ; default direction.
        inc hl              ; point to display direction.
        ld (hl),a           ; default direction.
        inc hl              ; player's next direction.
        ld (hl),a           ; default direction.
        out (254),a         ; set border colour while we're at it.
        call atroom         ; show current level layout.

        ld hl,168+8*256     ; coordinates.
        ld (encar2+1),hl    ; set up second car position.
        ld h,l              ; y coord at right.
        ld (encar1+1),hl    ; set up first car position.
        ld l,40             ; x at top of screen.
        ld (playx),hl       ; start player off here.
        ld hl,encar1        ; first car.
        call scar           ; show it.
        ld hl,encar2        ; second car.
        call scar           ; show it.
        call dplayr         ; show player sprite.
        call blkcar         ; make player's car black.

; Two-second delay before we start.

        ld b,100            ; delay length.
waitt   halt                ; wait for interrupt.
        djnz waitt          ; repeat.

mloop   halt                ; electron beam in top left.
        call dplayr         ; delete player.

; Make attributes blue ink again.

        call gpatts         ; get player attributes.
        defb 17,239,41      ; remove green paper, add blue paper + ink.
        call attblk         ; set road colours.

; Move player's car.

        ld a,(playd)        ; player direction.
        ld bc,(playx)       ; player coordinates.
        call movc           ; move coords.
        ld hl,(dispx)       ; new coords.
        ld (playx),hl       ; set new player position.

; Can we change direction?

        ld a,(playi)        ; player's intended direction.
        ld bc,(playx)       ; player coordinates.
        call movc           ; move coords.
        call z,setpn        ; set player's new direction.

; Change direction.

        ld a,(nplayd)       ; new player direction.
        ld (playd),a        ; set current direction.

        call dplayr         ; redisplay at new position.

; Set attributes of car.

        call blkcar         ; make player car black.

; Controls.
```

```
        ld a,239            ; keyboard row 6-0 = 61438.
        ld e,1              ; direction right.
        in a,(254)          ; read keys.
        rra                 ; player moved right?
        call nc,setpd       ; yes, set player direction.
        ld e,3              ; direction left.
        rra                 ; player moved left?
        call nc,setpd       ; yes, set player direction.
        ld a,247            ; 63486 is port for row 1-5.
        ld e,0              ; direction up.
        in a,(254)          ; read keys.
        and 2               ; check 2nd key in (2).
        call z,setpd        ; set direction.
        ld a,251            ; 64510 is port for row Q-T.
        ld e,2              ; direction down.
        in a,(254)          ; read keys.
        and e               ; check 2nd key from edge (W)..
        call z,setpd        ; set direction.

; Enemy cars.

        ld hl,encar1        ; enemy car 1.
        push hl             ; store pointer.
        call procar         ; process the car.
        pop hl              ; restore car pointer.
        call coldet         ; check for collisions.
        ld hl,encar2        ; enemy car 2.
        push hl             ; store pointer.
        halt                ; synchronise with display.
        call procar         ; process the car.
        pop hl              ; restore car pointer.
        call coldet         ; check for collisions.

; Count remaining yellow spaces.

        ld hl,22560         ; address.
        ld bc,704           ; attributes to count.
        ld a,YELLOB         ; attribute we're seeking.
        cpir                ; count characters.
        ld a,b              ; high byte of result.
        or c                ; combine with low byte.
        jp z,nexlev         ; none left, go to next level.

; End of main loop.

        jp mloop

; Black car on cyan paper.

blkcar call gpatts          ; get player attributes.
        defb 17,232,40      ; remove red paper/blue ink, add blue paper.

; Set 16x16 pixel attribute block.

attblk call attlin          ; paint horizontal line.
        call attlin         ; paint another line.
        ld a,c              ; vertical position.
        and 7               ; is it straddling cells?
        ret z               ; no, so no third line.

attlin call setatt          ; paint the road.
        call setatt         ; and again.
        ld a,b              ; horizontal position.
        and 7               ; straddling blocks?
        jr z,attln0         ; no, leave third cell as it is.
        call setatt         ; set attribute.
        dec l               ; back one cell again.
attln0 push de              ; preserve colours.
        ld de,30            ; distance to next one.
        add hl,de           ; point to next row down.
        pop de              ; restore colour masks.
        ret
```

```
; Set single cell attribute.

setatt ld a,(hl)           ; fetch attribute cell contents.
       and e               ; remove colour elements in c register.
       or d                ; add those in b to form new colour.
       ld (hl),a           ; set colour.
       inc l               ; next cell.
       ret

; Collision detection, based on coordinates.

coldet call getabc         ; get coords.
       ld a,(playx)        ; horizontal position.
       sub c               ; compare against car x.
       jr nc,coldt0        ; result was positive.
       neg                 ; it was negative, reverse sign.
coldt0 cp 16               ; within 15 pixels?
       ret nc              ; no collision.
       ld a,(playy)        ; player y.
       sub b               ; compare against car y.
       jr nc,coldt1        ; result was positive.
       neg                 ; it was negative, reverse sign.
coldt1 cp 16               ; within 15 pixels?
       ret nc              ; no collision.
       pop de              ; remove return address from stack.
       ret

setpd  ex af,af'
       ld a,e              ; direction.
       ld (playi),a        ; set intended direction.
       ex af,af'
       ret

setpn  ld a,(playi)        ; new intended direction.
       ld (nplayd),a       ; set next direction.
       ret

; Move coordinates of sprite in relevant direction.

movc   ld (dispx),bc       ; default position.
       and a               ; direction zero.
       jr z,movcu          ; move up.
       dec a               ; direction 1.
       jr z,movcr          ; move up.
       dec a               ; direction 2.
       jr z,movcd          ; move up.
movcl  dec b               ; left one pixel.
       dec b               ; left again.
movc0  call chkpix         ; check pixel attributes.
       ld (dispx),bc       ; new coords.
       ret
movcu  dec c               ; up a pixel.
       dec c               ; and again.
       jr movc0
movcr  inc b               ; right one pixel.
       inc b               ; right again.
       jr movc0
movcd  inc c               ; down a pixel.
       inc c               ; once more.
       jr movc0


; Check pixel attributes for collision.
; Any cells with green ink are solid.

chkpix call ataddp         ; get pixel attribute address.
       and 4               ; check ink colours.
       jr nz,chkpx0        ; invalid, block the move.
       inc hl              ; next square to the right.
       ld a,(hl)           ; get attributes.
       and 4               ; check ink colours.
       jr nz,chkpx0        ; invalid, block the move.
       inc hl              ; next square to the right.
```

```
        ld a,b              ; horizontal position.
        and 7               ; straddling cells?
        jr z,chkpx1         ; no, look down next.
        ld a,(hl)           ; get attributes.
        and 4               ; check ink colours.
        jr nz,chkpx0        ; invalid, block the move.
chkpx1  ld de,30            ; distance to next cell down.
        add hl,de           ; point there.
        ld a,(hl)           ; get attributes.
        and 4               ; check ink colours.
        jr nz,chkpx0        ; invalid, block the move.
        inc hl              ; next square to the right.
        ld a,(hl)           ; get attributes.
        and 4               ; check ink colours.
        jr nz,chkpx0        ; invalid, block the move.
        inc hl              ; next square to the right.
        ld a,b              ; horizontal position.
        and 7               ; straddling cells?
        jr z,chkpx2         ; no, look down next.
        ld a,(hl)           ; get attributes.
        and 4               ; check ink colours.
        jr nz,chkpx0        ; invalid, block the move.
chkpx2  ld a,c              ; distance from top of screen.
        and 7               ; are we straddling cells vertically?
        ret z               ; no, move is therefore okay.
        add hl,de           ; point there.
        ld a,(hl)           ; get attributes.
        and 4               ; check ink colours.
        jr nz,chkpx0        ; invalid, block the move.
        inc hl              ; next square to the right.
        ld a,(hl)           ; get attributes.
        and 4               ; check ink colours.
        jr nz,chkpx0        ; invalid, block the move.
        inc hl              ; next square to the right.
        ld a,b              ; horizontal position.
        and 7               ; straddling cells?
        ret z               ; no, move is fine.
        ld a,(hl)           ; get attributes.
        and 4               ; check ink colours.
        jr nz,chkpx0        ; invalid, block the move.
        ret                 ; go ahead.

chkpx0  pop de              ; remove return address from stack.
        ret

; Fill a block in the map.

; Get block address.

filblk  ld a,b              ; row number.
        rlca                ; multiply by 16.
        rlca
        rlca
        rlca
        add a,c             ; add to displacement total.
        ld e,a              ; that's displacement low.
        ld d,0              ; no high byte required.
        ld hl,rmdat         ; room data address.
        add hl,de           ; add to block address.

; Block address is in hl, let's fill it.

        ld (hl),2           ; set block on.
        ld de,16            ; distance to next block down.
        add hl,de           ; point there.
        ld a,(hl)           ; check it.
        and a               ; is it set yet?
        ret nz              ; yes, don't overwrite.
        ld (hl),1           ; set the shadow.
        ret

; Draw a screen consisting entirely of attribute blocks.
```

```
atroom ld hl,rmdat         ; room data.
       ld a,1              ; start at row 1.
       ld (dispx),a        ; set up coordinate.
       ld b,11             ; row count.
atrm0  push bc             ; store counter.
       ld b,15             ; column count.
       ld a,1              ; column number.
       ld (dispy),a        ; set to left of screen.
atrm1  push bc             ; store counter.
       ld a,(hl)           ; get next block type.
       push hl             ; store address of data.
       rlca                ; double block number.
       rlca                ; and again for multiple of 4.
       ld e,a              ; displacement to block address.
       ld d,0              ; no high byte required.
       ld hl,blkatt        ; block attributes.
       add hl,de           ; point to block we want.
       call atadd          ; get address for screen position.
       ldi                 ; transfer first block.
       ldi                 ; and the second.
       ld bc,30            ; distance to next row.
       ex de,hl            ; switch cell and screen address.
       add hl,bc           ; point to next row down.
       ex de,hl            ; switch them back.
       ldi                 ; do third cell.
       ldi                 ; fourth attribute cell.
       ld hl,dispy         ; column number.
       inc (hl)            ; move across one cell.
       inc (hl)            ; and another.
       pop hl              ; restore room address.
       pop bc              ; restore column counter.
       inc hl              ; point to next block.
       djnz atrm1          ; do rest of row.
       inc hl              ; skip one char so lines are round 16.
       ld a,(dispx)        ; vertical position.
       add a,2             ; look 2 cells down.
       ld (dispx),a        ; new row.
       pop bc              ; restore column counter.
       djnz atrm0          ; do remaining rows.
       ret

; Background block attributes.

blkatt defb YELLOB,YELLOB  ; space.
       defb YELLOB,YELLOB
       defb YELLOW,YELLOW  ; shadow space.
       defb YELLOB,YELLOB
       defb 124,68         ; black/white chequered flag pattern.
       defb 68,124

; Calculate address of attribute for character at (dispx, dispy).

atadd  push hl             ; need to preserve hl pair.
       ld hl,(dispx)       ; coords to check, in char coords.
       add hl,hl           ; multiply x and y by 8.
       add hl,hl
       add hl,hl
       ld b,h              ; copy y coord to b.
       ld c,l              ; put x coord in c.
       call ataddp         ; get pixel address.
       ex de,hl            ; put address in de.
       pop hl              ; restore hl.
       ret


; Get player attributes.

gpatts ld bc,(playx)       ; player coordinates.

; Calculate address of attribute for pixel at (c, b).

ataddp ld a,c              ; Look at the vertical first.
       rlca                ; divide by 64.
```

```
        rlca                ; quicker than 6 rrca operations.
        ld l,a              ; store in l register for now.
        and 3               ; mask to find segment.
        add a,88            ; attributes start at 88*256=22528.
        ld h,a              ; that's our high byte sorted.
        ld a,l              ; vertical/64 - same as vertical*4.
        and 224             ; want a multiple of 32.
        ld l,a              ; vertical element calculated.
        ld a,b              ; get horizontal position.
        rra                 ; divide by 8.
        rra
        rra
        and 31              ; want result in range 0-31.
        add a,l             ; add to existing low byte.
        ld l,a              ; that's the low byte done.
        ld a,(hl)           ; get cell contents.
        ret                 ; attribute address now in hl.

; Move car - change of direction required.

mcarcd ld a,(hl)            ; current direction.
       inc a                ; turn clockwise.
       and 3                ; only 4 directions.
       ld (hl),a            ; new direction.

; Move an enemy car.

mcar   push hl              ; preserve pointer to car.
       call getabc          ; fetch coordinates and direction.
       call movc            ; move the car.
       pop hl               ; refresh car pointer.
       jr nz,mcarcd         ; can't move there, turn around.
       inc hl               ; point to x.
       ld a,c               ; store x pos in c.
       ld (hl),a            ; x position.
       inc hl               ; point to y.
       ld (hl),b            ; new placing.
       or b                 ; combine the two.
       and 31               ; find position straddling cells.
       cp 8                 ; are we at a valid turning point?
       ret nz               ; no, can't change direction.
       ld a,r               ; crap random number.
       cp 23                ; check it's below this value.
       ret nc               ; it isn't, don't change.
       push hl              ; store car pointer.
       call random          ; get random number.
       pop hl               ; restore car.
       dec hl               ; back to x coordinate.
       dec hl               ; back again to direction.
       and 3                ; direction is in range 0-3.
       ld (hl),a            ; new direction.
       ret

; Fetch car coordinates and direction.

getabc ld a,(hl)            ; get direction.
       inc hl               ; point to x position.
       ld c,(hl)            ; x coordinate.
       inc hl               ; point to y.
       ld b,(hl)            ; y position.
       ret


; Process car to which hl points.

procar push hl              ; store pointer.
       push hl              ; store pointer.
       call scar            ; delete car.
       pop hl               ; restore pointer to car.
       call mcar            ; move car.
       pop hl               ; restore pointer to car.
```

```
; Show enemy car.

scar   call getabc         ; get coords and direction.
       jr dplay0

; Display player sprite.

dplayr ld bc,(playx)       ; player coords.
       ld a,(playd)        ; player direction.
dplay0 rrca                ; multiply by 32.
       rrca
       rrca
       ld e,a              ; sprite * 32 in low byte.
       ld d,0              ; no high byte.
       ld hl,cargfx        ; car graphics.
       add hl,de           ; add displacement to sprite.
       jr sprite           ; show the sprite.


; This is the sprite routine and expects coordinates in (c ,b) form,
; where c is the vertical coord from the top of the screen (0-176), and
; b is the horizontal coord from the left of the screen (0 to 240).
; Sprite data is stored as you'd expect in its unshifted form as this
; routine takes care of all the shifting itself.  This means that sprite
; handling isn't particularly fast but the graphics only take 1/8th of the
; space they would require in pre-shifted form.

; On entry HL must point to the unshifted sprite data.

sprit7 xor 7               ; complement last 3 bits.
       inc a               ; add one for luck!
sprit3 rl d                ; rotate left...
       rl c                ; ...into middle byte...
       rl e                ; ...and finally into left character cell.
       dec a               ; count shifts we've done.
       jr nz,sprit3        ; return until all shifts complete.

; Line of sprite image is now in e + c + d, we need it in form c + d + e.

       ld a,e              ; left edge of image is currently in e.
       ld e,d              ; put right edge there instead.
       ld d,c              ; middle bit goes in d.
       ld c,a              ; and the left edge back into c.
       jr sprit0           ; we've done the switch so transfer to screen.

sprite ld (dispx),bc       ; store coords in dispx for now.
       call scadd          ; calculate screen address.
       ld a,16             ; height of sprite in pixels.
sprit1 ex af,af'           ; store loop counter.
       push de             ; store screen address.
       ld c,(hl)           ; first sprite graphic.
       inc hl              ; increment poiinter to sprite data.
       ld d,(hl)           ; next bit of sprite image.
       inc hl              ; point to next row of sprite data.
       ld (sprtmp),hl      ; store it for later.
       ld e,0              ; blank right byte for now.
       ld a,b              ; b holds y position.
       and 7               ; how are we straddling character cells?
       jr z,sprit0         ; we're not straddling them, don't bother shifting.
       cp 5                ; 5 or more right shifts needed?
       jr nc,sprit7        ; yes, shift from left as it's quicker.
       and a               ; oops, carry flag is set so clear it.
sprit2 rr c                ; rotate left byte right...
       rr d                ; ...through middle byte...
       rr e                ; ...into right byte.
       dec a               ; one less shift to do.
       jr nz,sprit2        ; return until all shifts complete.
sprit0 pop hl              ; pop screen address from stack.
       ld a,(hl)           ; what's there already.
       xor c               ; merge in image data.
       ld (hl),a           ; place onto screen.
       inc l               ; next character cell to right please.
       ld a,(hl)           ; what's there already.
```

```
       xor d               ; merge with middle bit of image.
       ld (hl),a           ; put back onto screen.
       inc l               ; next bit of screen area.
       ld a,(hl)           ; what's already there.
       xor e               ; right edge of sprite image data.
       ld (hl),a           ; plonk it on screen.
       ld a,(dispx)        ; vertical coordinate.
       inc a               ; next line down.
       ld (dispx),a        ; store new position.
       and 63              ; are we moving to next third of screen?
       jr z,sprit4         ; yes so find next segment.
       and 7               ; moving into character cell below?
       jr z,sprit5         ; yes, find next row.
       dec l               ; left 2 bytes.
       dec l               ; not straddling 256-byte boundary here.
       inc h               ; next row of this character cell.
sprit6 ex de,hl            ; screen address in de.
       ld hl,(sprtmp)      ; restore graphic address.
       ex af,af'           ; restore loop counter.
       dec a               ; decrement it.
       jp nz,sprit1        ; not reached bottom of sprite yet to repeat.
       ret                 ; job done.
sprit4 ld de,30            ; next segment is 30 bytes on.
       add hl,de           ; add to screen address.
       jp sprit6           ; repeat.
sprit5 ld de,63774         ; minus 1762.
       add hl,de           ; subtract 1762 from physical screen address.
       jp sprit6           ; rejoin loop.


; This routine returns a screen address for (c, b) in de.

scadd  ld a,c              ; get vertical position.
       and 7               ; line 0-7 within character square.
       add a,64            ; 64 * 256 = 16384 (Start of screen display)
       ld d,a              ; line * 256.
       ld a,c              ; get vertical again.
       rrca                ; multiply by 32.
       rrca
       rrca
       and 24              ; high byte of segment displacement.
       add a,d             ; add to existing screen high byte.
       ld d,a              ; that's the high byte sorted.
       ld a,c              ; 8 character squares per segment.
       rlca                ; 8 pixels per cell, mulplied by 4 = 32.
       rlca                ; cell x 32 gives position within segment.
       and 224             ; make sure it's a multiple of 32.
       ld e,a              ; vertical coordinate calculation done.
       ld a,b              ; y coordinate.
       rrca                ; only need to divide by 8.
       rrca
       rrca
       and 31              ; squares 0 - 31 across screen.
       add a,e             ; add to total so far.
       ld e,a              ; hl = address of screen.
       ret

; Pseudo-random number generator.
; Steps a pointer through the ROM (held in seed), returning the contents
; of the byte at that location.

random ld hl,(seed)        ; pointer to ROM.
       res 5,h             ; stay within first 8K of ROM.
       ld a,(hl)           ; get "random" number from location.
       xor l               ; more randomness.
       inc hl              ; increment pointer.
       ld (seed),hl        ; new position.
       ret


; Sprite graphic data.
; Going up first.
```

```
cargfx defb 49,140,123,222,123,222,127,254,55,236,15,240,31,248,30,120
       defb 29,184,108,54,246,111,255,255,247,239,246,111,103,230,3,192

; Second image looks right.

       defb 60,0,126,14,126,31,61,223,11,238,127,248,252,254,217,127
       defb 217,127,252,254,127,248,11,238,61,223,126,31,126,14,60,0

; Third is pointing down.

       defb 3,192,103,230,246,111,247,239,255,255,246,111,108,54,29,184
       defb 30,120,31,248,15,240,55,236,127,254,123,222,123,222,49,140

; Last car looks left.

       defb 0,60,112,126,248,126,251,188,119,208,31,254,127,63,254,155
       defb 254,155,127,63,31,254,119,208,251,188,248,126,112,126,0,60

; Variables used by the game.

       org 32768

playi  equ $              ; intended direction when turn is possible.
playd  equ playi+1        ; player's current direction.
nplayd equ playd+1        ; next player direction.
playx  equ nplayd+1       ; player x.
playy  equ playx+1        ; player's y coordinate.

encar1 equ playy+1        ; enemy car 1.
encar2 equ encar1+3       ; enemy car 2.

dispx  equ encar2+3       ; general-use coordinates.
dispy  equ dispx+1
seed   equ dispy+1        ; random number seed.
sprtmp equ seed+2         ; sprite temporary address.
termin equ sprtmp+2       ; end of variables.

rmdat  equ 49152
```

If you have assembled this game and tried it out you will realise that it quickly becomes boring. It is very easy to stay out of the reach of enemy cars to cover one side of the track, then wait until the cars move and cover the other side. There is no hunter-killer aspect in this algorithm so the player is never chased down. What's more, this routine is so simple cars will reverse direction without warning. In most games this is only acceptable if a sprite reaches a dead end and cannot move in any other direction.

Perhaps we should instead be writing routines where aliens interact with the player, and home in on him. Well, the most basic algorithm would be something along the lines of a basic x/y coordinate check, moving an alien sprite towards the player. The routine below shows how this might be achieved, the homing routine `almov` is the one which moves the chasing sprite around. Try guiding the number 1 block around the screen with keys ASD and F, and the number 2 block will follow you around the screen. However, in doing this we soon discover the basic flaw with this type of chase - it is very easy to trap the enemy sprite in a corner because the routine isn't intelligent enough to move backwards in order to get around obstacles.

```
; Randomly cover screen with yellow blocks.

       ld de,1000         ; address in ROM.
       ld b,64            ; number of cells to colour.
yello0 push bc            ; store register.
       ld a,(de)          ; get first random coord.
       and 127            ; half height of screen.
```

```
        add a,32            ; at least 32 pixels down.
        ld c,a              ; x coord.
        inc de              ; next byte of ROM.
        ld a,(de)           ; fetch value.
        inc de              ; next byte of ROM.
        ld b,a              ; y coord.
        call ataddp         ; find attribute address.
        ld (hl),48          ; set attributes.
        pop bc              ; restore loop counter.
        djnz yello0         ; repeat a few times.


        ld ix,aldat         ; alien data.
        call dal            ; display alien.
        call dpl            ; display player.

mloop   halt                ; wait for electron beam.
        call dal            ; delete alien.
        call almov          ; alien movement.
        call dal            ; display alien.
        halt                ; wait for electron beam.
        call dpl            ; delete player.
        call plcon          ; player controls.
        call dpl            ; display the player.
        jp mloop            ; back to start of main loop.



aldat   defb 0,0,0          ; alien data.


; Display/delete alien.

dal     ld c,(ix)           ; vertical position.
        ld b,(ix+1)         ; horizontal position.
        ld (xcoord),bc      ; set sprite coordinates.
        ld hl,algfx         ; alien graphic.
        jp sprite           ; xor sprite onto screen.

; Display/delete player sprite.

dpl     ld bc,(playx)       ; coordinates.
        ld (xcoord),bc      ; set the display coordinates.
        ld hl,plgfx         ; player graphic.
        jp sprite           ; xor sprite on or off the screen.

; Player control.

plcon   ld bc,65022         ; port for keyboard row.
        in a,(c)            ; read keyboard.
        ld b,a              ; store result in b register.
        rr b                ; check outermost key.
        call nc,mpl         ; player left.
        rr b                ; check next key.
        call nc,mpr         ; player right.
        rr b                ; check next key.
        call nc,mpd         ; player down.
        rr b                ; check next key.
        call nc,mpu         ; player up.
        ret
mpl     ld hl,playy         ; coordinate.
        ld a,(hl)           ; check value.
        and a               ; at edge of screen?
        ret z               ; yes, can't move that way.
        sub 2               ; move 2 pixels.
        ld (hl),a           ; new setting.
        ret
mpr     ld hl,playy         ; coordinate.
        ld a,(hl)           ; check value.
        cp 240              ; at edge of screen?
        ret z               ; yes, can't move that way.
        add a,2             ; move 2 pixels.
        ld (hl),a           ; new setting.
```

```
        ret
mpu     ld hl,playx         ; coordinate.
        ld a,(hl)           ; check value.
        and a               ; at edge of screen?
        ret z               ; yes, can't move that way.
        sub 2               ; move 2 pixels.
        ld (hl),a           ; new setting.
        ret
mpd     ld hl,playx         ; coordinate.
        ld a,(hl)           ; check value.
        cp 176              ; at edge of screen?
        ret z               ; yes, can't move that way.
        add a,2             ; move 2 pixels.
        ld (hl),a           ; new setting.
        ret


; Alien movement routine.

almov   ld a,(playx)        ; player x coordinate.
        ld c,(ix)           ; alien x.
        ld b,(ix+1)         ; alien y.
        cp c                ; check alien x.
        jr z,alv0           ; they're equal, do horizontal.
        jr c,alu            ; alien is below, move up.
ald     inc c               ; alien is above, move down.
        jr alv0             ; now check position for walls.
alu     dec c               ; move down.
alv0    call alchk          ; check attributes.
        cp 56               ; are they okay?
        jr z,alv1           ; yes, set x coord.
        ld c,(ix)           ; restore old x coordinate.
        jr alh              ; now do horizontal.
alv1    ld (ix),c           ; new x coordinate.
alh     ld a,(playy)        ; player horizontal.
        cp b                ; check alien horizontal.
        jr z,alok           ; they're equal, check for collision.
        jr c,all            ; alien is to right, move left.
alr     inc b               ; alien is to left, move right.
        jr alok             ; check for walls.
all     dec b               ; move right.
alok    call alchk          ; check attributes.
        cp 56               ; are they okay?
        ret nz              ; no, don't set new y coord.
        ld (ix+1),b         ; set new y.
        ret


; Check attributes at alien position (c, b).

alchk   call ataddp         ; get attribute address.
        ld a,3              ; cells high.
alchk0  ex af,af'           ; store loop counter.
        ld a,(hl)           ; check cell colour.
        cp 56               ; is it black on white?
        ret nz              ; no, can't move here.
        inc hl              ; cell right.
        ld a,(hl)           ; check cell colour.
        cp 56               ; is it black on white?
        ret nz              ; no, can't move here.
        inc hl              ; cell right.
        ld a,(hl)           ; check cell colour.
        cp 56               ; is it black on white?
        ret nz              ; no, can't move here.
        ld de,30            ; distance to next cell down.
        add hl,de           ; look there.
        ex af,af'           ; height counter.
        dec a               ; one less to go.
        jr nz,alchk0        ; repeat for all rows.
        ld (ix),c           ; set new x.
        ld (ix+1),b         ; set new y.
        ret
```

```
; Calculate address of attribute for pixel at (c, b).

ataddp ld a,c              ; Look at the vertical first.
       rlca                ; divide by 64.
       rlca                ; quicker than 6 rrca operations.
       ld l,a              ; store in l register for now.
       and 3               ; mask to find segment.
       add a,88            ; attributes start at 88*256=22528.
       ld h,a              ; that's our high byte sorted.
       ld a,l              ; vertical/64 - same as vertical*4.
       and 224             ; want a multiple of 32.
       ld l,a              ; vertical element calculated.
       ld a,b              ; get horizontal position.
       rra                 ; divide by 8.
       rra
       rra
       and 31              ; want result in range 0-31.
       add a,l             ; add to existing low byte.
       ld l,a              ; that's the low byte done.
       ret                 ; attribute address now in hl.


playx  defb 80             ; player coordinates.
playy  defb 120
xcoord defb 0              ; general-purpose coordinates.
ycoord defb 0

; Shifter sprite routine.

sprit7 xor 7               ; complement last 3 bits.
       inc a               ; add one for luck!
sprit3 rl d                ; rotate left...
       rl c                ; ...into middle byte...
       rl e                ; ...and finally into left character cell.
       dec a               ; count shifts we've done.
       jr nz,sprit3        ; return until all shifts complete.

; Line of sprite image is now in e + c + d, we need it in form c + d + e.

       ld a,e              ; left edge of image is currently in e.
       ld e,d              ; put right edge there instead.
       ld d,c              ; middle bit goes in d.
       ld c,a              ; and the left edge back into c.
       jr sprit0           ; we've done the switch so transfer to screen.

sprite ld a,(xcoord)       ; draws sprite (hl).
       ld (tmp1),a         ; store vertical.
       call scadd          ; calculate screen address.
       ld a,16             ; height of sprite in pixels.
sprit1 ex af,af'           ; store loop counter.
       push de             ; store screen address.
       ld c,(hl)           ; first sprite graphic.
       inc hl              ; increment poiinter to sprite data.
       ld d,(hl)           ; next bit of sprite image.
       inc hl              ; point to next row of sprite data.
       ld (tmp0),hl        ; store in tmp0 for later.
       ld e,0              ; blank right byte for now.
       ld a,b              ; b holds y position.
       and 7               ; how are we straddling character cells?
       jr z,sprit0         ; we're not straddling them, don't bother shifting.
       cp 5                ; 5 or more right shifts needed?
       jr nc,sprit7        ; yes, shift from left as it's quicker.
       and a               ; oops, carry flag is set so clear it.
sprit2 rr c                ; rotate left byte right...
       rr d                ; ...through middle byte...
       rr e                ; ...into right byte.
       dec a               ; one less shift to do.
       jr nz,sprit2        ; return until all shifts complete.
sprit0 pop hl              ; pop screen address from stack.
       ld a,(hl)           ; what's there already.
       xor c               ; merge in image data.
       ld (hl),a           ; place onto screen.
       inc l               ; next character cell to right please.
```

```
        ld a,(hl)           ; what's there already.
        xor d               ; merge with middle bit of image.
        ld (hl),a           ; put back onto screen.
        inc hl              ; next bit of screen area.
        ld a,(hl)           ; what's already there.
        xor e               ; right edge of sprite image data.
        ld (hl),a           ; plonk it on screen.
        ld a,(tmp1)         ; temporary vertical coordinate.
        inc a               ; next line down.
        ld (tmp1),a         ; store new position.
        and 63              ; are we moving to next third of screen?
        jr z,sprit4         ; yes so find next segment.
        and 7               ; moving into character cell below?
        jr z,sprit5         ; yes, find next row.
        dec hl              ; left 2 bytes.
        dec l               ; not straddling 256-byte boundary here.
        inc h               ; next row of this character cell.
sprit6  ex de,hl            ; screen address in de.
        ld hl,(tmp0)        ; restore graphic address.
        ex af,af'           ; restore loop counter.
        dec a               ; decrement it.
        jp nz,sprit1        ; not reached bottom of sprite yet to repeat.
        ret                 ; job done.
sprit4  ld de,30            ; next segment is 30 bytes on.
        add hl,de           ; add to screen address.
        jp sprit6           ; repeat.
sprit5  ld de,63774         ; minus 1762.
        add hl,de           ; subtract 1762 from physical screen address.
        jp sprit6           ; rejoin loop.
scadd   ld a,(xcoord)       ; fetch vertical coordinate.
        ld e,a              ; store that in e.

; Find line within cell.

        and 7               ; line 0-7 within character square.
        add a,64            ; 64 * 256 = 16384 = start of screen display.
        ld d,a              ; line * 256.

; Find which third of the screen we're in.

        ld a,e              ; restore the vertical.
        and 192             ; segment 0, 1 or 2 multiplied by 64.
        rrca                ; divide this by 8.
        rrca
        rrca                ; segment 0-2 multiplied by 8.
        add a,d             ; add to d give segment start address.
        ld d,a

; Find character cell within segment.

        ld a,e              ; 8 character squares per segment.
        rlca                ; divide x by 8 and multiply by 32,
        rlca                ; net calculation: multiply by 4.
        and 224             ; mask off bits we don't want.
        ld e,a              ; vertical coordinate calculation done.

; Add the horizontal element.

        ld a,(ycoord)       ; y coordinate.
        rrca                ; only need to divide by 8.
        rrca
        rrca
        and 31              ; squares 0 - 31 across screen.
        add a,e             ; add to total so far.
        ld e,a              ; de = address of screen.
        ret

tmp0    defw 0
tmp1    defb 0


plgfx   defb 127,254,255,255,254,127,252,127,248,127,248,127,254,127,254,127
        defb 254,127,254,127,254,127,254,127,248,31,248,31,255,255,127,254
```

```
algfx  defb 127,254,254,63,248,15,240,135,227,231,231,231,255,199,255,15
       defb 252,31,248,127,241,255,227,255,224,7,224,7,255,255,127,254
```

The best alien movement routines use a combination of random elements and hunter-killer algorithms. In order to overcome the problem in the listing above we need an extra flag to indicate the enemy's present state or in this case its direction. We can move the sprite along in a certain direction until it becomes possible to switch course vertically or horizontally, whereupon a new direction is selected depending upon the player's position. However, should it not be possible to move in the desired direction we go in the opposite direction instead. Using this method a sprite can find its own way around most mazes without getting stuck too often. In fact, to absolutely guarantee that the sprite will not get stuck we can add a random element so that every so often the new direction is chosen on a random basis rather than the difference in x and y coordinates.

Cranking up the Difficulty Levels

The weighting applied to the direction-changing decision will determine the sprite's intelligence levels. If the new direction has a 90% chance of being chosen on a random basis and a 10% chance based on coordinates the alien will wander around aimlessly for a while and only home in on the player slowly. That said, a random decision can sometimes be the right one when chasing the player. An alien on a more difficult screen might have a 60% chance of choosing a new direction randomly, and a 40% chance of choosing the direction based on the player's relative position. This alien will track the player a little more closely. By tweaking these percentage levels it is possible to determine difficulty levels throughout a game and ensure a smooth transition from the simplest of starting screens to fiendishly difficult final levels.

Chapter Twelve - Timing

Timing is everything. A game is easily ruined if it runs too quickly or too slowly. Most Spectrum games will run too quickly if all they are doing is manipulating a few sprites, and need to be slowed down.

The Halt Instruction

We measure the speed of a Spectrum game by the amount of time it takes for a complete pass of the main loop, including all the jobs done by routines called within that loop. The simplest way to introduce a delay is to insert `halt` instructions to wait for an interrupt at certain points in the main loop to wait for an interrupt. As the Spectrum generates 50 interrupts per second, this means that main loops which have 1, 2 or 3 such pauses will run at 50, 25 or 17 frames per second respectively, so long as the other processing does not take up more than a frame to complete. Generally speaking, it is not a good idea to have the player sprite moving more slowly than 17 frames per second.

Actually, the halt instruction can be quite handy. In effect, it waits for the television scan line to reach the end of the screen. This means that a good time to delete, move then re-display a sprite is immediately after a halt, because the scan line won't catch up with the image, and there is no chance of flicker. If you have your game's status panel at the top of the screen, this means there is even further for the scan line to travel before it reaches the sprite area, and you can often squeeze in a couple of sprites after a halt without much danger of flickering.

The halt instruction can also be used in a loop to pause for longer periods. The following code will pause for 100 fiftieths of a second - or two seconds.

```
       ld b,100            ; time to pause.
delay  halt                ; wait for an interrupt.
       djnz delay          ; repeat.
```

The Spectrum's Clock and Vsync Routines

Unfortunately, halt is a blunt instrument. It always waits for the next interrupt, regardless of how long is left before the next one. Imagine a situation where your main loop takes 3/4 of a frame to do its processing most of the time, but every so often has periods where extra processing is involved, taking up an extra 1/2 a frame. Under these circumstances, a halt will keep the game at a constant 50 frames per second for the majority of the time, but as soon as the extra processing kicks in, the first interrupt has passed, and halt will wait for the next one, meaning that the game slows down to 25 frames per second periodically.

There is a way around this problem, and that is to count the number of frames that have elapsed since the last iteration of the main loop. On the Spectrum, the interrupt service routine in the ROM updates the Spectrum's 24-bit frames counter 50 times per second, as well as doing other things. This counter is stored in the system variables at address 23672, so by checking this location once every iteration of the loop, we can tell how many interrupts have occurred since the last time we were at the same point. Naturally, if you want to write your own interrupt routines you will either have to use `rst 56` to update the clock, or increment a frame counter yourself if you wish to use this method.

The following vsync routine is designed to stabilise a game to run at a more-or-less constant 25 frames per second:

```
wait   ld hl,pretim        ; previous time setting
       ld a,(23672)        ; current timer setting.
       sub (hl)            ; difference between the two.
       cp 2                ; have two frames elapsed yet?
       jr nc,wait0         ; yes, no more delay.
       jp wait
wait0  ld a,(23672)        ; current timer.
       ld (hl),a           ; store this setting.
       ret
pretim defb 0
```

Instead of simply sitting in a loop, you could perform some additional non-essential processing. This is a good point at which to play any beeper sound effects. A good idea is to set aside a byte to indicate the type of sound effect to play, then check that byte in your vsync routine and call the relevant sound effect routine. Your sound effect routines will also need regular checks to see if the frame counter has ticked over, and exit when it has.

There are other things you may wish to do with this spare CPU time. I sometimes cycle my sprites around the table I hold them in, changing the order in which they are displayed each loop to help prevent flickering.

Seeding Random Numbers

The Spectrum's frame counter is useful for something else: it can be used to initialise the seed for random numbers. Using the random number generator in the random numbers chapter, we can do this:

```
       ld a,(23672)        ; current timer.
       ld (seed),a         ; set first byte of random seed.
```

This is fine if we're working on genuine hardware, and will ensure a game does not start with the same sequence of random numbers every time it is played. Unfortunately, emulator authors have a nasty habit of automatically loading tape files once opened - a practice which not only makes development difficult, it results in the machine always being in the same state every time a particular game is loaded, meaning random numbers can follow the same sequence every time that game is played. The solution for the games programmer is to wait for a debounced keypress as soon as our game has loaded, after which we can set our seed. This introduces a human element and ensures the random number generator is different every time.

Chapter Thirteen - Double Buffering

Until now we have drawn all our graphics directly onto the screen, for reasons of speed and simplicity. However, there is one major disadvantage to this method: if the television scan line happens to be covering the particular screen area where we are deleting or redrawing our image then our graphics will appear to flicker. Unfortunately, on the Spectrum there is no easy way to tell where the scan line is at any given point so we have to find a way around this. One method which works well is to delete and redraw all sprites immediately following a halt instruction, before the scan has a chance to catch up with the image being drawn. The disadvantage to this method is that our sprite code has to be pretty fast, and even then it is not advisable to delete and re-draw more than two sprites per frame because by then the scan will be over the top border and into the screen area. Of course, locating the status panel at the top of the screen might give a little more time to draw our graphics, and if the game is to run at 25 frames per second we could employ a second halt instruction and manoeuvre another couple of sprites immediately afterwards. Ultimately, there comes a point where this breaks down. If our graphics are going to take a little longer to draw we need another way to hide the process from the player and we need to employ the use of a second buffer screen. This means that all the work involved in drawing and undrawing graphics is hidden from the player and all that is visible is each finished frame once it has been drawn.

There are two ways of doing this on a Spectrum. One method will only work on a 128K machine, so we will put that to one side for the time being. The other method actually tends to be more complicated in practice but will work on any Spectrum.

Creating a Screen Buffer

The simplest way to implement double buffering on a 48K Spectrum is to set up a dummy screen elsewhere in RAM, and draw all our background graphics and sprites there. As soon as our screen is complete we copy this dummy screen to the physical screen at address 16384 thus:

```
.
; code to draw all our sprites etc.
.
.
.
.

; now screen is drawn copy it to physical screen.

       ld hl,49152
       ld de,16384
       ld bc,6912
       ldir
```

While in theory this is perfect, in practice copying 6912 bytes of RAM (or 6144 bytes if we ignore the colour attributes) to the screen display every frame it is too slow for arcade games. The secret is to reduce the amount of screen RAM we need to copy each frame, and to find a faster way than by transferring it with the LDIR instruction.

The first way is to decide how big our screen is going to be. Most games separate the screen into 2 areas: a status panel to display score, lives and other bits of information, and a window where all the action takes place. As we don't need to update the status panel every frame our dummy screen only

needs to be as big as the action window. So if we were to have a status panel as an 80 x 192 pixel at the right edge of the screen that would leave us a 176x192 pixel window, meaning our dummy screen would only need to be 22 chars wide by 192 pixels high, or 22x192=4224 bytes. Manually moving 4224 bytes from one part of RAM to another is far less painful than manipulating 6114 bytes. The trick is to find a size which is large enough not to restrict gameplay while being small enough to be manipulated quickly. Of course, we may also want to make our buffer a little larger around the edges. While these edges are not displayed on the screen they are useful if we wish to clip sprites as they move into the action window from the sides.

Once we have set our buffer size in stone we need to write a routine to transfer it to the physical display file one or two bytes at a time. While we are at it, we can also re-order our buffer screen to use a more logical display method than the one used by the physical screen. We can make allowances for the peculiar ordering of the Spectrum's display file in our transfer rountine, meaning any graphics routines which make use of our dummy screen buffer can be simplified.

There are two really quick ways of moving a dummy screen to the display screen. The first, and most simple method, is to use lots of unrolled LDI instructions. The second, and more complicated method, makes use of PUSH and POP to transfer the data.

Let us start with LDI. If our buffer is 22 chars wide we might transfer a single line from the buffer to the screen display with 22 consecutive LDI instructions - it is much quicker to use lots of LDI instructions than to use a single LDIR. We could write a routine to transfer our data across a single line at a time, pointing HL to the start of each line of the buffer, DE to the line on the screen where it is needed, and then 22 LDI instructions to move the data across. However, as each LDI instruction takes two bytes of code, it stands to reason that such a routine would be at least twice the size of the buffer it moved. A considerable hit when dealing with a little over 40K of useful RAM. You may instead wish to move the LDI instructions to a subroutine which copies a pixel line, or perhaps a group of 8 pixel lines, at a time. This routine could then be called from within a loop - unrolled or not - which could take care of the HL and DE registers.

The second method is to transfer the buffer to the screen using PUSH and POP instructions. While this does have the advantage of being the fastest way there is, there are drawbacks. You do need complete control of the stack pointer so *you can't have any interrupts occurring mid-way through the routine*. The stack pointer must be stored away somewhere first, and restored immediately afterwards.

The Spectrum's stack is usually located below your program code, but this method involves setting the stack to point to each part of the buffer in turn, and then using POP to copy the contents of the dummy screen buffer into each of the register pairs in turn. The stack pointer is then moved to the relevant point in the screen display RAM, before the registers are PUSHED into memory in the reverse order to that in which they were POPPED. Ie, values are POPPED from the buffer going from the start of each line, and PUSHED to the screen in the reverse order, going from the end of the line to the beginning.

Below is the gist of the screen transfer routine from Rallybug. This used a buffer 30 characters wide, with 28 characters visible on screen. The remaining 2 characters were not displayed so that sprites moved onto the screen slowly from the edge, rather than suddenly appearing from nowhere. As the visible screen width is 28 characters wide, this requires 14 16-bit registers per line. Obviously, the Z80A doesn't have this many, even counting the alternate registers and IX and IY. As such, the Rallybug routine splits the display into two halves of 14 bytes each, requiring just 7 register pairs. The routine sets the stack pointer to the beginning of each buffer line in turn, then

POPs the data into AF, BC, DE and HL. It then swaps these registers into the alternate register set with EXX, and POPs 6 more bytes into BC, DE and HL. These registers now need to be unloaded into the screen area, so the stack pointer is set to point to the end of the relevant screen line, and HL, DE and BC are PUSHed into position. The alternate registers are then restored, and HL, DE, BC and AF are respectively copied into position. This is repeated over and over again for each half of each screen line, before the stack pointer is restored to its original position.

Complicated, yes. But incredibly fast.

```
SEG1   equ 16514
SEG2   equ 18434
SEG3   equ 20482

P0     equ 0
P1     equ 256
P2     equ 512
P3     equ 768
P4     equ 1024
P5     equ 1280
P6     equ 1536
P7     equ 1792

C0     equ 0
C1     equ 32
C2     equ 64
C3     equ 96
C4     equ 128
C5     equ 160
C6     equ 192
C7     equ 224


xfer   ld (stptr),sp       ; store stack pointer.

; Character line 0.

       ld sp,WINDOW        ; start of buffer line.
       pop af
       pop bc
       pop de
       pop hl
       exx
       pop bc
       pop de
       pop hl
       ld sp,SEG1+C0+P0+14 ; end of screen line.
       push hl
       push de
       push bc
       exx
       push hl
       push de
       push bc
       push af

       .
       .

       ld sp,WINDOW+4784   ; start of buffer line.
       pop af
       pop bc
       pop de
       pop hl
       exx
       pop bc
       pop de
       pop hl
       ld sp,SEG3+C7+P7+28 ; end of screen line.
```

```
        push hl
        push de
        push bc
        exx
        push hl
        push de
        push bc
        push af

okay    ld sp,(stptr)       ; restore stack pointer.
        ret
```

Scrolling the Buffer

Now we have our dummy screen, we can do anything we like to it without the risk of flicker or other graphical anomalies, because we only transfer the buffer to the physical screen when we have finished building the picture. We can place sprites, masked or otherwise, anywhere we like and in any order we like. We can move the screen around, and animate the background graphics, and most importantly, we can now scroll in any direction.

Different techniques are required for different types of scrolling, although they all have one thing in common: as scrolling is a processor-intensive task, unrolled loops are the order of the day. The simplest type of scroll is a left/right single pixel scroll. A right single pixel scroll requires us to set the hl register pair to the start of the buffer, then execute the following two operands over and over again until we reach the end of the buffer:

```
        rr (hl)             ; rotate carry flag and 8 bits right.
        inc hl              ; next buffer address.
```

Similarly, to execute a left single-pixel scroll we set hl to the last byte of the buffer and execute these two instructions until we reach the beginning of the buffer:

```
        rl (hl)             ; rotate carry flag and 8 bits right.
        dec hl              ; next buffer address.
```

For most of the time, however, we can get away with only incrementing or decrementing the l register, instead of the hl pair, speeding up the routine even more. This does have the drawback of having to know exactly when the high order byte of the address changes. For this reason, I usually set my buffer address in stone right at the beginning of the project, often at the very top of RAM, so I don't have to rewrite the scrolling routines when things get shifted around during the course of a project. As with the routine to transfer the buffer to the physical screen, a massive unrolled loop is very expensive in terms of RAM, so it is a good idea to write a smaller unrolled loop which scrolls, say, 256 bytes at a time, then call it 20 or so times, depending upon the chosen buffer size.

In addition to scrolling one pixel at a time, we can scroll four pixels fairly quickly too. By replacing `rl (hl)` with `rld` in the left scroll, and `rr (hl)` with `rrd` in the right scroll, we can move 4-pixels.

Vertical scrolling is done by shifting bytes around in RAM, in much the same way as the routine to transfer the dummy screen to the physical one. To scroll up one pixel, we set our FROM address to be the start of the second pixel line, the TO address to the address of the start of the buffer, then copy the data from the FROM address to the TO address until we reach the end of the buffer. To scroll down, we have to work in the opposite direction, so we set our FROM address to the end of the penultimate line of the buffer, our TO address to the end of the last line, and work backwards until we reach the start of the buffer. The added advantage of vertical scrolling is that we can scroll

up or down by more than one line, simply by altering the addresses, and the routine will run just as quickly. Generally speaking, it isn't a good idea to scroll by more than one pixel if your frame rate is lower than 25 frames per second, because the screen will appear to judder.

There is one other technique that can be employed with vertical scrolling, and it is one I employed when writing Megablast for Your Sinclair. This involves treating the dummy screen as wrap-around. In other words, you still use the same amount of RAM for the dummy buffer, but the part of the buffer from which you start copying to the top of the screen can change from one frame to the next. When you reach the end of the buffer, you skip back to the beginning. With this system, the routine to copy the buffer takes the address of the start of the buffer from a 16-bit pointer which could point to any line in the buffer, and copies the data to the physical screen line by line until it reaches the end of the buffer. At this point, the routine copies the data from the start of the buffer to the remainder of the physical screen. This makes the transfer routine a little slower, and complicates any other graphics routines - which also have to go back to the first line whenever they go beyond the last line in the buffer. It does, on the other hand, mean that no data needs to be shifted in order to scroll the screen. By changing the 16-bit pointer to the line which is first copied to the physical screen, scrolling is done automatically when the buffer is transferred.

Chapter Fourteen – More Sophisticated Movement

So far, we have moved sprites up, down, left and right by whole pixels. However, many games require more sophisticated sprite manipulation. Platform games require gravity, top-down racing games use rotational movement and others use inertia.

Jump and Inertia Tables

The simplest way of achieving gravity or inertia is to have a table of values. For example, the Egghead games make use of a jump table and maintain a pointer to the current position. Such a table might look like the one below.

```
; Jump table.
; Values >128 are going up, <128 are going down.
; When we hit value 128 we have reached maximum velocity.

jptr   defw jtabu
jtabu  defb 250,251,252
       defb 253,254,254
       defb 255,255,255
       defb 0,255,0
jtabd  defb 1,0,1,1,1,2
       defb 2,3,4,5,6,6
       defb 6,6,128
```

With the pointer stored in `jptr`, we might do something like this:

```
       ld hl,(jptr)        ; fetch jump pointer.
       ld a,(hl)           ; next value.
       cp 128              ; reached end of table?
       jr nz,skip          ; no, we're okay.
       dec hl              ; back to maximum velocity.
       ld a,(hl)           ; fetch max speed.
skip   inc hl              ; move pointer along.
       ld (jptr),hl        ; set next pointer position.
       ld hl,verpos        ; player's vertical position.
       add a,(hl)          ; add relevant amount.
       ld (hl),a           ; set player's new position.
```

To initiate a jump, we would set `jptr` to `jtabu`. To start falling, we would set it to `jtabd`.

Okay, so it's a bit simplistic. In practice, we would usually use the value from the jump table as a loop counter, moving the player up or down one pixel at a time, checking for collisions with platforms, walls, deadly items etc as we go. We might also use the end marker (128) to signify that the player had fallen too far, and set a flag so that the next time the player hits something solid, he loses a life. That said, you get the picture.

Fractional Coordinates

If we want more sophisticated gravity, inertia, or rotational movement we need fractional coordinates. Up until now, with the Spectrum's resolution at 256x192 pixels, we have only needed to use one byte per coordinate. If instead we use a two-byte register pair, the high byte for the integer and low byte for the fraction, we open up a whole new world of possibilities. This gives us 8 binary decimal places, allowing very precise and subtle movements. With a coordinate in the hl pair, we can set up the displacement in de, and add the two together. When plotting our sprites, we

simply use the high bytes as our x and y coordinates for our screen address calculation, and discard the low bytes which hold the fractions. The effect of adding a fraction to a coordinate will not be visible every frame, but even the smallest fraction, 1/256, will slowly move a sprite over time.

Now we can take a look at gravity. This is a constant force, in practice it accelerates an object towards the ground at 9.8m/s^2. To simulate it in a Spectrum game, we set up our vertical coordinate as a 16-bit word. We then set up a second 16-bit word for our momentum. Each frame, we add a tiny fraction to the momentum, then add the momentum to the vertical position. For example:

```
        ld hl,(vermom)      ; momentum.
        ld de,2             ; tiny fraction, 1/128.
        add hl,de           ; increase momentum.
        ld (vermom),hl      ; store momentum.
        ld de,(verpos)      ; vertical position.
        add hl,de           ; add momentum.
        ld (verpos),hl      ; store new position.
        ret
verpos defw 0               ; vertical position.
vermom defw 0               ; vertical momentum.
```

Then, to plot our sprites, we simply take the high byte of our vertical position, `verpos+1`, to give us the number of pixels from the top of the screen. Different values of de will vary the strength of the gravity, indeed we can even swap the direction by subtracting de from hl, or by adding a negative distance (65536-distance). We can apply the same to the y coordinate too, and have the sprite subject to momentum in all directions. This is how we would go about writing a Thrust-style game.

Rotational Movement

The other thing we might need for a Thrust game, top-down racers, or anything where circles or basic trigonometry is involved is a sine/cosine table. Mathematics isn't everybody's cup of tea, and if your trigonometry is a little rusty I suggest you read up on sines and cosines before continuing with the remainder of this chapter.

In mathematics, we can find the x and y distance from the centre of a circle given the radius and the angle by using sines and cosines. However, whereas in maths a circle is made up of either 360 degrees or 2 PI radians, it is more convenient for the Spectrum programmer to represent his angle as, say, an 8-bit value from 0 to 255, or even use fewer bits, depending on the number of positions the player sprite can take. He can then use this value to look up his 16-bit fractional values for the sine and cosine in a table. Assuming we have an 8-bit angle set up in the accumulator, and we wish to find the sine, we simply access the table in a manner similar to this:

```
        ld de,2             ; tiny fraction - 1/128.
        ld l,a              ; angle in low byte.
        ld h,0              ; zero displacement high byte.
        add hl,hl           ; double displacement as entries are 16-bit.
        ld de,sintab        ; address of sine table.
        add hl,de           ; add displacement for this angle.
        ld e,(hl)           ; fraction of sine.
        inc hl              ; point to second half.
        ld d,(hl)           ; integer part.
        ret                 ; return with sine in de.
```

Sinclair BASIC actually provides us with the values we require, with its SIN and COS functions. Using this, we can POKE the values returned into RAM and either save to tape, or save out the binary using an emulator such as SPIN. Alternatively, you may prefer to use another programming

language on the PC to generate a table of formatted sine values. to import into your source file, or include as a binary. For a sine table with 256 equally-spaced angles, we would need a total of 512 bytes, but we would need to be careful to convert the number returned by SIN into one our game will recognise. Multiplying the sine by 256 will give us our positive values, but where SIN returns a negative result, we might need to multiply the ABS value of the sine by 256, then either subtract that from 65536 or set bit d7 of the high byte to indicate that the number must be subtracted rather than added to our coordinate. With a sine table constructed in this manner, we don't need a separate table for cosines, as we just add or subtract 64, or a quarter-turn, to the angle before looking up the value in our table. To move a sprite at an angle of A, we add the sine of A to one coordinate, and the cosine of A to the other coordinate. By changing whether we add or subtract a quarter turn to obtain the cosine, and which plane uses sines and which uses cosines, we can start our circle at any of the 4 main compass points, and make it go in a clockwise or anti-clockwise direction.

Chapter Fifteen - Mathematics

Adding and subtracting is straightforward enough on the Spectrum's CPU, we have an abundance of instructions to perform these tasks. But unlike some later processors in the series, the programmer of the Z80A has to do his own multiplication and division. While such calculations are rare, they have their uses in certain types of game, and until you have routines to do the job, certain things are very tricky to do. For example, without Pythagoras' theorem, it can be difficult to program an enemy sprite to shoot at the player with any degree of accuracy.

Suppose sprite A needs to fire a shot at sprite B. We need to find the angle at which sprite A is to fire, and some trigonometry is necessary to do this. We know the coordinates of the sprite, Ax, Ay, Bx and By, and the distances between these, Bx-Ax and By-Ay, will give us the opposite and adjacent line lengths. Unfortunately, the only way to calculate the angle from the opposite and adjacent is to use arctangent, and as tangents are only suitable for certain angles, we are better off using sine or cosine instead. So in order to find the angle from sprite A to sprite B, we need to find the length of the hypotenuse.

The hypotenuse is calculated by squaring the x distance, adding it to the square of the y distance, then finding the square root. There are routines in the Sinclair ROM to do all of this, but there is one serious drawback: as anyone who has ever used Sinclair BASIC will tell you, the maths routines are incredibly slow for writing games. So we have to knuckle down and write our own.

Squaring our x and y distances means using a multiplication routine and multiplying the numbers by themselves. Thankfully, this part is relatively painless. Multiplication is achieved in the same way as you would perform long multiplication on paper, although this time we are working in binary. All that is required is shifting, testing bits, and adding. Where a bit exists in our first factor, we add the second factor to the total. Then we shift the second factor left, and test the next bit along in our first factor. The routine below, taken from Kuiper Pursuit, demonstrates thew technique by multiplying h by d and returning the result in hl.

```
imul   ld e,d              ; HL = H * D
       ld a,h              ; make accumulator first multiplier.
       ld hl,0             ; zeroise total.
       ld d,h              ; zeroise high byte so de=multiplier.
       ld b,8              ; repeat 8 times.
imul1  rra                 ; rotate rightmost bit into carry.
       jr nc,imul2         ; wasn't set.
       add hl,de           ; bit was set, so add de.
       and a               ; reset carry.
imul2  rl e                ; shift de 1 bit left.
       rl d
       djnz imul1          ; repeat 8 times.
       ret
```

Now we need a square root, which is where our problems begin. Square roots are a lot more complicated. This means doing a lot of divisions, so first we need a division routine. This can be seen to work in the opposite way to multiplication, by shifting and subtracting. The next routine, also from Kuiper Pursuit, divides hl by d and returns the result in hl.

```
idiv   ld b,8              ; bits to check.
       ld a,d              ; number by which to divide.
idiv3  rla                 ; check leftmost bit.
       jr c,idiv2          ; no more shifts required.
       inc b               ; extra shift needed.
       cp h
       jr nc,idiv2
```

```
       jp idiv3            ; repeat.

idiv2  xor a
       ld e,a
       ld c,a              ; result.
idiv1  sbc hl,de           ; do subtraction.
       jr nc,idiv0         ; no carry, keep the result.
       add hl,de           ; restore original value of hl.
idiv0  ccf                 ; reverse carry bit.
       rl c                ; rotate in to ac.
       rla
       rr d                ; divide de by 2.
       rr e
       djnz idiv1          ; repeat.
       ld h,a              ; copy result to hl.
       ld l,c
       ret
```

In the same way that multiplication is made up of shifting and adding, and division is done via shifting and subtracting, so square roots can be calculated by shifting and dividing. We're simply trying to find the "best fit" number which, when multiplied by itself, gives us the number with which we started. I won't go into detailed explanation as to how the following routine works - if you really are *that* interested, follow my comments and step it through a debugger. Taken from Blizzard's Rift, it returns the square root of hl in the accumulator.

```
isqr   ld (sqbuf0),hl      ; number for which we want to find square root.
       xor a               ; zeroise accumulator.
       ld (sqbuf2),a       ; result buffer.
       ld a,128            ; start division with highest bit.
       ld (sqbuf1),a       ; next divisor.
       ld b,8              ; 8 bits to divide.
isqr1  push bc             ; store loop counter.
       ld a,(sqbuf2)       ; current result.
       ld d,a
       ld a,(sqbuf1)       ; next bit to check.
       or d                ; combine with divisor.
       ld d,a              ; store low byte.
       xor a               ; HL = HL / D
       ld c,a              ; zeroise c.
       ld e,a              ; zeroise e.
       push de             ; remember divisor.
       ld hl,(sqbuf0)      ; original number.
       call idiv4          ; divide number by d.
       pop de              ; restore divisor.
       cp d                ; is divisor greater than result?
       jr c,isqr0          ; yes, don't store this bit then.
       ld a,d
       ld (sqbuf2),a       ; store new divisor.
isqr0  ld hl,sqbuf1        ; bit we tested.
       and a               ; clear carry flag.
       rr (hl)             ; next bit to right.
       pop bc              ; restore loop counter.
       djnz isqr1          ; repeat
       ld a,(sqbuf2)       ; return result in hl.
       ret
sqbuf0 defw 0
sqbuf1 defb 0
sqbuf2 defb 0
```

With the length of the hypotenuse calculated, we can simply divide the opposite line by the hypotenuse to find the cosine of the angle. A quick search of our sine table will then tell us what that angle is. Phew!

This is the entire calculation taken from Blizzard's Rift. Note that it uses the adjacent line length rather than the opposite, so finds the arccosine instead of the arcsine. It is also only used when the ship is above the gun turret, giving the player the opportunity to sneak up and attack from underneath. Nevertheless, it demonstrates how a sprite can fire at another with *deadly* accuracy. If you have ever played Blizzard's Rift, you will know exactly how lethal those gun turrets can be.

```
; Ship is above the gun so we can employ some basic trigonometry to aim it.
; We need to find the angle and to do this we divide the adjacent by
; the hypotenuse and find the arccosine.
; First of all we put the length of the opposite on the stack:

mgunx  ld a,(nshipy)       ; ship y coordinate.
       ld hl,guny          ; gun y coord.
       sub (hl)            ; find difference.
       jr nc,mgun0         ; result was positive.
       neg                 ; negative, make it positive.
mgun0  cp 5                ; y difference less than 5?
       jr c,mgunu          ; yes, point straight up.
       push af             ; place length of opposite on stack.

; Next we require the length of the hypotenuse and we can use good
; old Pythagoras' theorem for this.

       ld h,a              ; copy a to h.
       ld d,h              ; copy h to d.
       call imul           ; multiply integer parts to get 16-bit result.
       push hl             ; remember squared value.

       ld hl,nshipx        ; gun x coordinate.
       ld a,(gunx)         ; ship x coordinate.
       sub (hl)            ; find difference, will always be positive.
       ld h,a              ; put x difference in h.
       ld d,h              ; copy h to d.
       call imul           ; multiply h by d to get square.

       pop de              ; get last squared result.
       add hl,de           ; want the sum of the two.
       call isqr           ; find the square root, hypotenuse in a.
       pop de              ; opposite line now in d register.

       ld h,a              ; length of hypotenuse.
       ld l,0              ; no fraction or sign.
       ex de,hl            ; switch 'em.

; Opposite and hypotenuse are now in de and hl.
; We now divide the first by the second and find the arcsine.
; Remember - sine = opposite over hypotenuse.

       call div            ; division will give us the sine.
       ex de,hl            ; want result in de.
       call asn            ; get arcsine to find the angle.
       push af

; Okay, we have the angle but it's only 0 to half-pi radians (64 angles)
; so we need to make an adjustment based upon the quarter of the circle.
; We can establish which quarter of the circle our angle lies in by
; examining the differences between the ship and gun coordinates.

       ld a,(guny)         ; gun y position.
       ld hl,shipy         ; ship y.
       cp (hl)             ; is ship to the right?
       jr nc,mgun2         ; player to the left, angle in second quarter.

; Angle to play is in first quarter, so it needs subtracting from 64.

       ld a,64             ; pi/2 radians = 64 angles.
```

```
        pop bc              ; angle in b.
        sub b               ; do the subtraction.
        ld (ix+1),a         ; new angle.
        ret                 ; we have our angle.

; Second quarter - add literal 64 to our angle.

mgun2   pop af              ; original angle.
        add a,192           ; add pi/2 radians.
        ld (ix+1),a         ; new angle.
        ret                 ; job's a good 'un!
```

Chapter Sixteen - Music and AY Effects

The AY-3-8912

Introduced with the 128K models and pretty much standard since then, this is a very popular sound chip, used on various other computers, not to mention video games and pinball machines. Basically, it has 14 registers, which can be written to, or read from, via in and out instructions.

The first six registers control the tone for each of the three channels, and are paired in the little-endian way we would expect, ie register 0 is Channel A tone low, register 1 is channel A tone high, register 2 is channel B tone low, and so on. Register 6 controls the white noise period, values of 0 to 31 are valid. 0 gives the highest frequency noise, 31 the lowest. Register 7 is the mixer control. Bits d0-d5 select white noise, tone, neither, or both. To enable tone or white noise, a bit must be reset, so 0 outputs tone and noise from all three channels, 63 outputs nothing. Registers 8, 9 and 10 are envelope/amplitude controls. A value of 16 tells the chip to use the envelope generator, 0-15 will set the volume for that channel directly. In practice, you are better off controlling the volume yourself, along with the tone. By varying these from one frame to the next, it is possible to produce a variety of very good sound effects. Should you wish to use the envelope generator, the next two registers, 11 and 12, are paired to form the 16-bit period of the envelope, and register 13 determines the pattern. The 128K manual explains the full list of patterns, but I won't cover them here as I have never found them particularly useful myself.

To read a register, we write the number of the register to port 65533, then immediately read that port. To write to a register, we again send the number if the register to port 65533, and then the value to 49149. To the uninitiated, Z80 opcodes don't appear to be capable of writing to 16-bit port addresses. Don't let that confuse you, it's just that the way they are written is misleading. `out (c),a` actually means `out (bc),a` and `out (n),a` actually does `out (a*256+n),a`.

Reading a sound chip register does have some uses. You might want to read the volume registers 8, 9 and 10 and display volume bars - I did something along those lines in Egghead 5. Also, believe it or not, the Sinclair light gun is read via sound chip register 14. Yes, really. It only actually yields two pieces of information, whether or not the trigger is pressed, and whether or not the gun is pointed at a bright part of the screen, or indeed any bright object. It is up to the programmer what he does with that information.

Here's some rather basic code to write to the sound chip:

```
; Write the contents of our AY buffer to the AY registers.

w8912  ld hl,snddat        ; start of AY-3-8912 register data.
       ld e,0              ; start with register 0.
       ld d,14             ; 14 to write.
       ld c,253            ; low byte of port to write.
w8912a ld b,255            ; 255*256+253 = port 65533 = select soundchip
register.
       out (c),e           ; tell chip which register we're writing.
       ld a,(hl)           ; value to write.
       ld b,191            ; 191*256+253 = port 49149 = write value to register.
       out (c),a           ; this is what we're putting there.
       inc e               ; next sound chip register.
       inc hl              ; next byte to write.
       dec d               ; decrement loop counter.
       jp nz,w8912a        ; repeat until done.
       ret
```

```
snddat defw 0              ; tone registers, channel A.
       defw 0              ; channel B tone registers.
       defw 0              ; as above, channel C.
sndwnp defb 0              ; white noise period.
sndmix defb 60             ; tone/noise mixer control.
sndv1  defb 0              ; channel A amplitude/envelope generator.
sndv2  defb 0              ; channel B amplitude/envelope.
sndv3  defb 0              ; channel C amplitude/envelope.
sndenv defw 600            ; duration of each note.
       defb 0
```

By calling `w8912` once every iteration of the main loop, the sound is constantly updated. It is then up to you to update the buffer as each noise changes from one frame to the next. Think of it as "animating" sound. However, just because you stop updating the sound registers the sound won't stop playing. The AY chip will keep playing your tone or noise until instructed to stop. A quick way to do this is to set the three amplitude registers to 0. In the example above, write a zero to `sndv1`, `sndv2` and `sndv3` then call `w8912`.

Using Music Drivers

Most 128K music drivers, and some 48K ones, have two entry points. An initialisation/termination routine which stops all sound and resets the driver to the beginning of the tune, and a service routine to be called repeatedly, usually 50 times per second. A good place to store music is usually 49152, the start of the switchable RAM bank area. If you know the start address of the driver, or are in a position to determine this yourself, the very beginning is usually the initialisation address which sets up your music at the start. More often than not, the code at this point either simply jumps to another address, or loads a register or register pair before jumping elsewhere. The service routine tends to immediately follow this jp instruction. If your driver has no other documentation, you may have to disassemble the code to find this address, usually 3-6 bytes in.

To use a music driver, call the initialisation address prior to starting, and also when you want to turn it off. Between these points, you need to call the service routine repeatedly. This can either be done manually, or by setting up an interrupt to do the job automatically. If you choose to do this manually, for example in menu code, bear in mind that clearing the screen and displaying a menu, high score table, instructions etc will take more than 1/50th of a second to do, so this will delay your routine and could sound odd. It might be better to write a routine to clear the screen over several frames with some sort of special effect, punctuated with halts and calls to the service routine every frame.

Chapter Seventeen - Interrupts

Setting up your own interrupts can a nightmare the first time you try it, as it is a complicated business. With practice, it becomes a little easier. To make the Spectrum run our own interrupt routine, we have to tell it where the routine is, put the machine into interrupt mode 2, and ensure that interrupts are enabled. Sound simple enough? The tricky part is telling the Spectrum where our routine is located.

With the machine in mode 2, the Z80 uses the i register to determine the high byte of the address of the pointer to the interrupt service routine address. The low byte is supplied by the hardware. In practice, we never know what the low byte is going to be - so you see the problem? The low byte could be 0, it could be 255, or it could be anywhere in between. This means we need a whole block of 257 bytes consisting of pointers to the start address of our service routine. As the low byte supplied by the hardware could be odd or even, we have to make sure that the low byte and the high byte of the address of our service routine are identical. This seriously restricts where we can locate our routine. We should also only locate our table of pointers and our routine in uncontended RAM. Do not place them below address 32768. Even paging in an uncontended RAM bank for the purpose, such as bank 1, will produce problems on certain models of Spectrum. Personally, I find bank 0 to be as good a place as any.

Let us say we choose address 51400 as the location of our interrupt routine. This is valid as both the high byte and low byte are 200, since 200*256+200 = 51400. We then need a table of 129 pointers all pointing to this address, or 257 instances of defb 200, located at the start of a 256-byte page boundary. Assuming we put it high up out of the way, we could start it at 254*256 = 65024.

We would do this:

```
        org 51400

int     ; interrupt service routine.


        org 65024

        ; pointers to interrupt routine.

        defb 200,200,200,200
        defb 200,200,200,200
        .
        .
        defb 200,200,200,200
        defb 200
```

Ugh! Still, now we come to our interrupt routine. Interrupts can occur during any period, so we have to preserve any registers we are likely to use, perform our code, optionally call the ROM service routine, restore the registers, re-enable interrupts, then return from the interrupt with a reti. Our routine might resemble this:

```
int     push af             ; preserve registers.
        push bc
        push hl
        push de
        push ix
        call 49158          ; play music.
        rst 56              ; ROM routine, read keys and update clock.
        pop ix              ; restore registers.
        pop de
        pop hl
```

```
pop bc
pop af
ei                  ; always re-enable interrupts before returning.
reti                ; done.
ret
```

If you are not reading the keyboard via the system variables you may wish to dispense with the `rst 56`. Doing so will free up the iy registers. However, if your game's timing counts the frames using the method described in the timing chapter, you will need to increment the timer yourself:

```
ld hl,23672         ; frames counter.
inc (hl)            ; move it along.
```

With all this in place, we are ready to set off our interrupts. We have to point the i register at the table of pointers and select interrupt mode 2. This code will do the job for us:

```
di                  ; interrupts off as a precaution.
ld a,200            ; high byte of pointer table location.
ld i,a              ; set high byte.
im2                 ; select interrupt mode 2.
ei                  ; enable interrupts.
```

Chapter Eighteen - Making Games Load and Run Automatically

While this is simple enough to achieve for an experienced Sinclair BASIC programmer, it is an area often overlooked. In particular, programmers migrating to the Spectrum from other machines will not be familiar with the way this is done.

In order to run a machine code routine, we have to start it from BASIC. This means writing a small BASIC loader program, which clears the space for the machine code, loads that code, and then runs it. The simplest sort of loader would be along these lines:

```
10 CLEAR 24575: LOAD ""CODE: RANDOMIZE USR 24576
```

The first command, CLEAR, sets RAMTOP below the area occupied by the machine code, so BASIC doesn't overwrite it. It also clears the screen and moves the stack out of the way. The number that follows should usually be one byte below the first byte of your game. LOAD ""CODE loads the next code file on the tape, and RANDOMIZE USR effectively calls the machine code routine at the address specified, in this case 24576. This should be the entry point for your game. On a Spectrum, The ROM sits in the first 16K, and this is followed by various other things such as screen RAM, system variables and BASIC. A safe place for your code is above this area, all the way up to the top of RAM at address 65535. With just a short BASIC loader a start address of 24576, or even 24000 will give you plenty of room for your game.

This loader program is then saved to tape using a command like this

SAVE "name" LINE 10

LINE 10 indicates that on loading, the BASIC program is to auto-run from line 10.

After the BASIC loader comes the code file. You can save a code file like this:

SAVE "name" CODE 24576,40960

CODE tells the Spectrum to save a code file, as opposed to BASIC. The first number after this is the start address of the block of code, and the last number is its length.

That is simple enough, but what if we want to add a loading screen? Well, that is straightforward enough. We can load a screen using

```
LOAD ""SCREEN$
```

What this will do is load a block of code up to 6912 bytes long, to the start of the screen display at address 16384. Putting the screen file there is a bit trickier, because we cannot simply save out the screen as a file as the bottom two lines would be overwritten with the `Start tape, then press any key` message. So we load our picture into a point in RAM - say, 32768 - then use

```
SAVE "name" CODE 32768,6912
```

6912 is the size of the Spectrum's display RAM. When we reload the block from tape using `LOAD ""SCREEN$`, we are specifying that we want to force the code file to be loaded into screen memory. Under these circumstances it doesn't matter where the code file was located when it was saved.

Now we have another problem: wouldn't the `Bytes: name` message that is printed up on loading

the code block overwrite part of the screen? Well, yes it would. We can overcome this by poking the output stream.

```
POKE 23739,111
```

Will do the trick for us. So our BASIC loader now looks like this:

```
10 CLEAR 24575: LOAD ""SCREEN$: POKE 23739,111: LOAD ""CODE: RANDOMIZE USR 24576
```

Chapter Nineteen – Game Design

Fifty Percent Art, Fifty Percent Science

By now you should have enough knowledge to be capable of putting instructions together to form a game engine. Congratulations, you are now fifty percent of the way to writing a game. This chapter aims to deal with the remaining half. It is easy to fall into the trap of thinking that writing a game is as simple as learning how to order a huge array of instructions into a coherent sequence in order to carry out a grand plan. The technical side is, of course, essential. However, a good game developer needs to be much more than a skilled technician. He needs to be an artist too. That does not mean graphics either, although pleasing visuals are no bad thing. I am, of course, talking about the art of game design.

One could choose to go down the technical route, writing code to dazzle other "techies" who know how the Spectrum works and what it can ordinarily do. That is unlikely to impress everybody, however. For a start, your game may look amazing but could be somewhat dull to play and not hold the imagination for more than five minutes. Which is more, not every Spectrum fan knows the limitations of the machine. If you really wish to maximise your audience you need to appeal to those who neither know nor care what the Spectrum can do. Consider this: imagine if two Commodore 64 programmers were to release games on the same day. One wrote a Space Invaders clone which amazed the technical guys from the Commodore scene and one wrote a technically unremarkable but addictive game unlike anything you have seen before. As a Spectrum owner who has little knowledge of the C64's hardware limits, which would you be most likely to check out? The Space Invaders clone may display amazing technical wizardry for that machine, but is it likely to be doing anything you haven't seen before on some other format? Now consider that scenario the other way around: why would a Commodore, Amstrad, Acorn or Oric fan want to download your game? What makes it so special that he can't find something similar or better on his favoured machine?

If you wish to maximise interest, it is a good idea to offer something unique.

Gameplay Mechanics

Program code is like a big Lego construction made up of thousands of tiny little bricks, the individual instructions telling the CPU what to do. Gameplay mechanics are more akin to Duplo; bigger chunks, but still with infinite ways to rearrange them in order to achieve whichever effect is desired. One approach to game design might be to consider the following steps.

First, start by considering the player's control method. How does he move? Is he limited in some way, and if so, how? Will he be able to perform actions or complete tasks to increase or modify his manoeuvrability in some way? Generally speaking, any control method you can devise has probably been done before, but that shouldn't stop you asking these questions and coming up with interesting variations.

Second, consider the tasks the player is to undertake. What is his goal? How does he achieve victory? Is he shooting things, collecting them, growing them or preventing something from happening? Is he moving things around a play area for some purpose? Answering these questions and putting them together with a different control technique is where your game can start to differentiate itself from the vast ocean platformers, shoot-em-ups, maze or puzzle games already out

there. Start simple at first and add ideas as you go along. Don't be afraid to take others out if they don't work, even if you spent hours agonising over the code. If a mechanic doesn't work, it has no place in your game.

The third step is to consider the hazards the player will face. Is there a time limit? Are there moving enemies in the game? How do they move around, in a predictable manner or in another more interesting way? Are they deadly to the touch or do they sap a player's energy? Should they be destroyed or avoided or both? Do they change as the game progresses? Can they be used in some way?

When you have answered those questions, you need to consider whether the player can collect little bonuses or complete mini-tasks which give extra abilities, increase his score/bonus, add to his lives or in some other way aid him. Can they be achieved in a particular order to give an even bigger bonus? You may also wish to consider the opposite: are there items which might be collected which hinder the player by, for example, reversing his controls or slowing him down. One common mistake (I've made it myself) is to reduce the player's bonus opportunities as the game progresses. As the game becomes harder to play, it's usually a good idea to increase the bonuses in order to give the player a fighting chance.

If you really want to go to town, you may wish to add a little more depth. Is there a bigger picture for the player to consider? Does he need something to think about while he's dodging, jumping or blasting away? Are his achievements filling in a map, or a grid? Is he playing a giant game of noughts and crosses, animal, vegetable or mineral or battleships? Could you include some elements from left-field as the player makes his way through your game?

Play your game a lot while you are developing it and keep asking questions. Be insecure.

When you have done all of that, you may wish to consider the graphics, storyline and perhaps even music. Avoid the temptation to start with the storyline when developing a game for an eight bit machine, it simply doesn't have the graphics – or memory - to produce the atmosphere to do a brilliant storyline justice. You're not going to write the next Grim Fandango. Fit your storyline around the game design, not the other way around.

Finally, no matter how good and original your game is, remember that you cannot please all of the people all of the time. Just enjoy what you are doing, have fun and don't take it all too seriously.

Happy developing!

Appendix

Useful ROM and RAM Addresses

0: start of ROM
654: ROM routine which returns keypress (0-39) in e register, or 255 if nothing pressed.
949: BEEPER routine. Set the duration in DE and the pitch in HL.
3503: ROM routine to clear the screen, setting it to the colour in (23693).
6683: Displays the value in the BC register pair, up to a value of 9999.
8252: Displays a string at address DE with length BC on the screen.
8859: ROM routine to set the border colour to the value in the accumulator.
15616: address of ROM font, 96 chars * 8 bytes.
16384: 256x192 pixel display
22528: 32x24 colour attributes
23296: system variables
23606: pointer to font, minus 256. (256 = 32 * 8 bytes, 32 is the code for first printable character)
23560: ASCII code of the last keypress.
23672: clock, incremented 50 times per second.
23693: PAPER/INK/BRIGHT colour.
23695: PAPER/INK/BRIGHT colour.
23734: I/O channels
23755: BASIC area, followed by space for machine code programs. Extra hardware may move this.
24000: Arguably the lowest realistic starting point for a game, allowing 41536 bytes.
32767: last byte of RAM on a 16K Spectrum.
32768: Beginning of uncontended, faster RAM.
65535: last byte of RAM on a 48K Spectrum.