# ZX Spectrum Next Programming Notes

Theodore (Alex) Evans

September 22, 2019

# Contents

# Chapter 1

# Introduction

The ZX Spectrum Next is an extension of the original ZX Spectrum implemented in FPGA which implements many of the common additions to the system includin the characteristics of all of the original ZX Spectrum line, including the Timex/Sinclair 2068, along with a number of characteristics to modernize the design.

This document is an attempt to consolidate the programming interface for the ZX Spectrum Next into a single location. This document started when much of the documentation on the ZX Spectrum Next site (https://www.specnext.com/) was out of date and/or difficult to figure out. The way to figure out how things actually worked was to either dig through the forums and ask questions or find someones code that implemented a particular bit of functionality and reverse engineer it. The situation has greatly improved and this document may even be redundant at this point.

Description from http://www.specnext.com/about/:

The Spectrum Next is fully implemented with FPGA technology, ensuring it can be upgraded and enhanced while remaining truly compatible with the original hardware by using special memory chips and clever design. Here's what under the hood of the machine:

- Processor: Z80 normal and turbo modes
- Memory: 1024Kb RAM (expandable to 2048Kb on board)
- Video: Hardware sprites, 256 colours mode and more.
- Video Output: RGB, VGA, HDMI
- Storage: SD Card slot, with DivMMC-compatible protocol

- Audio: Turbo Sound Next (3x AY-3-8912 audio chips with stereo output)
- Joystick: DB9 compatible with Cursor, Kempston and Interface 2 protocols (selectable)
- PS/2 port: Mouse with Kempston mode emulation and an external keyboard
- Special: Multiface functionality for memory access, savegames, cheats etc.
- Tape support: Mic and Ear ports for tape loading and saving
- Expansion: Original external bus expansion port and accelerator expansion port
- Accelerator board (optional): GPU / 1Ghz CPU / 512Mb RAM
- Network (optional): Wi Fi module
- Extras: Real Time Clock (optional), internal speaker (optional)

# Chapter 2

# Video

ZX Spectrum Next video splits the display types into four categories (ULA, tilemap, layer 2, and sprites) which have their own sets of controls for colour palettes, clipping, and scrolling. Some aspects of ULA and tilemap are tied together, but all the rest operate in a largely independent manner using a layering system. The ULA category has a number of separate video modes that it can use. One of these (LoRes) is incompatible with using tilemaps.

## 2.1 Video Layering and Transparency

Video is rendered as three layers which are referred to as ULA (which includes the tilemap), layer 2, and sprites. The ordering of the layers is controlled by Next port $15 (21) bits 4-2:

Table 2.1: Video Layering

| Value | Top | Middle | Bottom |
|---|---|---|---|
| 000 | Sprites | Layer2 | ULA |
| 001 | Layer2 | Sprites | ULA |
| 010 | Sprites | ULA | Layer2 |
| 011 | Layer2 | ULA | Sprites |
| 100 | ULA | Sprites | Layer2 |
| 101 | ULA | Layer2 | Sprites |
| 110 | Sprites | | ULA+Layer2 (>7 = 7) |
| 111 | Sprites | | ULA+Layer2-5 (<0 = 0 / >7 = 7) |

Transparency for Layer 2, ULA, and LoRes are controlled by Next register \$14 (20) and defaults to \$E3. This colour ignores the state of the least significant blue bit, so \$E3 equates to both \$1C6 and \$1C7. For Sprites and Tilemaps transparency is determined by colour index. For Sprites this is controlled by register \$4B (with only the least significant 4-bits being relevant for 16-colour Sprites). For Tilemaps, the transparency index is set by register \$4C. If all layers are transparent, the transparency fallback colour is displayed. This is set by register \$4A.

(R/W) \$4A (74) ⇒ Transparency colour fallback

bits 7-0 = Set the 8 bit colour used if all layers are transparent.

(black on reset = 0)

## 2.2 Palette

**ULANext Colour Palette** Each video mode group has a pair of palettes assigned to it a primary and an alternate palette.

(R/W) \$43 (67) ⇒ Palette Control

bit 7 = '1' to disable palette write auto-increment.
bits 6-4 = Select palette for reading or writing:
000 = ULA first palette
100 = ULA second palette
001 = Layer 2 first palette
101 = Layer 2 second palette
010 = Sprites first palette
110 = Sprites second palette
011 = Tilemap first palette
111 = Tilemap second palette
bit 3 = Select Sprites palette (0 = first palette, 1 = second palette)
bit 2 = Select Layer 2 palette (0 = first palette, 1 = second palette)
bit 1 = Select ULA palette (0 = first palette, 1 = second palette)
bit 0 = Enable ULANext mode if 1. (0 after a reset)

(R/W) \$40 (64) ⇒ Palette Index

bits 7-0 = Select the palette index to change the associated colour.

For the ULA only, INKs are mapped to indices 0-7, Bright INKS to indices 8-15, PAPERs to indices 16-23 and Bright PAPERs to indices 24-31.

In ULANext mode, INKs come from a subset of indices 0-127 and PAPERs come from a subset of indices 128-255. The number of active indices depends on the number of attribute bits assigned to INK and PAPER out of the attribute byte. The ULA always takes border colour from paper.

(R/W) \$41 (65) ⇒ Palette Value (8 bit colour)

bits 7-0 = Colour for the palette index selected by the register \$40.

(Format is RRRGGGBB - the lower blue bit of the 9-bit colour will be a logical OR of blue bits 1 and 0 of this 8-bit value.)
After the write, the palette index is auto-incremented to the next index if the auto-increment is enabled at reg \$43. Reads do not auto-increment.

(R/W) \$44 (68) ⇒ Palette Value (9 bit colour)
Two consecutive writes are needed to write the 9 bit colour

1st write:
    bits 7-0 = RRRGGGBB
2nd write. If writing a L2 palette

    bit 7 = 1 for L2 priority colour, 0 for normal
    Priority colour will always be on top even on an SLU priority arrangement. If you need the exact same colour on priority and non priority locations you will need to program the same colour twice changing bit 7 to 0 for the second colour
    bits 6-1 = Reserved, must be 0
    bit 0 = lsb B
If writing another palette
    bits 7-1 = Reserved, must be 0
    bit 0 = lsb B

After the two consecutive writes the palette index is auto-incremented if the auto-increment is enabled by reg \$43.
Reads only return the 2nd byte and do not auto-increment.

**ULAPlus Colour Palette** From v3.00, the ZX Next emulates ULAPlus using the last 64 (192-255) entries of the ULA palette

**I/O ports** ULAplus is controlled by two ports.

\$BF3B is the register port (write only)

The byte output will be interpreted as follows:

Bits 0-5: Select the register sub-group
Bits 6-7: Select the register group. Two groups are currently available:
  00=palette group
  When this group is selected, the sub-group determines the entry in the palette table (0-63).
  01=mode group
  The sub-group is (optionally) used to mirror the video functionality of Timex port $FF as follows:
Bits 0-1: Screen mode.
  000=screen 0 (bank 5)
  001=screen 1 (bank 5)
  010=hi-colour (bank 5)
  100=screen 0 (bank 7)
  101=screen 1 (bank 7)
  110=hi-colour (bank 7)
  110=hi-res (bank 5)
  111=hi-res (bank 7)
Bits 3-5: Sets the screen colour in hi-res mode.
  000=Black on White
  001=Blue on Yellow
  010=Red on Cyan
  011=Magenta on Green
  100=Green on Magenta
  101=Cyan on Red
  110=Yellow on Blue
  111=White on Black

$FF3B is the data port (read/write)

When the palette group is selected, the byte written will describe the color.

When the mode group is selected, the byte output will be interpreted as follows:

Bit 0: ULAplus palette on (1) / off (0)
Bit 1: (optional) grayscale: on (1) / off (0) (same as turing the color off on the television)

Implementations that support the Timex video modes use the $FF register as the primary means to set the video mode, as per the Timex machines. It is left to the individual implementations to determine if reading the port returns the previous write or the floating bus.

**GRB palette entries** G3R3B2 encoding
For a device using the GRB colour space the palette entry is interpreted as follows

|  |  |
|---|---|
| Bits 0-1: | Blue intensity. |
| Bits 2-4: | Red intensity. |
| Bits 5-7: | Green intensity. |

This colour space uses a sub-set of 9-bit GRB. The missing lowest blue bit is set to OR of the other two blue bits (Bb becomes 000 for 00, and Bb1 for anything else). This gives access to a fixed half the potential 512 colour palette. The reduces the jump in intensity in the lower range in the earlier version of the specification. It also means the standard palette can now be represented by the ULAplus palette.

**Grayscale palette entries** In grayscale mode, each palette entry describes an intensity from zero to 255. This can be achieved by simply removing the colour from the output signal.

**Limitations** Although in theory 64 colours can be displayed at once, in practice this is usually not possible except when displaying colour bars, because the four CLUTs are mutually exclusive; it is not possible to mix colours from two CLUTs in the same cell. However, with software palette cycling it is possible to display all 256 colours on screen at once.

**Emulation** The 64 colour mode lookup table is organized as 4 palettes of 16 colours.

Bits 7 and 6 of each Spectrum attribute byte (normally used for FLASH and BRIGHT) will be used as an index value (0-3) to select one of the four colour palettes.

Each colour palette has 16 entries (8 for INK, 8 for PAPER). Bits 0 to 2 (INK) and 3 to 5 (PAPER) of the attribute byte will be used as indexes to retrieve colour data from the selected palette.

With the standard Spectrum display, the BORDER colour is the same as the PAPER colour in the first CLUT. For example BORDER 0 would set the border to the same colour as PAPER 0 (with the BRIGHT and FLASH bits not set).

The complete index can be calculated as
ink_colour = (FLASH * 2 + BRIGHT) * 16 + INK paper_colour = (FLASH * 2 + BRIGHT) * 16 + PAPER + 8

**Palette file format** The palette format doubles as the BASIC patch loader. This enables you to edit patches produced by other people.

```
; 64 colour palette file format (internal) - version 1.0
; copyright (c) 2009 Andrew Owen
;
; The palette file is stored as a BASIC program with embedded machine code

header:

db 0x00 ; program file
db 0x14, 0x01, "64colour" ; file name
dw 0x0097 ; data length
dw 0x0000 ; autostart line
dw 0x0097 ; program length

basic:

; 0 RANDOMIZE USR ((PEEK VAL "2
; 3635"+VAL "256"*PEEK VAL "23636"
; )+VAL "48"): LOAD "": REM

db 0x00, 0x00, 0x93, 0x00, 0xf9, 0xc0, 0x28, 0x28
db 0xbe, 0xb0, 0x22, 0x32, 0x33, 0x36, 0x33, 0x35
db 0x22, 0x2b, 0xb0, 0x22, 0x32, 0x35, 0x36, 0x22
db 0x2a, 0xbe, 0xb0, 0x22, 0x32, 0x33, 0x36, 0x33
db 0x36, 0x22, 0x29, 0x2b, 0xb0, 0x22, 0x34, 0x38
db 0x22, 0x29, 0x3a, 0xef, 0x22, 0x22, 0x3a, 0xea

start:

di ; disable interrupts
ld hl, 38 ; HL = length of code
```

```
add hl, bc ; BC = entry point (start) from BASIC
ld bc, 0xbf3b ; register select
ld a, 64 ; mode group
out (c), a ;
ld a, 1 ;
ld b, 0xff ; choose register port
out (c), a ; turn palette mode on
xor a ; first register

setreg:

ld b, 0xbf ; choose register port
out (c), a ; select register
ex af, af' ; save current register select
ld a, (hl) ; get data
ld b, 0xff ; choose data port
out (c), a ; set it
ex af, af' ; restore current register
inc hl ; advance pointer
inc a ; increase register
cp 64 ; are we nearly there yet?
jr nz, setreg ; repeat until all 64 have been done
ei ; enable interrupts
ret ; return

; this is where the actual data is stored. The following is an example palette.

registers:

db 0x00, 0x02, 0x18, 0x1b, 0xc0, 0xc3, 0xd8, 0xdb ; INK
db 0x00, 0x02, 0x18, 0x1b, 0xc0, 0xc3, 0xd8, 0xdb ; PAPER
db 0x00, 0x03, 0x1c, 0x1f, 0xe0, 0xe3, 0xfc, 0xff ; +BRIGHT
db 0x00, 0x03, 0x1c, 0x1f, 0xe0, 0xe3, 0xfc, 0xff ;
db 0xdb, 0xd8, 0xc3, 0xc0, 0x1b, 0x18, 0x02, 0x00 ; +FLASH
db 0xdb, 0xd8, 0xc3, 0xc0, 0x1b, 0x18, 0x02, 0x00 ;
db 0xff, 0xfc, 0xe3, 0xe0, 0x1f, 0x1c, 0x03, 0x00 ; +BRIGHT/
db 0xff, 0xfc, 0xe3, 0xe0, 0x1f, 0x1c, 0x03, 0x00 ; +FLASH

terminating_byte:
```

```
db 0x0d
```

## 2.3 ULA group

The ULA layer supports ZX Spectrum video, Timex video modes, and the Spectrum Next's lores video mode all use 16k memory bank 5 or 7 with the data coming from some combination of addresses \$0000-\$17FF (bitmap 1), \$1800-\$1AFF (attribute 1), \$2000-\$37FF (bitmap 2), and \$3800-\$3AFF (attribute 2) within the selected bank. Assuming default memory mapping and the use of bank 5 this will be mapped as some combination of memory \$4000-\$57FF, \$5800-\$5AFF, \$6000-\$77FF, \$780-\$7AFF. All of the modes other than the lores mode can either use the default ZX Spectrum colours, or ULANext mode which uses a 256 entry palette to determine the colour. In the Spectrum and Timex modes all colours are either Paper (foreground), paper (background), or border colours.

### 2.3.1 Colour Attributes

The ZX Spectrum Next has two major modes for colour attributes allowing a total of nine ways to map the palette to the original ZX Spectrum attributes. One with flashing enabled and eight with flashing disabled. This mapping is controlled by Next registers \$42 (66, palette format) and \$43 (67, palette control). Palette control switches between flashing enabled (0, default) and flashing disabled (1)

Table 2.2: Flashing Enabled

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Function | F | B | $P_2$ | $P_1$ | $P_0$ | $I_2$ | $I_1$ | $I_0$ |

Flashing Enabled is similar to the original Spectrum colour attributes. INKs are mapped to indices 0-7, Bright INKS to indices 8-15, PAPERs to indices 16-23 and Bright PAPERs to indices 24-31.

The ULA next modes use a varying number of bits from the attribute byte to determine the ink colour as the palette index from the appropriate bits (all others being zero) and the paper colours coming from the indicated value+128 with palette format 255 being a special case where all the bits determine the ink colour while the paper is always palette index 128. The

ULA always takes border colour from paper.

Table 2.3: ULA Next

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| format 1 | $P_6$ | $P_5$ | $P_4$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ | $I_0$ |
| format 3 | $P_5$ | $P_4$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ | $I_1$ | $I_0$ |
| format 7 | $P_4$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ | $I_2$ | $I_1$ | $I_0$ |
| format 15 | $P_3$ | $P_2$ | $P_1$ | $P_0$ | $I_3$ | $I_2$ | $I_1$ | $I_0$ |
| format 31 | $P_2$ | $P_1$ | $P_0$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | $I_0$ |
| format 63 | $P_1$ | $P_0$ | $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | $I_0$ |
| format 127 | $P_0$ | $I_6$ | $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | $I_0$ |
| format 255 | $I_7$ | $I_6$ | $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | $I_0$ |

### 2.3.2 ZX Spectrum Mode

Timex mode 0

This is the default ULA mode and has its origins in the original ZX Spectrum. It uses 256 × 192 pixels with 8 × 8 colour attribute areas mapped into a 32 × 24 grid. If Timex modes are not enabled, this and the LoRes mode are the only ones available, so you would switch back to this mode by writing 000xxxxx to Next register $15 (21, the sprites and layers register). If another Timex mode is enabled, then this is mode 0 so you would write 0 to port $ff to enable it. This is a 256 × 192 video mode. The bitmap 1 area is used for selection between ink and paper colours with one bit per pixel and the attribute 1 area for colour attributes.

The easiest way to visualize the mapping of this mode is to think of the 256 × 192 area as being divided into a 32 × 24 grid of 8 × 8 characters. IF we consider X and Y as the position in the grid and R to the the row within the character. For ink/paper selection, 0=paper, 1=ink and the entries are stored left to right as lsb to msb within the bye. The address for a pixel value is: $0R_4R_3Y_2Y_1Y_0R_2R_1R_0C_4C_3C_2C_1C_0$. Each 8 × 8 cell has its own colour attribute where the address for an attribute cell is $0110R_4R_3R_2R_1R_0C_4C_3C_2C_1C_0$ in other words mapped lineally column-wise starting at the beginning of the attribute 1 area.

Code:

```
;; from any other Timex mode:
  ld a,$00
  ld c,$ff
  out (c),a

  ;; from LoRes mode:
  ld bc,$243B ; next register select port
  ld a,$15
  out (c),a
  ld bc,$253B ; next register r/w port
  in a,(c)
  and $7f
  out (c),a
```

### 2.3.3 Alternate Page Mode

Timex mode 1

This mode is the same as ZX Spectrum mode at alternate addresses. Alternate page mode is selected by enabling Timex modes by writing 00xxxx1xx to Next register $08 (8, Peripheral 3 setting) then writing 1 to the Timex ULA port ($ff). It is identical to ZX Spectrum mode except the pixel are mapped to the bitmap 2 area giving use pixel addresses of $1R_4R_3Y_2Y_1Y_0R_2R_1R_0C_4C_3C_2C_1C_0$ and the attributes to the attribute 2 area with addresses of $1110R_4R_3R_2R_1R_0C_4C_3C_2C_1C_0$.

Code:

```
;; disable LoRes mode:
ld bc,$243B ; next register select port
ld a,$15
out (c),a
ld bc,$253B ; next register r/w port
in a,(c)
and $7f
out (c),a
;; set Timex mode
ld bc,$243B ; next register select port
ld a,$08
out (c),a
ld bc,$253B ; next register r/w port
in a,(c)
```

```
or $04
out (c),a
;; set alternate page mode
ld c,$ff
ld a,$01
out (c),a
```

### 2.3.4 Timex Hi-Colour Mode

Timex mode 2

This mode is a 256 × 192 video mode with 8 × 1 colour attribute mapping on a 32 × 192 grid. It is selected by writing 2 to the Timex ULA port ($ff). Pixel mapping in this mode is the same as in ZX Spectrum mode using the bitmap 1 area based on $0R_4R_3Y_2Y_1Y_0R_2R_1R_0C_4C_3C_2C_1C_0$. The colour attributes use the bitmap 2 area with 8 × 1 colour attribute areas corresponding to the addresses $1R_4R_3Y_2Y_1Y_0R_2R_1R_0C_4C_3C_2C_1C_0$.

Code:

```
;; disable LoRes mode:
ld bc,$243B ; next register select port
ld a,$15
out (c),a
ld bc,$253B ; next register r/w port
in a,(c)
and $7f
out (c),a
;; set Timex mode
ld bc,$243B ; next register select port
ld a,$08
out (c),a
ld bc,$253B ; next register r/w port
in a,(c)
or $04
out (c),a
;; set hi-colour mode
ld c,$ff
ld a,$02
out (c),a
```

### 2.3.5 Timex Hi-Resolution Mode

Timex mode 6

This is a monochrome 512 × 192 video mode. It is selected by writing to the Timex ULA port ($ff with values that also select which two colours (or colour entries in ULANext mode) you use.

Table 2.4: Hi-Resolution Colours

| Port 0xff bits 5-3 | Attribute | Ink | Paper |
|---|---|---|---|
| 000 | 01111000 | black | white |
| 001 | 01110001 | blue | yellow |
| 010 | 01101010 | red | cyan |
| 011 | 01100011 | magenta | green |
| 100 | 01011100 | green | magenta |
| 101 | 01010101 | cyan | red |
| 110 | 01001110 | yellow | blue |
| 111 | 01000111 | white | black |

Pixels are mapped into both the bitmap 1 and bitmap 2 areas where 8-pixel wide character columns alternate between the two bitmap areas. The pixels within a byte being rendered left to right lsb to msb as in other Spectrum video modes. The addresses for each row within a character are based on a 64 × 32 grid of 8 × 8 characters which using a 64 × 24 R, C, and Y scheme gives us addresses of the form $C_0R_4R_3Y_2Y_1Y_0R_2R_1R_0C_5C_4C_3C_2C_1$.

Code:

```
;; disable LoRes mode:
ld bc,$243B ; next register select port
ld a,$15
out (c),a
ld bc,$253B ; next register r/w port
in a,(c)
and $7f
out (c),a
;; set Timex mode
ld bc,$243B ; next register select port
ld a,$08
out (c),a
ld bc,$253B ; next register r/w port
```

```
in a,(c)
or $04
out (c),a
;; set hi-res mode, black on white
ld c,$ff
ld a,$06
out (c),a
```

### 2.3.6 Lo-Resolution Mode

This is a Spectrum Next specific video mode with a resolution of $128 \times 96$ replacing the old Radistan mode. It allows for independent selection from the 256 entries in the ULA palette on a pixel by pixel basis. The pixel data is mapped into the bitmap 1 and bitmap 2 areas. It is selected by writing $100xxxxx$ to Next register $15 (21, the sprites and layers register). Each byte corresponds to a ULA palette entry and bytes are mapped linearly in a row-wise fashion.

Code:

```
;; enable LoRes mode:
ld bc,$243B ; next register select port
ld a,$15
out (c),a
ld bc,$253B ; next register r/w port
in a,(c)
or $80
out (c),a
```

### 2.3.7 Programming

(R/W) $15 (21) ⇒ Sprite and Layers system

bit 7 = LoRes mode, 128 x 96 x 256 colours (1 = enabled)
bit 6 = Sprite priority (1 = sprite 0 on top, 0 = sprite 127 on top)
bit 5 = Enable sprite clipping in over border mode (1 = enabled)
bits 4-2 = set layers priorities:
Reset default is 000, sprites over the Layer 2, over the ULA graphics
000 - S L U
001 - L S U

010 - S U L
011 - L U S
100 - U S L
101 - U L S
110 - S(U+L) ULA and Layer 2 combined, colours clamped to 7
111 - S(U+L-5) ULA and Layer 2 combined, colours clamped to [0,7]

bit 1 = Over border (1 = yes)(Back to 0 after a reset)
bit 0 = Sprites visible (1 = visible)(Back to 0 after a reset)

Port $ff (255) Timex Sinclair/floating bus
Disable with bit 2 to Nextreg $08

bit 7: memory paging (not on Next)
bit 6: disables generation of interrupts
bits 3-5: set hi-res mode colour combination
bits 0-2: screen mode
000=normal ULA mode
001=alternate ULA address
010=hi-colour
110=hi-res

For information on controlling the palette see the Palette section.

(R/W) $14 (20) ⇒ Global transparency color

bits 7-0 = Transparency color value ($E3 after a reset)

(Note: this value is 8-bit, so the transparency is compared against only by the MSB bits of the final 9-bit colour)
(Note2: this only affects Layer 2, ULA and LoRes. Sprites use register $4B for transparency and tilemap uses nextreg $4C)

(R/W) $1A (26) ⇒ Clip Window ULA/LoRes

bits 7-0 = Coord. of the clip window
1st write = X1 position
2nd write = X2 position
3rd write = Y1 position
4rd write = Y2 position

The values are 0,255,0,191 after a Reset
Reads do not advance the clip position

(W) $1C (28) ⇒ Clip Window control

bits 7-4 = Reserved, must be 0
bit 3 - reset the tilemap clip index
bit 2 - reset the ULA/LoRes clip index.
bit 1 - reset the sprite clip index.
bit 0 - reset the Layer 2 clip index.

(R) \$1C (28) ⇒ Clip Window control
(may change)

bits 7-6 = Tilemap clip index
bits 5-4 = Layer 2 clip index
bits 3-2 = Sprite clip index
bits 1-0 = ULA clip index

(R/W) \$32 (50) ⇒ ULA / LoRes Offset X

bits 7-0 = X Offset (0-255)(Reset to 0 after a reset)

ULA can only scroll in multiples of 8 pixels so the lowest 3 bits have no effect at this time.
LoRes scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

(R/W) \$33 (51) ⇒ ULA / LoRes Offset Y

bits 7-0 = Y Offset (0-191)(Reset to 0 after a reset)

LoRes scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

(R/W) \$42 (66) ⇒ ULANext Attribute Byte Format

bits 7-0 = Mask indicating which bits of an attribute byte are used to represent INK. The rest will represent PAPER.

(15 on reset)
The mask can only indicate a solid sequence of bits on the right side of the attribute byte (1, 3, 7, 15, 31, 63, 127 or 255).
INKs are mapped to base index 0 in the palette and PAPERs and border are mapped to base index 128 in the palette.
The 255 value enables the full ink colour mode making all the palette entries INK. PAPER and border both take on the fallback colour (nextreg \$4A) in this mode.

Port \$7ffd (32765) ZX Spectrum 128 Memory
Disable with bit 5 port \$7ffd

bits 6-7: reserved
bit 5: Lock memory paging (unlocks only on reset)
bit 4: ROM Select (low bit of ROM select for +2/+3)
bit 3: Shadow screen toggle
bits 0-2: Bank number for slot 4

## 2.4 Tilemap Mode

February 25, 2019 Phoebus Dokos

### 2.4.1 General Description

The tilemap is a hardware character oriented display that comes in two resolutions: $40 \times 32$ ($320 \times 256$ pixels) and $80 \times 32$ ($640 \times 256$ pixels).

The display area on screen is the same as the sprite layer, meaning it overlaps the standard $256 \times 192$ area by 32 pixels on all sides. Vertically this is larger than the physical HDMI display, which will cut off the top and bottom character rows making the visible area $40 \times 30$ or $80 \times 30$, but the full area is visible on VGA.

The obvious application for the tilemap is for a fast, clearly readable and wide multicoloured character display. Less obvious perhaps is that it can also be used to make fast and wide resolution full colour backgrounds with easily animated components.

The tilemap is defined by two data structures.

### 2.4.2 Data Structures

**Tilemap** The first data structure is the tilemap itself which indicates what characters occupy each cell on screen. Each tilemap entry is two bytes so for $40 \times 32$ resolution, a full size tilemap will occupy 2560 bytes, and for $80 \times 32$ resolution the space taken is twice that at 5120 bytes. The tilemap entries are stored in X-major order and each two-byte tilemap entry is stored little endian:

Tilemap Entry

bits 15-12 : palette offset

bit 11 : x mirror
bit 10 : y mirror
bit 9 : rotate
bit 8 : ULA over tilemap (in 512 tile mode, bit 8 of the tile number)
bits 7-0 : tile number

The character displayed is indicated by the "tile number" which can be thought of as an ASCII code. The tile number is normally eight bits allowing up to 256 unique tiles to be displayed but this can be extended to nine bits for 512 unique tiles if 512 tile mode is enabled via the Tilemap Control register.

The other bits are tile attributes that modify how the tile image is drawn. Their function is the same as the equivalent sprite attributes for sprites. Bits apply rotation then mirroring, and colour can be shifted with a palette offset. If 512 tile mode is not enabled, bit 8 will determine if the tile is above or below the ULA display on a per tile basis.

**Tile Definitions** The second data structure is the tile definitions themselves.

Each tile, identified by tile number, is $8 \times 8$ pixels in size with each pixel four bits to select one of 16 colours. A tile definition occupies 32 bytes and is defined in X major order with packing in the X direction in the same way that 4-bit sprites are defined. The 4-bit colour of each pixel is augmented by the 4-bit palette offset from the tilemap in the most significant bits to form an 8-bit colour index that is looked up in the tilemap palette to determine the final 9-bit colour sent to the display.

Tiles are therefore defined using 16 colours with the tilemap palette offset able to act as index into the tilemap palette to vary the display colour. One of the 16 colours is defined as transparent in the Transparency Index register.

### 2.4.3 Memory Organization & Display Layer

The tilemap is a logical extension of the ULA and its data structures are contained in the ULAís 16k bank 5. If both the ULA and tilemap are enabled, this means that the tilemapís map and tile definitions should be arranged within the 16k to avoid overlap with the display ram used by the ULA.

The tilemap exists on the same display layer as the ULA. The graphics generated by the ULA and tilemap are combined before being forwarded to the SLU layer system as layer U.

### 2.4.4 Combining ULA & Tilemap

The combination of the ULA and tilemap is done in one of two modes: the standard mode or the stencil mode.

The standard mode uses bit 8 of a tile's tilemap entry to determine if a tile is above or below the ULA. The source of the final pixel generated is then the topmost non-transparent pixel. If the ULA or tilemap is disabled then they are treated as transparent.

The stencil mode will only be applied if both the ULA and tilemap are enabled. In the stencil mode, the final pixel will be transparent if either the ULA or tilemap are transparent. Otherwise the final pixel is a logical AND of the corresponding colour bits. The stencil mode allows one layer to act as a cut-out for the other.

### 2.4.5 Programming Tilemap mode

(R/W) \$6B (107) $\Rightarrow$ Tilemap Control

bit 7 = 1 to enable the tilemap
bit 6 = 0 for $40 \times 32$, 1 for $80 \times 32$
bit 5 = 1 to eliminate the attribute entry in the tilemap
bit 4 = palette select
bits 3-2 = Reserved set to 0
bit 1 = 1 to activate 512 tile mode
bit 0 = 1 to force tilemap on top of ULA

Bits 7 & 6 enable the tilemap and select resolution. Bit 4 selects one of two tilemap palettes used for final colour lookup. Bit 5 changes the structure of the tilemap so that it contains only 8-bit tilemap entries instead of 16-bit tilemap entries. If 8-bit, the tilemap only contains tile numbers and the attributes are instead taken from nextreg \$6C.

Bit 1 activates 512 tile mode. In this mode, the "ULA over tilemap" bit in a tile's attribute is re-purposed as the ninth bit of the tile number, allowing up to 512 unique tiles to be displayed. In this mode, the ULA is always on top of the tilemap.

Bit 0 forces the tilemap to be on top of the ULA. It can be useful in 512 tile mode to change the relative display order of the ULA and tilemap.

(R/W) \$6C (108) ⇒ Default Tilemap Attribute

> bits 7-4 = Palette Offset
> bit 3 = X mirror
> bit 2 = Y mirror
> bit 1 = Rotate
> bit 0 = ULA over tilemap
> (bit 8 of the tile number if 512 tile mode is enabled)

If bit 5 of nextreg \$6B is set, the tilemapís structure is modified to contain only 8-bit tile numbers instead of the usual 16-bit tilemap entries. In this case, the tile attributes used are taken from this register instead.

(R/W) \$6E (110) ⇒ Tilemap Base Address

> bits 7-6 = Read back as zero, write values ignored
> bits 5-0 = MSB of address of the tilemap in Bank 5

This register determines the tilemapís base address in bank 5. The base address is the MSB of an offset into the 16k bank, allowing the tilemap to begin at any multiple of 256 bytes in the bank. Writing a physical MSB address in \$40-\$7f or \$c0-\$ff, corresponding to traditional ULA physical addresses, is permitted. The value read back should be treated as a fully significant 8-bit value.

The tilemap will be $40 \times 32$ or $80 \times 32$ in size depending on the resolution selected in nextreg \$6B. Each entry in the tilemap is normally two bytes but can be one byte if attributes are eliminated by setting bit 5 of nextreg \$6B.

(R/W) \$6F (111) ⇒ Tile Definitions Base Address

> bits 7-6 = Read back as zero, write values ignored
> bits 5-0 = MSB of address of tile definitions in Bank 5

This register determines the base address of tile definitions in bank 5. As with nextreg \$6E, the base address is the MSB of the an offset into the 16k bank, allowing tile definitions to begin at any multiple of 256 bytes in the bank. Writing a physical MSB address in \$40-\$7f or \$c0-\$ff, corresponding to traditional ULA physical addresses, is permitted. The value read back should be treated as a fully significant 8-bit value.

Each tile definition is 32 bytes in size and is located at address:
Tile Def Base Addr + 32 * (Tile Number)

(R/W) $4C (76) ⇒ Transparency index for the tilemap

> bits 7-4 = Reserved, must be 0
> bits 3-0 = Set the index value ($F after reset)

Defines the transparent colour index for tiles. The 4-bit pixels of a tile definition are compared to this value to determine if they are transparent.

For palette information see palette section.

(R/W) $1B (27) ⇒ Clip Window Tilemap

> bits 7-0 = Coord. of the clip window
> 1st write = X1 position
> 2nd write = X2 position
> 3rd write = Y1 position
> 4rd write = Y2 position

The values are 0,159,0,255 after a Reset
Reads do not advance the clip position

The tilemap display surface extends 32 pixels around the central 256 × 192 display. The origin of the clip window is the top left corner of this area 32 pixels to the left and 32 pixels above the central 256 × 192 display. The X coordinates are internally doubled to cover the full 320 pixel width of the surface. The clip window indicates the portion of the tilemap display that is non-transparent and its indicated extent is inclusive; it will extend from X1*2 to X2*2+1 horizontally and from Y1 to Y2 vertically.

(R/W) $2F (47) ⇒ Tilemap Offset X MSB

> bits 7-2 = Reserved, must be 0
> bits 1-0 = MSB X Offset

Meaningful Range is 0-319 in 40 char mode, 0-639 in 80 char mode

(R/W) $30 (48) ⇒ Tilemap Offset X LSB

> bits 7-0 = LSB X Offset

Meaningful range is 0-319 in 40 char mode, 0-639 in 80 char mode

(R/W) $31 (49) ⇒ Tilemap Offset Y

> bits 7-0 = Y Offset (0-191)

These are scroll registers for scrolling the tilemap area. As with other layers, the scroll region wraps.

(R/W) \$68 (104) ⇒ ULA Control

> bit 7 = 1 to disable ULA output
> bit 6 = 0 to select the ULA colour for blending in SLU modes 6 & 7
> = 1 to select the ULA/tilemap mix for blending in SLU modes 6 & 7
> bits 5-1 = Reserved must be 0
> bit 0 = 1 to enable stencil mode when both the ULA and tilemap are enabled
> (if either are transparent the result is transparent otherwise the result is a logical AND of both colours)

Bit 0 can be set to choose stencil mode for the combined output of the ULA and tilemap. Bit 6 determines what colour is used in SLU modes 6 & 7 where the ULA is combined with Layer 2 to generate highlighting effects.

**Changes Since 2.00.26**

1. 512 Tile Mode. In 2.00.26, the 512 tile mode was automatically selected when the ULA was disabled. With the ULA disabled, the tilemap attribute bit “ULA on top” was re-purposed to be bit 8 of the tile number. In 2.00.27, selection of the 512 tile mode is moved to bit 1 of Tilemap Control nextreg \$6B. This way 512 tile mode can be independently chosen without disabling the ULA. The “ULA on top” bit is still taken as bit 8 of the tile number and in the 512 mode, the tilemap is always displayed underneath the ULA.
2. Tilemap Always On Top of ULA. In 2.00.27, bit 0 of Tilemap Control nextreg \$6B is used to indicate that the tilemap should always be displayed on top of the ULA. This allows the tilemap to display over the ULA when in 512 mode.

**Future Direction** The following compatible changes may be applied at a later date:

1. Addition of a bit to Tilemap Control to select a reduced tilemap area of size $32 \times 24$ or $64 \times 24$ that covers the ULA screen.
2. Addition of a bit to Tilemap Control to select split addressing where the tilemap’s tiles and attributes as well as the tile definitions are split between the two 8k halves of the 16k ULA ram in the same way that

the two Timex display files are split. The intention is to make it easier for the tilemap to co-exist with all the display modes of the ULA.

## 2.5 Layer 2

Layer 2 is a linearly mapped, row-wise, upper left to lower right, 256 × 192 × 256 bit-map graphics area. Both the main version (8k pages 16-21/16k banks 8-10) and a shadow version (8k pages 22-27/16k banks 11-13) are six contiguous 8k pages (three contiguous 16k blocks) in the extended RAM space indicated by the contents of Next registers $12 (18 Layer 2 RAM page, default 8) and $13 (19, Layer 2 shadow page, default 11). The layer 2 pages can be accessed either by mapping the pages into normal RAM, or write only access using port $123B to fix them in the same space as the ROMs at $0000-$3FFF. The colours come from the indices in the layer 2 palette. Layer 2 is drawn according to the values in registers $16 (22, Layer 2 Offset X, default 0) and $17 (23, Layer 2 Offset Y, default 0).

### 2.5.1 Programming

Port $123b (4667) Layer 2

> bits 6-7: Video RAM bank select
> bit 3: Shadow layer 2 select
> bit 1: Layer 2 visible
> bit 0: Enable layer 2 write paging

For information on controlling the palette see the Palette section.

(R/W) $14 (20) ⇒ Global transparency color

> bits 7-0 = Transparency color value ($E3 after a reset)

(Note: this value is 8-bit, so the transparency is compared against only by the MSB bits of the final 9-bit colour)
(Note2: this only affects Layer 2, ULA and LoRes. Sprites use register $4B for transparency and tilemap uses nextreg $4C)

(R/W) $16 (22) ⇒ Layer2 Offset X

> bits 7-0 = X Offset (0-255)(0 after a reset)

(R/W) $17 (23) ⇒ Layer2 Offset Y

bits 7-0 = Y Offset (0-191)(0 after a reset)

(R/W) $18 (24) ⇒ Clip Window Layer 2

bits 7-0 = Coords of the clip window
1st write - X1 position
2nd write - X2 position
3rd write - Y1 position
4rd write - Y2 position

Reads do not advance the clip position
The values are 0,255,0,191 after a Reset

(W) $1C (28) ⇒ Clip Window control

bits 7-4 = Reserved, must be 0
bit 3 - reset the tilemap clip index
bit 2 - reset the ULA/LoRes clip index.
bit 1 - reset the sprite clip index.
bit 0 - reset the Layer 2 clip index.

(R) $1C (28) ⇒ Clip Window control
(may change)

bits 7-6 = Tilemap clip index
bits 5-4 = Layer 2 clip index
bits 3-2 = Sprite clip index
bits 1-0 = ULA clip index

(R/W) $12 (18) ⇒ Layer 2 RAM bank

bits 7-6 = Reserved, must be 0
bits 5-0 = RAM bank (point to bank 8 after a Reset, NextZXOS modifies to 9)

(R/W) $13 (19) ⇒ Layer 2 RAM shadow bank

bits 7-6 = Reserved, must be 0
bits 5-0 = RAM bank (point to bank 11 after a Reset, NextZXOS modifies to 12)

## 2.6 Sprites

February 25, 2019 Victor Trucco

The Spectrum Next has a hardware sprite system with the following characteristics:

- Total of 128 sprites
- Display surface is $320 \times 256$ overlapping the ULA screen by 32 pixels on each side
- Minimum of 100 sprites per scanline*
- Choice of 512 colours for each pixel
- Site of each sprite is $16 \times 16$ pixels but sprites can be magnified $2\times$, $4\times$ or $8\times$ horizontally and vertically
- Sprites can be mirrored and rotated
- Sprites can be grouped together to form larger sprites under the control of a single anchor
- A 16K pattern memory can contain 64 8-bit sprite images or 128 4-bit sprite images and combinations in-between
- A per sprite palette offset allows sprites to share images but colour them differently
- A nextreg interface allows the copper to move sprites during the video frame

*A minimum of 100 $16 \times 16$ sprites is guaranteed to be displayed in any scanline. Any additional sprites will not be displayed with the hardware ensuring sprites are not partially plotted.

The actual limit is determined by how many 28MHz clock cycles there are in a scanline. The sprite hardware is able to plot one pixel cycle and uses one cycle to qualify each sprite. Since the number of cycles there are in a scanline varies with video timing (HDMI, VGA), the number of pixels that can be plotted also varies but the minimum will be 1600 pixels per line including overhead cycles needed to qualify 100 sprites. Since sprites magified horizontally involve plotting more pixels, $2\times$, $4\times$, and $8\times$ sprites will take more cycles to plot and the presence of these sprites in a line will reduce the total number of sprites that can be plotted.

### 2.6.1 Sprite Patterns

Sprite patterns are the images that each sprite can take on. The images are stored in a 16K memory internal to the FPGA and are identified by pattern number. A particular sprite chooses a pattern by storing a pattern number in its attributes.

All sprites are 16 × 16 pixels in size but the come in two flavours: 4-bit and 8-bit. The bit width describes how many bits are used to code the colour of each pixel.

An 8-bit sprite uses a full byte to colour each of its pixels so that each pixel can be one of 256 colours. In this case, a 16 × 16 sprite requires 256 bytes of pattern memory to store its image.

A 4-bit sprite uses a nibble to colour each of its pixels so that each pixel can be one of 16 colours. In this case, a 16 × 16 sprite requires just 128 bytes of pattern memory to store its image.

The 16K pattern memory can contain any combination of these images, whether they are 128 bytes or 256 bytes and their locations in the pattern memory are described by a pattern number. This pattern number is 7 bits with bits named as follows:

### Pattern Number

```
N5 N4 N3 N2 N1 N0 N6
```

N6, despite the name, is the least significant bit.

This 7-bit pattern number can identify 128 patterns in the 16k pattern memory, each of which are 128 bytes in size. The full 7-bits are therefore used for 4-bit sprites.

For 8-bit sprites, N6=0 always. The remaining 6 bits can identify 64 patterns, each of which is 256 bytes in size.

The N5:N0,N6 bits are stored in a particular sprite's attributes to identify which image a sprite uses.

**8-Bit Sprite Patterns** The 16×16 pixel image uses 8-bits for each pixel so that each pixel can be one of 256 colours. One colour indicates transparency and this is programmed into the Sprite Transparency Index register (nextreg $4B). By default the transparent value is $E3.

As an example of an 8-bit sprite, let's have a look at figure 1.1.

Using the default palette, which is initialised with RGB332 colours from 0-255, the hexadecimal values for this pattern arranged in a 16 × 16 array are shown below:

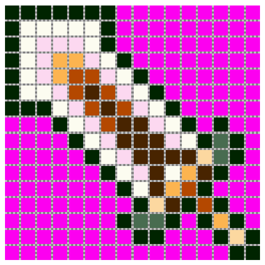

Figure 2.1: Pattern Example

```
04040404040404E3E3E3E3E3E3E3E3E3
04FFFFFFFFFF04E3E3E3E3E3E3E3E3E3
04FFFBFBFBFF04E3E3E3E3E3E3E3E3E3
04FFFBF5F5FBFF04E3E3E3E3E3E3E3E3
04FFFBF5A8A8FBFF04E3E3E3E3E3E3E3
04FFFFFBA844A8FBFF04E3E3E3E3E3E3
040404FFFBA844A8FBFF04E3E3E3E3E3
E3E3E304FFFBA84444FBFF04E304E3E3
E3E3E3E304FFFB444444FBFF044D04E3
E3E3E3E3E304FFFB44444444FA4D04E3
E3E3E3E3E3E304FFFB44FFF54404E3E3
E3E3E3E3E3E3E304FF44F5A804E3E3E3
E3E3E3E3E3E3E3E304FA4404A804E3E3
E3E3E3E3E3E3E3044D4D04E304F504E3
E3E3E3E3E3E3E3E30404E3E3E304FA04
E3E3E3E3E3E3E3E3E3E3E3E3E3E30404
```

Here $E3 is used as the transparent index.

These 256 bytes would be stored in pattern memory in left to right, top to bottom order.

**4-Bit Sprite Patterns** The $16 \times 16$ pixel image uses 4-bits for each pixel so that each pixel can be one of 16 colours. One colour indicates transparency and this is programmed into the lower 4-bits of the Sprite Transparency Index register (nextreg $4B). By default the transparency value is $3. Note that the same register is shared with 8-bit patterns to identify the transparent index.

Since each pixel only occupies 4-bits, two pixels are stored in each byte. The leftmost pixel is stored in the upper 4-bits and the rightmost pixel is stored in the lower 4-bits.

As an example we will use the same sprite image as was given in the 8-bit pattern example. Here only the lower 4 bits of each pixel is retained to confine each pixel's color to 4-bits:

```
4444444333333333
4FFFFF4333333333
4FBBBF4333333333
4FB55BF433333333
4FB588BF43333333
4FFB848BF4333333
444FB848BF433333
3334FB844BF43433
33334FB444BF4D43
333334FB4444AD43
3333334FB4F54433
33333334F4584333
333333334A448433
33333334DD434543
33333333443334A4
3333333333333344
```

$3 is used as the transparent index.

These 128 bytes would be stored in pattern memory in left to right, top to bottom order.

The actual colour that will appear on screen will depend on the palette, described below. The default palette will not likely generate suitable colours for 4-bit sprites.

### 2.6.2 Sprite Palette

Each pixel of a sprite image is 8-bit for 8-bit patterns or 4-bit for 4-bit patterns. The pixel value is known as a pixel colour index. This colour index is combined with the sprite's palette offset. The palette offset is a 4-bit value added to the top 4-bits of the pixel colour index. The purpose of the palette offset is to allow a sprite to change the colour of an image.

The final sprite colour index generated by the sprite hardware is then the sum of the pixel index and the 4-bit palette offset. In pictures using binary math:

```
8-bit Sprite
PPPP0000
+ IIIIIIII
----------
SSSSSSSS

4-bit Sprite
PPPP0000
+ 0000IIII
----------
SSSSSSSS = PPPPIIII
```

Where “PPPP” is the 4-bit palette offset from the sprite’s attributes and the “I”s represent the pixel value from the sprite pattern. The final sprite index is represented by the 8-bit value “SSSSSSSS”.

For 4-bit sprites the palette offset can be thought of as selecting one of 16 different 16-colour palettes.

This final 8-bit sprite index is then passed through the sprite palette which acts like a lookup table that returns the 9-bit RGB333 colour associated with the sprite index.

At power up, the sprite palette is initialized such that the sprite index passes through unchanged and is therefore interpretted as an RGB332 colour. The missing third blue bit is generated as the logical OR of the two other blue bits. In short, for 8-bit sprites, the sprite index also acts like the colour when using the default palette.

### 2.6.3 Sprite Attributes

A sprite’s attributes is a list of properties that determine how and where the sprite is drawn.

Each sprite is described by either 4 or 5 attribute bytes listed below:

Sprite Attribute 0

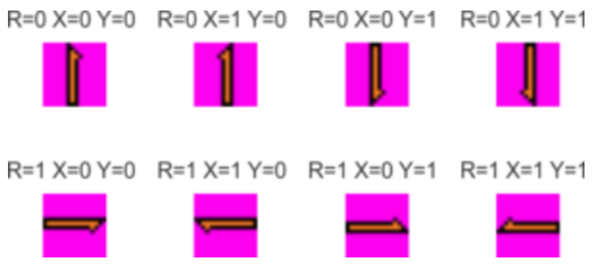

Figure 2.2: All Rotate and Mirror Flags

```
X X X X X X X X
```

The least significant eight bits of the sprite's X coordinate. The ninth bit is found in sprite attribute 2.

Sprite Attribute 1

```
Y Y Y Y Y Y Y Y
```

The least significant eight bits of the sprite's Y coordinate. The ninth bit is optional and is found in attribute 4.

Sprite Attribute 2

```
P P P P XM YM R X8/PR
```

P = 4-bit Palette Offset
XM = 1 to mirror the sprite image horizontally
YM = 1 to mirror the sprite image vertically
R = 1 to rotate the sprite image 90 degrees clockwise
X8 = Ninth bit of the sprite's X coordinate
PR = 1 to indicate P is relative to the anchor's palette offset (relative sprites only)
Rotation is applied before mirroring.
Relative sprites, described below, replace X8 with PR.

Sprite Attribute 3

```
V E N5 N4 N3 N2 N1 N0
```

V = 1 to make the sprite visible
E = 1 to enable attribute byte 4

N = Sprite pattern to use 0-63
If E=0, the sprite is fully described by sprite attributes 0-3. The sprite pattern is an 8-bit one identified by pattern N=0-63. The sprite is an anchor and cannot be made relative. The sprite is displayed as if sprite attribute 4 is zero.
If E=1, the sprite is further described by sprite attribute 4.

Sprite Attribute 4

A. Extended Anchor Sprite

   ```
   H N6 T X X Y Y Y8
   ```

   H = 1 if the sprite pattern is 4-bit
   N6 = 7th pattern bit if the sprite pattern is 4-bit
   T = 0 if relative sprites are composite type else 1 for unified type
   XX = Magnification in the X direction (00 = 1×, 01 = 2×, 10 = 4×4, 11 = 8×)
   YY = Magnification in the Y direction (00 = 1×, 01 = 2×, 10 = 4×, 11 = 8×)
   Y8 = Ninth bit of the sprite's Y coordinate
   H,N6 must not equal 0,1 as this combination is used to indicate a relative sprite.

B. Relative Sprite, Composite Type

   ```
   0 1 N6 X X Y Y PO
   ```

   N6 = 7th pattern bit if the sprite pattern is 4-bit
   XX = Magnification in the X direction (00 = 1×, 01 = 2×, 10 = 4×, 11 = 8×)
   YY = Magnification in the Y direction (00 = 1×, 01 = 2×, 10 = 4×, 11 = 8×)
   PO = 1 to indicate the sprite pattern number is relative to the anchor's

C. Relative Sprite, Unified Type

   ```
   0 1 N6 0 0 0 0 PO
   ```

   N6 = 7th pattern bit if the sprite pattern is 4-bit
   PO = 1 to indicate the sprite pattern number is relative to the anchor's

The display surface for sprites is 320 × 256. The X coordinate of the sprite is nine bits, ranging over 0-511, and the Y coordinate is optionally nine bits again ranging over 0-511 or is eight bits ranging over 0-255. The full extent 0-511 wraps on both axes, meaning a sprite 16 pixels wide plotted at X coordinate 511 would see its first pixel not displayed (coordinate 511) and the following pixels displayed in coordinates 0-14.

The full display area is visible in VGA. However, the HDMI display is vertically shorter so the top eight pixel rows (Y = 0-7) and the bottom eight pixel rows (Y = 248-255) will not be visible on an HDMI display.

Sprites can be fully described by sprite attributes 0-3 if the E bit in sprite attribute 3 is zero. These sprites are compatible with the original sprite module from core versions prior to 2.00.26.

If the E bit is set then a fifth sprite attribute, sprite attribute 4, becomes active. This attribute introduces scaling, 4-bit patterns, and relative sprites. Scaling is self-explanatory and 4-bit patterns were described in the last section. Relative sprites are described in the next section.

### 2.6.4 Relative Sprites

Normal sprites (sprites that are not relative) are known as anchor sprites. As the sprite module draws sprites in the order 0-127 (there are 128 sprites), it internally stores characteristics of the last anchor sprite seen. If following sprites are relative, they inherit some of these characteristics, which allows relative sprites to have, among other things, coordinates relative to the anchor. This means moving the anchor sprite also causes its relatives to move with it.

There are two types of relative sprites supported known as "Composite Sprites" and "Unified Sprites". The type is determined by the anchor in the T bit of sprite attribute 4.

A. Composite Sprites
   The sprite module records the following information from the anchor:
   - Anchor.visible
   - Anchor.Y
   - Anchor.palette_offset
   - Anchor.N (pattern number)
   - Anchor.H (indicates if the sprite uses 4-bit patterns)

   These recorded items are not used by composite sprites:
   - Anchor.rotate
   - Anchor.xmirror
   - Anchor.ymirror
   - Anchor.xscale

- Anchor.yscale

The anchor determines if all its relative sprites use 4-bit patterns or not.

The visibility of a particular relative sprite is the result of ANDing the anchor's visibility with the relative sprite's visibility. In other words, if the anchor is invisible then so are all its relatives.

Relative sprites only have 8-bit X and Y coordinates (the ninth bits are taken for other purposes). These are signed offsets from the anchor's X,Y coordinate. Moving the anchor moves all its relatives along with it.

If the relative sprite has its PR bit set in sprite attribute 2, then the anchor's palette offset is added to the relative sprite's to determine the active palette offset for the relative sprite. Otherwise the relative sprite uses its own palette offset as usual.

If the relative sprite has its PO bit set in sprite attribute 4, then the anchor's pattern number is added to the relative sprite's to determine the pattern used for display. Otherwise the relative sprite uses its own pattern number as usual. The intention is to supply a method to easily animate a large sprite by manipulating the pattern number in the anchor.

A composite sprite is like a collection of independent sprites tied to an anchor.

B. Unified Sprites

Unified sprites are a further extension of the composite type. The same information is recorded from the anchor and the same behaviour as described under composite sprites applies.

The difference is the collection of anchor and relatives is treated as if it were a single $16 \times 16$ sprite. The anchor's rotation, mirror, and scaling bits apply to all its relatives. Rotating the anchor causes all the relatives to rotate around the anchor. Mirroring the anchor causes the relatives to mirror around the anchor. The sprite hardware will automatically adjust X,Y coords and rotation, scaling and mirror bits of all relatives according to settings in the anchor.

Unified sprites should be defined as if all its parts are $16 \times 16$ in size with the anchor controlling the look of the whole.

A unified sprite is like a big version of an individual $16 \times 16$ sprite controlled by the anchor.

### 2.6.5 Programming Sprites

Sprites are created via three io registers and a nextreg interface.

Port $303B (W)

```
X S S S S S S S
N6 X N N N N N N
```

A write to this port has two effects.

One is it selects one of 128 sprites for writing sprite attributes via port $57.

The other is it selects one of 128 4-bit patterns in pattern memory for writing sprite patterns via port $5B. The N6 bit shown is the least significant in the 7-bit pattern number and should always be zero when selecting one of 64 8-bit patterns indicated by N.

Port $57 (W)

Once a sprite is selected via port $303B, its attributes can be written to this port one byte after another. Sprites can have either four or five attribute bytes and the internal attribute pointer will move onto the next sprite after those four or five attribute bytes are written. This means you can select a sprite via port $303B and write attributes for as many sequential sprites as desired. The attribute pointer will roll over from sprite 127 to sprite 0.

Port $5B (W)

Once a pattern number is selected via port $303B, the 256-byte or 128-byte pattern can be written to this port. The internal pattern pointer auto-increments after each write so as many sequential patterns as desired can be written. The internal pattern pointer will roll over from pattern 127 to pattern 0 (4-bit patterns) or from pattern 63 to pattern 0 (8-bit patterns) automatically.

Port $303B (R)

```
0 0 0 0 0 0 M C
```

M = 1 if the maximum number of sprites per line was exceeded
C = 1 if any two displayed sprites collide on screen
Reading this port automatically resets the M and C bits.

Besides the i/o interface, there is a nextreg interface to sprite attributes. The nextreg interface allows the copper to manipulate sprites and grants the program random access to a sprite's individual attribute bytes.

(R/W) \$34 (52) ⇒ Sprite Number
If the sprite number is in lockstep with io port \$303B (nextreg \$09 bit 4 is set)

bits 7 = Pattern address offset (Add 128 to pattern address)
bits 6-0 = Sprite number 0-127, Pattern number 0-63

Selects which sprite has its attributes connected to the following registers.

Effectively performs an out to port \$303B with the same value

Otherwise

bit 7 = Ignored
bits 6-0 = Sprite number 0-127

Selects which sprite has its attributes connected to the following registers. Bit 7 always reads back as zero.

This nextreg can operate in two modes.

If nextreg \$09 bit 4 is set, then this register is kept in lockstep with i/o port \$303B. A write to this nextreg is equivalent to a write to port \$303B and vice versa. In this mode, the i/o interface and nextreg interface are exactly equivalent.

If nextreg \$09 bit 4 is reset, then the nextreg interface is decoupled from i/o port \$303B. This nextreg is used to select a particular sprite 0-127 and this is completely independent from the sprite selected for the i/o interface. This independence allows the copper, for example, to manipulate different sprites than the cpu using the i/o interface.

(W) \$35 (53) ⇒ Sprite Attribute 0
(W) \$75 (117) ⇒ Sprite Attribute 0 with automatic post increment of Sprite Number

bits 7-0 = LSB of X coordinate

A write to nextreg \$75 also increases the selected sprite in nextreg \$34.

(W) \$36 (54) ⇒ Sprite Attribute 1
(W) \$76 (118) ⇒ Sprite Attribute 1 with automatic post increment of Sprite Number

bits 7-0 = LSB of Y coordinate

A write to nextreg \$76 also increases the selected sprite in nextreg \$34.

(W) \$37 (55) ⇒ Sprite Attribute 2
(W) \$77 (119) ⇒ Sprite Attribute 2 with automatic post increment of Sprite Number

bits 7-4 = Palette offset added to top 4 bits of sprite colour index
bit 3 = X mirror
bit 2 = Y mirror
bit 1 = Rotate
bit 0 = MSB of X coordinate

A write to nextreg \$77 also increases the selected sprite in nextreg \$34.

(W) \$38 (56) ⇒ Sprite Attribute 3
(W) \$78 (120) ⇒ Sprite Attribute 3 with automatic post increment of Sprite Number

bit 7 = Visible flag (1 = displayed)
bit 6 = Extended attribute (1 = Sprite Attribute 4 is active)
bits 5-0 = Pattern used by sprite (0-63)

A write to nextreg \$78 also increases the selected sprite in nextreg \$34.

(W) \$39 (57) ⇒ Sprite Attribute 4
(W) \$79 (121) ⇒ Sprite Attribute 4 with automatic post increment of Sprite Number
4-bit Sprites

bit 7 = H (1 = sprite uses 4-bit patterns)
bit 6 = N6 (0 = use the first 128 bytes of the pattern else use the last 128 bytes)
bit 5 = 1 if relative sprites are composite, 0 if relative sprites are unified Scaling
bits 4-3 = X scaling (00 = 1x, 01 = 2x, 10 = 4x, 11 = 8x)
bits 2-1 = Y scaling (00 = 1x, 01 = 2x, 10 = 4x, 11 = 8x)
bit 0 = MSB of Y coordinate

A relative mode is enabled if H,N6 = 01. The byte format for relative sprites is described above.
A write to nextreg \$79 also increases the selected sprite in nextreg \$34.

### 2.6.6 Global Control of Sprites

The following nextreg are also of interest for sprites.

(R/W) $09 (09) ⇒ Peripheral 4 setting:

bit 7 = Mono setting for AY 2 (1 = mono, 0 default)
bit 6 = Mono setting for AY 1 (1 = mono, 0 default)
bit 5 = Mono setting for AY 0 (1 = mono, 0 default)
bit 4 = Sprite id lockstep (1 = Nextreg $34 and IO Port $303B are in lockstep, 0 default)
bit 3 = Disables Kempston port ($DF) if set
bit 2 = Disables divMMC ports ($E3, $E7, $EB) if set
bits 1-0 = scanlines (0 after a PoR or Hard-reset)
00 = scanlines off
01 = scanlines 75%
10 = scanlines 50%
11 = scanlines 25%

Bit 4 determines if the i/o interface and nextreg interface operate in lockstep.

(R/W) $15 (21) ⇒ Sprite and Layers system

bit 7 = LoRes mode, 128 × 96 × 256 colours (1 = enabled)
bit 6 = Sprite priority (1 = sprite 0 on top, 0 = sprite 127 on top)
bit 5 = Enable sprite clipping in over border mode (1 = enabled)
bits 4-2 = set layers priorities:
Reset default is 000, sprites over the Layer 2, over the ULA graphics
000 – S L U
001 – L S U
010 – S U L
011 – L U S
100 – U S L
101 – U L S
110 – S(U+L) ULA and Layer 2 combined, colours clamped to 7
111 – S(U+L-5) ULA and Layer 2 combined, colours clamped to [0,7]
bit 1 = Over border (1 = yes)(Back to 0 after a reset)
bit 0 = Sprites visible (1 = visible)(Back to 0 after a reset)

Bit 0 must be set for sprites to be visible.
Bit 1 set allows sprites to be visible in the border area. When this bit is reset, sprites will not display outside the 256 × 192 area of the ULA display.

Bit 5 set enables clipping when sprites are visible in the border area. If reset, no clipping is applied and sprites will be visible in the full 320 × 256 space.

The sprite module draws sprites in the order 0-127 in each scanline. Bit 6 determines whether sprite 0 is topmost or sprite 127 is topmost.

Bits 4:2 determine layer priority and how sprites overlay or are obscured by other layers.

(R/W) $19 (25) ⇒ Clip Window Sprites

> bits 7-0 = Cood. of the clip window
>
> 1st write – X1 position
> 2nd write – X2 position
> 3rd write – Y1 position
> 4rd write – Y2 position

The values are 0,255,0,191 after a Reset
Reads do not advance the clip position

When the clip window is enabled for sprites in "over border" mode, the X coords are internally doubled and the clip window origin is moved to the sprite origin inside the border.

Sprites will only be visible inside the clipping window. When not in over-border mode (bit 1 of nextreg $15) the clipping window is given in ULA screen coordinates with 0,0 correspoding to the top left corner of the ULA screen. In over-border mode, the clipping window's origin is moved to the sprite coordinate origin 32 pixels to the left and 32 pixels above the ULA screen origin.

Regardless, sprite position is always in sprite coordinates with 32,32 corresponding to the top left corner of the ULA screen.

(W) $1C (28) ⇒ Clip Window control

> bits 7-4 = Reserved, must be 0
> bit 3 – reset the tilemap clip index
> bit 2 – reset the ULA/LoRes clip index.
> bit 1 – reset the sprite clip index.
> bit 0 – reset the Layer 2 clip index.

Can be used to reset nextreg $19.

See palette section on sprite palettes

(R/W) $4B (75) ⇒ Transparency index for sprites

bits 7-0 = Set the index value ($E3 after reset)

For 4-bit sprites only the bottom 4-bits are relevant.
Determines the transparent colour index used for sprites.

# Chapter 3

# Audio

## 3.1 Internal Speaker

The baseline sound of the ZX Spectrum was produced by toggling the Ear bit (bit 4) of $fe (254) The ULA port to produce 1-bit audio. It is enabled by bit 4 of Next register $08 (8). While this does work on the ZX Spectrum Next, there are other much better methods and this is only supported for backward compatibility.

Code:

```
;; enable internal speaker
ld bc,$243B
ld a,$08
out (c),a
ld bc,$253B
in a,(c)
or $10
out (c),a
```

## 3.2 Spectdrum/Convox

This is 8-bit D/A audio. It is enabled by setting bit 3 of Next register $08 (8). After that audio can be controlled by writing linear 8-bit unsigned values to port $df (223).

Code:

```
;; enable SpecDrum/Convox audio
ld bc,$243B
ld a,$08
out (c),a
ld bc,$253B
in a,(c)
or $08
out (c),a
```

## 3.3 Turbosound

TurboSound consists of the implementation of three AY-3-8912 chips. To enable TurboSound set bit 1 of Next Register $08 (8). Once enabled the sound chips and registers of the sound chips are selected using port $fffd (65533) TurboSound Next Control while the registers are accessed using $bffd () Sound Chip Register Access. To enable access to a particular chip write 111111xx to the control register where 01=AY1, 10=AY2, and 11=AY3. Access to particular registers of the selected chip is selected by writing the register number to the control register. You can then access a chip register using the access port.

Code:

```
;; enable TurboSound audio
ld bc,$243B
ld a,$08
out (c),a
ld bc,$253B
in a,(c)
or $02
out (c),a
```

Each of the three AY chips has three channels, A, B, and C whose mapping is controlled by bit 5 of Next register 0x08 (8).

- (R/W) 0x00 (0) ⇒ Channel A fine tune
- (R/W) 0x01 (1) ⇒ Channel A coarse tune (4 bits)
- (R/W) 0x02 (2) ⇒ Channel B fine tune

- (R/W) 0x03 (3) ⇒ Channel B coarse tune (4 bits)
- (R/W) 0x04 (4) ⇒ Channel C fine tune
- (R/W) 0x05 (5) ⇒ Channel C coarse tune (4 bits)
- (R/W) 0x06 (6) ⇒ Noise period (5 bits)
- (R/W) 0x07 (7) ⇒ Tone Enable
  - bit 5: Channel C tone enable (0=enable, 1=disable)
  - bit 4: Channel B tone enable (0=enable, 1=disable)
  - bit 3: Channel A tone enable (0=enable, 1=disable)
  - bit 2: Channel C noise enable (0=enable, 1=disable)
  - bit 1: Channel B noise enable (0=enable, 1=disable)
  - bit 0: Channel A noise enable (0=enable, 1=disable)
- (R/W) 0x08 (8) ⇒ Channel A amplitude
  - bit 4: 0=fixed amplitude, 1=use envelope generator (bits 0-3 ignored)
  - bits 0-3: value of fixed amplitude
- (R/W) 0x09 (9) ⇒ Channel B amplitude
  - bit 4: 0=fixed amplitude, 1=use envelope generator (bits 0-3 ignored)
  - bits 0-3: value of fixed amplitude
- (R/W) 0x0A (10) ⇒ Channel C amplitude
  - bit 4: 0=fixed amplitude, 1=use envelope generator (bits 0-3 ignored)
  - bits 0-3: value of fixed amplitude
- (R/W) 0x0B (11) ⇒ Envelope period fine
- (R/W) 0x0C (12) ⇒ Envelope period coarse
- (R/W) 0x0D (13) ⇒ Envelope shape
  - bit 3: Continue: 0=drop to amplitude 0 after 1 cycle, 1=use 'Hold' value
  - bit 2: Attack: 0=generator counts down, 1=generator counts up
  - bit 1: Alternate:
    - hold=0: 0=generator resets after each cycle, 1=generator reverses direction each cycle
    - hold=1: 0=hold final value, 1=hold initial value
  - bit 0: Hold: 0=cycle continuously, 1=perform one cycle and hold

# Chapter 4

# Memory Management

## 4.1 ZX Spectrum 128

128-style memory management can only alter the bank addressed at $c000 (16k-slot 4, or 8k-slot 7-8). The active 16k-bank at $c000 is selected by writing the 3 LSBs of the 16k-bank number to the bottom 3 bits of Memory Paging Control ($7FFD), and the 3 MSBs to the bottom 3 bits of Next Memory Bank Select ($DFFD). (The reason for the division is that the original Spectrum 128, having only 128k of memory, only needed 3 bits.)

On an unexpanded Next, this allows any 16k-bank to be paged in at $c000. On an expanded next, there are not enough bits available to access the banks at the bottom of the expanded memory, so Next memory management must be used to access these.

If you are using the standard interrupt handler or OS routines, then any time you write to Memory Paging Control ($7FFD) you should also store the value at $5B5C. Any time you write to Plus 3 Memory Paging Control ($1FFD) you should also store the value at $5B67. There is no corresponding system variable for the Next-only Next Memory Bank Select ($DFFD) and standard OS routines may not support the extended banks properly.

**128 Special Paging Mode** "Special paging mode" (also called "AllRam mode" or "CP/M mode") is enabled by writing a value with the LSB set to Plus 3 Memory Paging Control ($1FFD). Depending on the 3 low bits of this value a memory configuration is selected as follows:

Table 4.1: Special Paging Modes

| Bits 2-0 | Slot 1 | Slot 2 | Slot 3 | Slot 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 2 | 3 |
| 3 | 4 | 5 | 6 | 7 |
| 5 | 4 | 5 | 6 | 3 |
| 7 | 4 | 7 | 6 | 3 |

## 4.2 ZX Spectrum Next

The 8k-bank accessed in an 8k-slot is selected by writing the 8k-bank number to the bottom 7 bits of the 8 Next registers from ($50) upwards. $50 addresses 8k-slot 0, $51 addresses 8k-slot 1, and so on.

In addition, in 8k-slots 1 and 2 only, the ROM can be paged in by selecting the otherwise non-existent 8k-page $FF. Whether the high or the low 8k of the ROM is mapped is determined by which 8k-slot is used.

**Interactions between paging methods** Changes made in 128 style and Next style memory management are synchronized. The most recent change always has priority. This means that

using 128-style memory management to select a new 16k-bank in 16k-slot 4 will update the MMU registers for the two 8k-slots with the corresponding 8k-bank numbers. enabling AllRam mode will update all of the 8k-bank values with the appropriate 8k-slot numbers. These may then be overwritten using Next memory management without needing to alter the value at port $1FFD. Since the 128-style memory management ports are not readable, there is no synchronization applicable in the other direction.

**ROM paging and selection** $0000-$3fff is usually mapped to ROM. This area can only be fully remapped using Next memory management. ROM is not considered one of the numbered banks; it is mapped to the two 8k-banks by default, or by setting their 8k-bank numbers to 255.

The 128k Spectrum has 2 ROM pages. Which of these is mapped is selected by altering Bit 4 of Memory Paging Control ($7FFD). The +2a/+3 has 4 ROM pages; the extra bit needed to select between these is bit 2 of Plus 3 Memory Paging Control ($1FFD). This maintains compatibility with the original machines' ROM paging as long as the ROM is not paged out.

**Paging out ROM** ROM can be paged out by enabling AllRam mode, or by using Next memory management. Beware that some programs may assume that they can find ROM service routines at fixed addresses between $0000-$3fff. More importantly, if the default interrupt mode (IM 1) is set, the Z80 will jump the program counter to $0038 every frame expecting to find an interrupt handler there. If it does not, pain and suffering will likely result. DI is your friend. On the plus side, this does allow you to write your own interrupt handler without the nuisance of using IM 2.

**Layer 2 Switching** Layer 2 switching can allow any 16k-bank to be written to (but not read) in 16k-slot 1, by writing the 16k-bank number to Layer 2 RAM Page Register ($12) and then enabling Layer 2 paging by writing a value with the LSB set to Layer 2 Access Port ($123B).

Writing to this area will then write the appropriate area of memory, whereas reading from it will give the area mapped by other memory management.

**Screen** 16k-Bank 5 is the bank read by the ULA to determine what to show on screen. The ULA connects directly to the larger memory space ignoring mapping; the screen is always 16k-Bank 5, no matter where in memory it is (or if it is switched in at all). Setting bit 3 of Memory Paging Control ($7FFD) will have the ULA read from 16k-bank 7 (the "shadow screen") instead, which can be used as an alternate screen. Beware that this does not map 16k-bank 7 into RAM; to alter 16k-bank 7 it must be mapped by other means.

# Chapter 5

# zxnDMA

February 25, 2019 Phoebus Dokos Off Hardware, Resources,

The ZX Spectrum Next DMA (zxnDMA)

## 5.1 Overview

The ZX Spectrum Next DMA (zxnDMA) is a single channel dma device that implements a subset of the Z80 DMA functionality. The subset is large enough to be compatible with common uses of the similar Datagear interface available for standard ZX Spectrum computers and compatibles. It also adds a burst mode capability that can deliver audio at programmable sample rates to the DAC device.

## 5.2 Accessing the zxnDMA

The zxnDMA is mapped to a single Read/Write IO Port 0x6B which is the same one used by the Datagear but unlike the Datagear it doesn't also map itself to a second port 0x0B similar to the MB-02 interface.

`PORT $6b: zxnDMA`

## 5.3 Description

The normal Z80 DMA (Z8410) chip is a pipelined device and because of that it has numerous off-by-one idiosyncrasies and requirements on the order that certain commands should be carried out. These issues are not duplicated in the zxnDMA. You can continue to program the zxnDMA as if it is were a Z8410 DMA device but it can also be programmed in a simpler manner.

The single channel of the zxnDMA chip consists of two ports named A and B. Transfers can occur in either direction between ports A and B, each port can describe a target in memory or IO, and each can be configured to autoincrement, autodecrement or stay fixed after a byte is transferred.

A special feature of the zxnDMA can force each byte transfer to take a fixed amount of time so that the zxnDMA can be used to deliver sampled audio.

## 5.4 Modes of Operation

The zxnDMA can operate in a z80-dma compatibility mode.

The z80-dma compatibility mode is selected by setting bit 6 of nextreg $06. In this mode, all transfers involve length+1 bytes which is the same behaviour as the z80-dma chip. In zxn-dma mode, the transfer length is exactly the number of bytes programmed. This mode is mainly present to accommodate existing spectrum software that uses the z80-dma and for cp/m programs that may have a z80-dma option.

The zxnDMA can also operate in either burst or continuous modes.

Continuous mode means the DMA chip runs to completion without allowing the CPU to run. When the CPU starts the DMA, the DMA operation will complete before the CPU executes its next instruction.

Burst mode nominally means the DMA lets the CPU run if either port is not ready. This condition can't happen in the zxnDMA chip except when operated in the special fixed time transfer mode. In this mode, the zxnDMA will let the CPU run while it waits for the fixed time to expire between bytes transferred.

Note that there is no byte transfer mode as in the Z80 DMA.

## 5.5 Programming the zxnDMA

Like the Z80 DMA chip, the zxnDMA has seven write registers named WR0-WR6 that control the device. Each register WR0-WR6 can have zero or more parameters associated with it.

In a first write to the zxnDMA port, the write value is compared against a bitmask to determine which of the WR0-WR6 is the target. Remaining bits in the written value can contain data as well as a list of associated parameter bits. The parameter bits determine if further writes are expected to deliver parameter values. If there are multiple parameter bits set, the expected order of parameter values written is determined by parameter bit position from right to left (bit 0 through bit 7). Once all parameters are written, the zxnDMA again expects a regular register write selecting WR0-WR6.

The table X.Y describes the registers and the bitmask required to select them on the zxnDMA.

Table 5.1: zxnDMA Registers

| Register Group | Register Function Description | Bitmask | Notes |
|---|---|---|---|
| WR0 | Direction Operation and Port A configuration | 0XXXXXAA | AA must NOT be 00 |
| WR1 | Port A configuration | 0XXXX100 | |
| WR2 | Port B configuration | 0XXXX000 | |
| WR3 | Activation | 1XXXXX00 | It's best to use WR6 |
| WR5 | Ready and Stop configuration | 10XXX010 | |
| WR6 | Command Register | 1XXXXX11 | |

## 5.6 zxnDMA Registers

These are described below following the same convention used by Zilog for its DMA chip:

**WR0 – Write Register Group 0**

```
D7  D6  D5  D4  D3  D2  D1  D0  BASE REGISTER BYTE
 0   |   |   |   |   |   |   |
     |   |   |   |   |   0   0  Do not use
     |   |   |   |   |   0   1  Transfer (Prefer this for Z80 DMA compatibility)
     |   |   |   |   |   1   0  Do not use (Behaves like Transfer, Search on Z80
     |   |   |   |   |                         DMA)
     |   |   |   |   |   1   1  Do not use (Behaves like Transfer, Search/Trans-
     |   |   |   |   |                        fer on Z80 DMA)
     |   |   |   |   0 = Port B -> Port A (Byte transfer direction)
     |   |   |   |   1 = Port A -> Port B
     |   |   |   V
D7  D6  D5  D4  D3  D2  D1  D0  PORT A STARTING ADDRESS (LOW BYTE)
     |   |   V
D7  D6  D5  D4  D3  D2  D1  D0  PORT A STARTING ADDRESS (HIGH BYTE)
     |   V
D7  D6  D5  D4  D3  D2  D1  D0  BLOCK LENGTH (LOW BYTE)
     V
D7  D6  D5  D4  D3  D2  D1  D0  BLOCK LENGTH (HIGH BYTE)
```

Several registers are accessible from WR0. The first write to WR0 is to the base register byte. Bits D6:D3 are optionally set to indicate that associated registers in this group will be written next. The order the writes come in are from D3 to D6 (right to left). For example, if bits D6 and D3 are set, the next two writes will be directed to PORT A STARTING ADDRESS LOW followed by BLOCK LENGTH HIGH.

### WR1 – Write Register Group 1

```
D7  D6  D5  D4  D3  D2  D1  D0  BASE REGISTER BYTE
 0   |   |   |   |   1   0   0
     |   |   |   |
     |   |   |   0 = Port A is memory
     |   |   |   1 = Port A is IO
     |   |   |
     |   0   0 = Port A address decrements
     |   0   1 = Port A address increments
     |   1   0 = Port A address is fixed
     |   1   1 = Port A address is fixed
     |
     V
D7  D6  D5  D4  D3  D2  D1  D0  PORT A VARIABLE TIMING BYTE
 0   0   0   0   0   0   |   |
                         0   0 = Cycle Length = 4
```

```
                                  0   1 = Cycle Length = 3
                                  1   0 = Cycle Length = 2
                                  1   1 = Do not use
```

The cycle length is the number of cycles used in a read or write operation. The first cycle asserts signals and the last cycle releases them. There is no half cycle timing for the control signals.

**WR2 – Write Register Group 2**

```
D7  D6  D5  D4  D3  D2  D1  D0  BASE REGISTER BYTE
 0   |   |   |   |   0   0   0
     |   |   |   |
     |   |   |   0 = Port B is memory
     |   |   |   1 = Port B is IO
     |   |   |
     |   0   0 = Port B address decrements
     |   0   1 = Port B address increments
     |   1   0 = Port B address is fixed
     |   1   1 = Port B address is fixed
     |
     V
D7  D6  D5  D4  D3  D2  D1  D0  PORT B VARIABLE TIMING BYTE
 0   0   |   0   0   0   |   |
         |               0   0 = Cycle Length = 4
         |               0   1 = Cycle Length = 3
         |               1   0 = Cycle Length = 2
         |               1   1 = Do not use
         |
         V
D7  D6  D5  D4  D3  D2  D1  D0  ZXN PRESCALAR (FIXED TIME TRANSFER)
```

The ZXN PRESCALAR is a feature of the zxnDMA implementation. If non-zero, a delay will be inserted after each byte is transferred such that the total time needed for each transfer is determined by the prescalar. This works in both the continuous mode and the burst mode. If the DMA is operated in burst mode, the DMA will give up any waiting time to the CPU so that the CPU can run while the DMA is idle.

The rate of transfer is given by the formula "Frate = 875kHz / prescalar" or, rearranged, "prescalar = 875kHz / Frate". The formula is framed in terms of a sample rate (Frate) but Frate can be inverted to set a transfer time for each byte instead. The 875kHz constant is a nominal value assuming a 28MHz system clock; the system clock actually varies from this depending on the video timing selected by the user (HDMI, VGA0-6) so for complete accuracy the constant should be prorated according to documentation for nextreg $11.

In a DMA audio setting, selecting a sample rate of 16kHz would mean setting the prescalar value to 55. This sample period is constant across changes in CPU speed.

## WR3 – Write Register Group 3

```
D7  D6  D5  D4  D3  D2  D1  D0  BASE REGISTER BYTE
 1   |   0   0   0   0   0   0
     |
     1 = DMA Enable
```

The Z80 DMA defines more fields but they are ignored by the zxnDMA.

The two other registers defined by the Z80 DMA in this group on D4 and D3 are implemented by the zxnDMA but they do nothing.

It is preferred to start the DMA by writing an Enable DMA command to WR6.

## WR4 – Write Register Group 4

```
D7  D6  D5  D4  D3  D2  D1  D0  BASE REGISTER BYTE
 1   |   |   0   |   |   0   1
     |   |       |   |
     0   0 = Do not use (Behaves like Continuous mode, Byte mode on Z80 DMA)
     0   1 = Continuous mode
     1   0 = Burst mode
     1   1 = Do not use
                 |   |
                 |   V
D7  D6  D5  D4  D3  D2  D1  D0  PORT B STARTING ADDRESS (LOW BYTE)
```

```
                 |
                 V
D7  D6  D5  D4  D3  D2  D1  D0  PORT B STARTING ADDRESS (HIGH BYTE)
```

The Z80 DMA defines three more registers in this group through D4 that define interrupt behaviour. Interrups and pulse generation are not implemented in the zxnDMA nor are these registers available for writing.

### WR5 – Write Register Group 5

```
D7  D6  D5  D4  D3  D2  D1  D0  BASE REGISTER BYTE
 1   0   |   |   0   0   1   0
         |   |
         |   0 = /ce only
         |   1 = /ce & /wait multiplexed
         |
         0 = Stop on end of block
         1 = Auto restart on end of block
```

The /ce & /wait mode is implemented in the zxnDMA but is not currently used. This mode has an external device using the DMA's /ce pin to insert wait states during the DMA's transfer.

The auto restart feature causes the DMA to automatically reload its source and destination addresses and reset its byte counter to zero to repeat the last transfer when a previous one is finished.

### WR6 – Command Register

```
D7  D6  D5  D4  D3  D2  D1  D0  BASE REGISTER BYTE
 1   ?   ?   ?   ?   ?   1   1
     |   |   |   |   |
     1   0   0   0   0 = \$C3 = Reset
     1   0   0   0   1 = \$C7 = Reset Port A Timing
     1   0   0   1   0 = \$CB = Reset Port B Timing
     0   1   1   1   1 = \$BF = Read Status Byte
     0   0   0   1   0 = \$8B = Reinitialize Status Byte
     0   1   0   0   1 = \$A7 = Initialize Read Sequence
     1   0   0   1   1 = \$CF = Load
     1   0   1   0   0 = \$D3 = Continue
     0   0   0   0   1 = \$87 = Enable DMA
     0   0   0   0   0 = \$83 = Disable DMA
 +-- 0   1   1   1   0 = \$BB = Read Mask Follows
 |
D7  D6  D5  D4  D3  D2  D1  D0  READ MASK
 0   |   |   |   |   |   |   |
     |   |   |   |   |   |   V
D7  D6  D5  D4  D3  D2  D1  D0  Status Byte
     |   |   |   |   |   |
     |   |   |   |   |   V
D7  D6  D5  D4  D3  D2  D1  D0  Byte Counter Low
     |   |   |   |   |
     |   |   |   |   V
D7  D6  D5  D4  D3  D2  D1  D0  Byte Counter High
     |   |   |   |
     |   |   |   V
D7  D6  D5  D4  D3  D2  D1  D0  Port A Address Low
     |   |   |
     |   |   V
D7  D6  D5  D4  D3  D2  D1  D0  Port A Address High
     |   |
     |   V
D7  D6  D5  D4  D3  D2  D1  D0  Port B Address Low
     |
     V
D7  D6  D5  D4  D3  D2  D1  D0  Port B Address High
```

Unimplemented Z80 DMA commands are ignored.

Prior to starting the DMA, a LOAD command must be issued to copy the Port A and Port B addresses into the DMA's internal pointers. Then an ìEnable DMAî command is issued to start the DMA.

The ìContinueî command resets the DMA's byte counter so that a following ìEnable DMAî allows the DMA to repeat the last transfer but using the current internal address pointers. I.e. it continues from where the last copy operation left off.

Registers can be read via an IO read from the DMA port after setting the read mask. (At power up the read mask is set to $7f). Register values are the current internal dma counter values. So ìPort Address A Lowî is the lower 8-bits of Port A's next transfer address. Once the end of the read mask is reached, further reads loop around to the first one.

The format of the DMA status byte is as follows:

00E1101T

E is set to 0 if the total block length has been transferred at least once.

T is set to 1 if at least one byte has been transferred.

**Operating speed** The zxnDMA operates at the same speed as the CPU, that is 3.5MHz, 7MHz or 14MHz. This is a contended clock that is modified by the ULA and the auto-slowdown by Layer2.

Auto-slowdown occurs without user intervention if speed exceeds 7Mhz and the active Layer2 display is being generated (higher speed operation resumes when the active Layer2 display is not generated). Programmers do NOT need to account for speed differences regarding DMA transfers as this happens automatically.

Because of this, the cycle lengths for Ports A and B can be set to their minimum values without ill effects. The cycle lengths specified for Ports A and B are intended to selectively slow down read or write cycles for hardware that cannot operate at the DMA's full speed.

**The DMA and Interrupts** The zxnDMA cannot currently generate interrupts.

The other side of this is that while the DMA controls the bus, the Z80 cannot respond to interrupts. On the Z80, the nmi interrupt is edge triggered so if an nmi occurs the fact that it occurred is stored internally in the Z80 so that it will respond when it is woken up. On the other hand, maskable interrupts are level triggered. That is, the Z80 must be active to regularly sample the /INT line to determine if a maskable interrupt is occurring. On the Spectrum and the ZX Next, the ULA (and line interrupt) are only asserted for a fixed amount of time 30 cycles at 3.5MHz. If the DMA is executing a transfer while the interrupt is asserted, the CPU will not be able to see this and it will most likely miss the interrupt. In burst mode, the CPU will never miss these interrupts, although this may change if multiple channels are implemented.

## 5.7 Programming examples

A simple way to program the DMA is to walk down the list of registers WR0-WR5, sending desired settings to each. Then start the DMA by sending a LOAD command followed by an ENABLE_DMA command to WR6. Once more familiar with the DMA, you will discover that the amount of information sent can be reduced to what changes between transfers.

1. Assembly
   Short example program to DMA memory to the screen then DMA a sprite image from memory to sprite RAM, and then showing said sprite scroll across the screen.

```
;------------------------------------------------------------------------------
device zxspectrum48
;------------------------------------------------------------------------------
; DEFINE testing
;------------------------------------------------------------------------------
; DMA (Register 6)
;
;------------------------------------------------------------------------------
;zxnDMA programming example
;------------------------------------------------------------------------------
;(c) Jim Bagley
;------------------------------------------------------------------------------
DMA_RESET equ $c3
DMA_RESET_PORT_A_TIMING equ $c7
DMA_RESET_PORT_B_TIMING equ $cb
DMA_LOAD equ $cf ; %11001111
DMA_CONTINUE equ $d3
DMA_DISABLE_INTERUPTS equ $af
DMA_ENABLE_INTERUPTS equ $ab
DMA_RESET_DISABLE_INTERUPTS equ $a3
DMA_ENABLE_AFTER_RETI equ $b7
DMA_READ_STATUS_BYTE equ $bf
DMA_REINIT_STATUS_BYTE equ $8b
DMA_START_READ_SEQUENCE equ $a7
DMA_FORCE_READY equ $b3
DMA_DISABLE equ $83
DMA_ENABLE equ $87
DMA_WRITE_REGISTER_COMMAND equ $bb
```

```
DMA_BURST equ %11001101
DMA_CONTINUOUS equ %10101101
ZXN_DMA_PORT equ $6b
SPRITE_STATUS_SLOT_SELECT equ $303B
SPRITE_IMAGE_PORT equ $5b
SPRITE_INFO_PORT equ $57
;-------------------------------------------------------------------------------

IFDEF testing
org $6000
ELSE
org $2000
ENDIF

start
ld hl,$0000
ld de,$4000
ld bc,$800
call TransferDMA ; copy some random data to the screen pointing
; to ROM for now, for the purpose of showing
; how to do a DMA copy.
ld a,0 ; sprite image number we want to update
ld bc,SPRITE_STATUS_SLOT_SELECT
out (c),a ; set the sprite image number
ld bc,1*256 ; number to transfer (1)
ld hl,testsprite ; from
call TransferDMASprite ; transfer to sprite ram

nextreg 21,1 ; turn sprite on. for more info on this check
; out https://www.specnext.com/tbblue-io-port-system/
ld de,0
ld (xpos),de ; set initial X position ( doesn't need it for
; this demo, but if you run the .loop again it
; will continue from where it was
ld a,$20
ld (ypos),a ; set initial Y position

.loop
ld a,0 ; sprite number we want to position
ld bc,SPRITE_STATUS_SLOT_SELECT
```

```
out (c),a
ld de,(xpos)
ld hl,(ypos) ; ignores H so doing this rather than
; ld a,(ypos):ld l,a
ld bc,(image) ; not flipped or palette shifted
call SetSprite

halt

ld de,(xpos)
inc de
ld (xpos),de
ld a,d
cp $01
jr nz,.loop ; if high byte of xpos is not 1 (right of
; screen )
ld a,e
cp $20+1
jr nz,.loop ; if low byte is not $21 just off the right of
; the screen, $20 is off screen but as the
; INC DE is just above and not updated sprite
; after it, it needs to be $21
xor a
ret ; return back to basic with OK

xpos dw 0 ; x position
ypos db 0 ; y position
; these next two BITS and IMAGE are swapped
; as bits needs to go into B register image
; db 0+$80 ; use image 0 (for the image we
; transfered)+$80 to set the sprite to active
bits db 0 ; not flipped or palette shifted

c1 = %11100000
c2 = %11000000
c3 = %10100000
c4 = %10000000
c5 = %01100000
c6 = %01000000
c7 = %00100000
```

```
c8 = %00000000

testsprite
db c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1
db c1,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c1
db c1,c2,c3,c3,c3,c3,c3,c3,c3,c3,c3,c3,c3,c3,c2,c1
db c1,c2,c3,c4,c4,c4,c4,c4,c4,c4,c4,c4,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c5,c5,c5,c5,c5,c5,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c6,c6,c6,c6,c6,c6,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c6,c7,c7,c7,c7,c6,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c6,c7,c8,c8,c7,c6,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c6,c7,c8,c8,c7,c6,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c6,c7,c7,c7,c7,c6,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c6,c6,c6,c6,c6,c6,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c5,c5,c5,c5,c5,c5,c5,c5,c4,c3,c2,c1
db c1,c2,c3,c4,c4,c4,c4,c4,c4,c4,c4,c4,c4,c3,c2,c1
db c1,c2,c3,c3,c3,c3,c3,c3,c3,c3,c3,c3,c3,c3,c2,c1
db c1,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c2,c1
db c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1,c1

;-------------------------------------------------
; de = X
; l = Y
; b = bits
; c = sprite image
SetSprite
push bc
ld bc,SPRITE_INFO_PORT
out (c),e ; Xpos
out (c),l ; Ypos
pop hl
ld a,d
and 1
or h
out (c),a
ld a,l:or $80
out (c),a ; image
ret

;---------------------------------
```

```
; hl = source
; de = destination
; bc = length
;-------------------------------
TransferDMA
di
ld (DMASource),hl
ld (DMADest),de
ld (DMALength),bc
ld hl,DMACode
ld b,DMACode_Len
ld c,ZXN_DMA_PORT
otir
ei
ret

DMACode db DMA_DISABLE
db %01111101 ; R0-Transfer mode, A -> B, write adress
; + block length
DMASource dw 0 ; R0-Port A, Start address
; (source address)
DMALength dw 0 ; R0-Block length (length in bytes)
db %01010100 ; R1-write A time byte, increment, to
; memory, bitmask
db %00000010 ; 2t
db %01010000 ; R2-write B time byte, increment, to
; memory, bitmask
db %00000010 ; R2-Cycle length port B
db DMA_CONTINUOUS ; R4-Continuous mode (use this for block
; transfer), write dest adress
DMADest dw 0 ; R4-Dest address (destination address)
db %10000010 ; R5-Restart on end of block, RDY active
; LOW
db DMA_LOAD ; R6-Load
db DMA_ENABLE ; R6-Enable DMA

DMACode_Len equ $-DMACode

;-----------------------------------------------------------------------------
; hl = source
; bc = length
; set port to write to with TBBLUE_REGISTER_SELECT
; prior to call
;------------------------------------------------------------------------------
TransferDMAPort
di
ld (DMASourceP),hl
ld (DMALengthP),bc
ld hl,DMACodeP
ld b,DMACode_LenP
ld c,ZXN_DMA_PORT
otir
ei
ret

DMACodeP db DMA_DISABLE
db %01111101 ; R0-Transfer mode, A -> B, write adress
; + block length
DMASourceP dw 0 ; R0-Port A, Start address (source address)
DMALengthP dw 0 ; R0-Block length (length in bytes)
db %01010100 ; R1-read A time byte, increment, to
; memory, bitmask
db %00000010 ; R1-Cycle length port A
db %01101000 ; R2-write B time byte, increment, to
; memory, bitmask
db %00000010 ; R2-Cycle length port B
db %10101101 ; R4-Continuous mode (use this for block
; transfer), write dest adress
dw $253b ; R4-Dest address (destination address)
db %10000010 ; R5-Restart on end of block, RDY active
; LOW
db DMA_LOAD ; R6-Load
db DMA_ENABLE ; R6-Enable DMA

DMACode_LenP equ $-DMACodeP
;------------------------------------------------------------------------------
; hl = source
; bc = length
;------------------------------------------------------------------------------
TransferDMASprite
```

```
di
ld (DMASourceS),hl
ld (DMALengthS),bc
ld hl,DMACodeS
ld b,DMACode_LenS
ld c,ZXN_DMA_PORT
otir
ei
ret

DMACodeS db DMA_DISABLE
db %01111101 ; R0-Transfer mode, A -> B, write adress
; + block length
DMASourceS dw 0 ; R0-Port A, Start address (source address)
DMALengthS dw 0 ; R0-Block length (length in bytes)
db %01010100 ; R1-read A time byte, increment, to
; memory, bitmask
db %00000010 ; R1-Cycle length port A
db %01101000 ; R2-write B time byte, increment, to
; memory, bitmask
db %00000010 ; R2-Cycle length port B
db %10101101 ; R4-Continuous mode (use this for block
; transfer), write dest adress
dw SPRITE_IMAGE_PORT ; R4-Dest address (destination address)
db %10000010 ; R5-Restart on end of block, RDY active
; LOW
db DMA_LOAD ; R6-Load
db DMA_ENABLE ; R6-Enable DMA
DMACode_LenS equ $-DMACodeS
;-------------------------------------------------------------------------
; de = dest, a = fill value, bc = lenth
;-------------------------------------------------------------------------
DMAFill
di
ld (FillValue),a
ld (DMACDest),de
ld (DMACLength),bc
ld hl,DMACCode
ld b,DMACCode_Len
ld c,ZXN_DMA_PORT
```

```
otir
ei
ret

FillValue db 22
DMACCode db DMA_DISABLE
db %01111101
DMACSource dw FillValue
DMACLength dw 0
db %00100100,%00010000,%10101101
DMACDest dw 0
db DMA_LOAD,DMA_ENABLE
DMACCode_Len equ $-DMACCode

;-------------------------------------------------------------------------------
; End of file
;-------------------------------------------------------------------------------

IFDEF testing
savesna "dmatest.sna",start
ELSE
fin
savebin "DMATEST",start,fin-start
ENDIF
```

# Chapter 6

# Copper and Display Timing

From: KevB (aka 9bitcolour)

**Introduction** The ZX Spectrum Next includes a co-processor named "COPPER". It functions in a similar way to the Copper found in the Commodore Amiga Agnus custom chip. It's role is to free the Z80 of tasks that require the writing of hardware registers at precise pixel co-ordinates.

**Overview** The ZX Spectrum Next COPPER has three instructions: NOOP, MOVE, WAIT.

NOOP is used to fine tune timing. MOVE writes data to a specific range of hardware registers. WAIT waits for a pixel position on the video display.

These instructions are stored in 2k (2048 BYTES) of dedicated write-only program RAM also known as a "Copper list".

Each instruction is 16 bits (WORD) in size allowing for a maximum of 1024 instructions to be stored in the program RAM. The COPPER uses an internal 10 bit program counter (PC) which wraps to zero at the end of the list. The PC can be reset to zero, this is the default value after a hard/soft reset.

The instructions are stored in big endian format and transferred to the 2k program RAM using the Z80 or DMA (bits 15..8 followed by bits 7..0).

Three write-only hardware registers control access to the program RAM as well as the operating modes.

System performance is not affected when the COPPER is executing instructions.

The hardware registers and COPPER program RAM are not connected to the main memory BUS. The overall design of this system together with the use of alternate clock edges means that contention between the COPPER, Z80 and DMA has been eliminated.

The COPPER has a base clock speed of 13.5Mhz for HDMI and 14Mhz for VGA.

The bandwidth is around 14 million single cycle NOOP/WAIT instructions and 7 million two cycle MOVE instructions per second.

## 6.1 Timing

To fully understand the COPPER, you must first understand the display timing for each of the machines and video modes found in the ZX Spectrum Next.

There are several display timing configurations due to the four machine types, two refresh rates, two video systems (VGA/HDMI) and Timex HIRES mode.

Details of these timings are outlined in this chapter.

**Machines** The ZX Spectrum Next has four machine types (48k, 128k, Pentagon, and HDMI). The machine timing and HDMI determine the number of T-states per line which determines the base dot clock frequency and Z80/DMA clock speed.

This guide groups machine types by their timing for convenience. The HDMI video mode overrides the default machine timing so it is included as an extra machine type which does not exist in the official documentation.

**Display** The ZX Spectrum Next doesn't have video modes based on resolution that you would expect to find on graphics card based hardware. There is one fixed resolution of $256 \times 192$ which can be doubled to $512 \times 192$ in Timex HIRES mode. What it does have is the ability to set the refresh rate from 50Hz to 60Hz and horizontal dot clock. This in turn together with the VGA and HDMI timing affects the vertical line count giving several combinations in total.

VGA modes 0..6 are included as one single VGA mode as the internal machine timing is constant across those seven refresh rate steps.

More details can be found in Video modes.

**Resolution** There are two main horizontal resolutions: standard $256 \times 192$ and Timex HIRES $512 \times 192$. Details of LORES $128 \times 96$ are not included to simplify this guide.

The frame buffer height is fixed at 192 pixels and surrounded by a large border and overscan as well as horizontal and vertical blanking periods.

There are five vertical line counts: 261, 262, 311, 312, 320. Several pixels are hidden in the overscan and blanking periods beyond the visible border.

The result is $256 \times 192$ and $512 \times 192$ pixel resolutions with a large border.

The colour of the visible border beyond the frame buffer can be manipulated. Visual changes will not show during the overscan and blanking periods.

**Dot Clock** The dot clock on the ZX Spectrum Next runs at 13.5Mhz for HDMI and around 14Mhz for VGA. The COPPER clock runs at the same frequency as the dot clock. For v3.00 the copper runs at twice the frequency of the dot clock.

The number of dot clocks per line is calculated by multiplying the number of 3.5Mhz Z80 T-states per line by four. Example: 228Ts * 4 = 912 dot clocks.

The number of dot clocks per second is calculated by the following:

T-states per line * 4 * line count * refresh rate

In standard $256 \times 192$ resolution the duration of one pixel is two dot clocks. In Timex HIRES $512 \times 192$ resolution the duration of one pixel is one dot clock.

Details of the dot clock counts can be found in tables 5.1 and 5.2.

**Coordinates** The top left pixel of the frame buffer is line 0 and horizontal dot clock 0. This is also known as "0,0".

Table 6.1: Vertical Line Counts and Dot Clock Combinations

| System | Lines | Clocks |
|---|---|---|
| 48K VGA 50Hz | 312 | 224.0 * 4 = 896 |
| 128K VGA 50Hz | 311 | 228.0 * 4 = 912 |
| PENTAGON VGA 50Hz | 320 | 224.0 * 4 = 896 |
| 48K VGA 60Hz | 262 | 224.0 * 4 = 896 |
| 128K VGA 60Hz | 261 | 228.0 * 4 = 912 |
| HDMI 50Hz | 312 | 216.0 * 4 = 864 |
| HDMI 60Hz | 262 | 214.5 * 4 = 858 |

Table 6.2: Dot Clocks per Second

| System | Lines | Clocks | Freq |
|---|---|---|---|
| 48K VGA 50Hz | 312 | 13 977 600 | 14.0Mhz (28Mhz) |
| 128K VGA 50Hz | 311 | 14 181 600 | 14.2Mhz (28Mhz) |
| PENTAGON VGA 50Hz | 320 | 14 336 000 | 14.3Mhz (28Mhz) |
| 48K VGA 60Hz | 262 | 14 085 120 | 14.1Mhz (28Mhz) |
| 128K VGA 60Hz | 261 | 14 281 920 | 14.3Mhz (28Mhz) |
| HDMI 50Hz | 312 | 13 478 400 | 13.5Mhz (27Mhz) |
| HDMI 60Hz | 262 | 13 487 760 | 13.5Mhz (27Mhz) |

The bottom right pixel of the frame buffer in standard $256 \times 192$ resolution is line 191 and horizontal dot clocks 510+511.

The bottom right pixel of the frame buffer in Timex HIRES $512 \times 192$ resolution is line 191 and horizontal dot clock 511.

The line one pixel above the frame buffer is the last line of the video frame and equal to the total line count minus one (312-1 for example).

The line one pixel below the frame buffer is line 192.

The COPPER horizontal dot clock compare is locked to every eight pixels in standard $256 \times 192$ resolution and every sixteen pixels in Timex HIRES $512 \times 192$ resolution. The NOOP instruction can be used to fine tune timing in single dot clock steps.

**Compare** The COPPER uses a 9 bit vertical line compare allowing it to handle the various line counts.

The COPPER horizontal compare is 6 bits meaning that it can wait for 64 positions across each line. The range of this value is limited by the machine timing as that determines the number of dot clocks per line.

Table 6.3: Maximum Horizontal COPPER Compare

| System | Max |
|---|---|
| HDMI | 52 |
| Pentagon | 54 |
| 48k | 54 |
| 128k | 55 |

Each horizontal compare is in steps of 16 dot clocks to cover the full range across a raster line.

16 dot clocks = 4 pixels in lo $128 \times 96$ resolution

16 dot clocks = 8 pixels in standard $256 \times 192$ resolution

16 dot clocks = 16 pixels in high $512 \times 192$ resolution

There is some slack to consider after the maximum horizontal compare value. The slack is calculated using the following:

dot clocks per line - maximum horizontal compare * 16

Table 6.4: Slack Dot Clocks After Maximum Compare

| clocks/line | | slack |
|---|---|---|
| 858 | (52 * 16 = 832) | 26 dot clocks |
| 864 | (52 * 16 = 832) | 32 dot clocks |
| 896 | (54 * 16 = 864) | 32 dot clocks |
| 912 | (55 * 16 = 880) | 32 dot clocks |

Table 5.5 provides details of the horizontal display, left/right border, blanking and COPPER dot clock/pixel position compare values:

Table 5.6 provides a detailed list of vertical display, top/bottom border and blanking as well as maximum COPPER line compare. It also provides the ULA VBLANK interrupt line number.

Note: The HDMI overscan and blanking period is larger than that of a VGA monitor which can auto-adjust alignment. The following data is based on visible results from various monitors thus subject to refinement.

Pixels are visible during DISPLAY/BORDER and hidden during BLANKING.

Table 6.5: Horizontal Timing

| Compare | Standard | Timex | HDMI | 48k | 128k | Pentagon |
|---|---|---|---|---|---|---|
| 0-31 | 0-255 | 0-511 | Display | Display | Display | Display |
| 32-36 | 256-295 | 512-591 | R-Border | R-Border | R-Border | R-Border |
| 37 | 296-303 | 592-607 | R-Border | R-Border | Blanking | Blanking |
| 38-48 | 304-391 | 608-783 | Blanking | Blanking | Blanking | Blanking |
| 49 | 392-399 | 784-799 | L-Border | Blanking | Blanking | L-Border |
| 50-52 | 400-423 | 800-847 | L-Border | L-Border | L-Border | L-Border |
| 53-54 | 424-439 | 848-879 | – | L-Border | L-Border | L-Border |
| 55 | 440-447 | 880-895 | – | – | L-Border | – |

– Dot clock compare is out of range.

Table 6.6: Vertical Timing

| Line | HDMI 50Hz | HDMI 60Hz | 48k 50Hz | 48k 60Hz | 128k 50Hz | 128k 60Hz | Pentagon |
|---|---|---|---|---|---|---|---|
| 0-191 | Display | Display | Display | Display | Display | Display | Display |
| 192-211 | B-Border | B-Border | B-Border | B-Border | B-Border | B-Border | B-Border |
| 212-224 | B-Border | Blanking | B-Border | B-Border | B-Border | B-Border | B-Border |
| 225-231 | B-Border | Blanking | B-Border | Blanking | B-Border | Blanking | B-Border |
| 232-238 | Blanking | Blanking | B-Border | Blanking | B-Border | Blanking | B-Border |
| 239 | Blanking | Blanking | B-Border | T-Border | B-Border | T-Border | B-Border* |
| 240 | Blanking | Blanking | B-Border | T-Border | B-Border | T-Border | B-Border |
| 241-244 | Blanking | Blanking | B-Border | T-Border | B-Border | T-Border | Blanking |
| 245-247 | Blanking | T-Border | B-Border | T-Border | B-Border | T-Border | Blanking |
| 248 | Blanking | T-Border | B-Border* | T-Border | B-Border* | T-Border | Blanking |
| 249-255 | Blanking | T-Border | Blanking | T-Border | Blanking | T-Border | Blanking |
| 255 | Blanking | T-Border | Blanking | T-Border | Blanking | T-Border | T-Border |
| 256 | Blanking* | T-Border | Blanking | T-Border | Blanking | T-Border | T-Border |
| 257-260 | Blanking | T-Border | Blanking | T-Border | Blanking | T-Border | T-Border |
| 261 | Blanking | T-Border | Blanking | T-Border | Blanking | – | T-Border |
| 262 | Blanking | – | Blanking | – | Blanking | – | T-Border |
| 263-271 | Blanking | – | T-Border | – | T-Border | – | T-Border |
| 272-310 | T-Border | – | T-Border | – | T-Border | – | T-Border |
| 311 | T-Border | – | T-Border | – | – | – | T-Border |
| 312-319 | – | – | – | – | – | – | T-Border |

– Line compare is out of range
* ULA VBLANK interrupt.

**Overscan** The visible area of the display can extend to resolutions exceeding 256 × 192.

The 50/60 Hz refresh rate mode dictates the vertical limit.

VGA and HDMI differ with VGA providing more visible pixels beyond the range of HDMI. Table 5.7 provides ideal extended pixel resolutions:

Maximum Extended VGA Resolutions

50Hz = 352 × 288 (standard 256 resolution)

60Hz = 352 × 240 (standard 256 resolution)

Table 5.8 provides COPPER horizontal position and vertical line compare parameters for ideal extended resolutions:

Table 6.7: Ideal Extended Resolutions for Both VGA and HDMI

| Freq | Resolution | Top | Bottom | Left | Right |
|---|---|---|---|---|---|
| 50Hz | 336x288 | 32 | 32 | 40 | 40 |
| 60Hz | 336x240 | 24 | 24 | 40 | 40 |

Table 6.8: Ideal Extended Resolution Display Parameters

| Timing | Video | Ref | Lines | Top | Bot | Left | Right | Ext | Res |
|---|---|---|---|---|---|---|---|---|---|
| 0/1 48k | VGA | 50Hz | 312 | 280 | 223 | 51.1 | 36.15 | 80x64 | 336x256 |
| 0/1 48k | VGA | 60Hz | 262 | 246 | 207 | 51.1 | 36.15 | 80x48 | 336x240 |
| 2/3 128k | VGA | 50Hz | 311 | 279 | 223 | 52.1 | 36.15 | 80x64 | 336x256 |
| 2/3 128k | VGA | 60Hz | 261 | 245 | 207 | 52.1 | 36.15 | 80x48 | 336x240 |
| 4 Pentagon | VGA | 50Hz | 320 | 288 | 223 | 51.1 | 36.15 | 80x64 | 336x256 |
| 0/1 48k | HDMI | 50Hz | 312 | 280 | 223 | 49.1 | 36.15 | 80x64 | 336x256 |
| 0/1 48k | HDMI | 60Hz | 262 | 246 | 207 | 48.11 | 36.15 | 80x48 | 336x240 |
| 2/3 128k | HDMI | 50Hz | 312 | 280 | 223 | 49.1 | 36.15 | 80x64 | 336x256 |
| 2/3 128k | HDMI | 60Hz | 262 | 246 | 207 | 48.11 | 36.15 | 80x48 | 336x240 |
| 4 Pentagon | HDMI | 50Hz | 312 | 280 | 223 | 49.1 | 36.15 | 80x64 | 336x256 |
| 4 Pentagon | HDMI | 60Hz | 262 | 246 | 207 | 48.11 | 36.15 | 80x48 | 336x240 |

TOP: Initial line of the extended top border area - see notes below*
BOT: Last line of the extended bottom border area - see notes below*
LEFT: First pixel of the extended left border area - see notes below**
RIGHT: Last pixel of the extended right border area - see notes below**
* Line compare value for MOVE (bits 8..0).
** The integer part is the horizontal value for MOVE (bits 14..9).
** The fractional part is specified in dot clocks (NOOP instructions).

## 6.2 Instructions

This section describes the behaviour of the COPPER instructions as well as the bit definitions and execution time.

The three 16 bit COPPER instructions are comprised of the following bit definitions:

**NOOP** NOOP (no-operation) executes in one dot clock. It is useful for fine tuning timing, initialising COPPER RAM and 'NOP' out COPPER program instructions.

It can be used to align colour and display changes to half pixel positions in standard 256 × 192 resolution. Its duration is equal to one Timex HIRES pixel.

Table 6.9: Instruction Bit Definition

| Name | 15-8 | 7-0 | Clocks |
|---|---|---|---|
| NOOP | 00000000 | 00000000 | 1 |
| MOVE | 0RRRRRRR | DDDDDDDD | 2 |
| WAIT | 1HHHHHHV | VVVVVVVV | 1 |

| | |
|---|---|
| H | 6 bit horizontal dot clock compare |
| V | 9 bit vertical line compare |
| R | 7 bit Next register 0x00..0x7F |
| D | 8 bit data |

This guide uses the name 'NOOP' to avoid confusion with the Z80 opcode NOP.

**MOVE** MOVE executes in two dot clocks. It moves 8 bits of data into any of the Next hardware registers in the range $00 (0) .. $7F (127).

The WORD value $0000 is reserved for the NOOP instruction so no register access is carried out for that special case. Register $00 is read-only so not affected by the restriction of not being able to write zero to it.

This instruction can perform 7 million register writes per second for VGA and 6.75 million register writes per second for HDMI.

**WAIT** WAIT executes in one dot clock. It performs a compare with the current vertical line number and the current horizontal dot clock.

WAIT will hold until the current raster line matches the 9 bit value stored in bits 8..0. When the line compare matches, WAIT will still hold if the current horizontal dot clock is less than the value in bits 14..9.

This compare logic means that out of order vertical line compares will cause the COPPER to wait until the next video frame as the test is for an exact match of the line number. The COPPER will continue to the next instruction after an out of order horizontal pixel position compare as the test checks for the current dot clock being greater than or equal to the compare value.

WAIT will stop the COPPER when a compare is made against an out of range vertical line or horizontal dot clock position as they will never occur

A standard way to terminate a COPPER program is to wait for line 511 and horizontal position 63. This encodes into the instruction WORD $FFFF.

The horizontal dot clock position compare includes an adjustment meaning that the compare completes three dot clocks early in standard 256 × 192 resolution and two dot clocks early in Timex HIRES 512 × 192 resolution. In practice, a pixel position can be specified with clocks to spare to write a register value before the pixel is displayed. This saves software having to auto-adjust positions to arrive early. It also means that a wait for 0,0 can affect the first pixel of the frame buffer before it is displayed and set the scroll registers without visual artefacts.

**Example** The following example provides a simple COPPER program to move data to a hardware register at two specific pixel positions. The BYTES for the program are listed in the left column:

```
         PAL8 equ   0x41            ; 8 bit palette hardware register

$80,$00        WAIT  0,0            ; wait for pixel position 0,0 (H,V)
$00,$00        NOOP                 ; fine tune timing by one dot clock
$41,$E0        MOVE  PAL8,11100000b ; write RED to palette register

$C0,$BF        WAIT  32,191         ; wait for pixel position 256,191
$00,$00        NOOP                 ; fine tune timing by one dot clock
$41,$00        MOVE  PAL8,00000000b ; write BLACK to palette register

$FF,$FF        WAIT  63,511         ; wait for an out of range position
```

## 6.3 Control

The COPPER is controlled by the following three write-only registers:

$60 (96) Copper data
$61 (97) Copper control LO BYTE
$62 (98) Copper control HI BYTE

The COPPER instructions are written one BYTE at a time to the program RAM using register $60 (Copper data).

An index system is used to select the destination write address within the 2K program RAM. Eleven bits are needed to represent the index. Registers $61 and $62 hold this 11 bit index.

The index increments each time one BYTE is written to register $60. The index wraps to zero when the last BYTE of program RAM is written.

The instruction data is normally written in big endian format although there is no rule stating that partial instruction BYTES cannot be written. It is safe to write to the COPPER program RAM while the COPPER is executing as long the instruction data written does not create a mall formed instruction which comprises of one half of the current executing instruction and one half the new instruction - this could result in unexpected behaviour.

The Z80 and DMA can be used to write the instruction data.

Writing to program RAM while the COPPER is running has no impact on system performance as the RAM is contention free. COPPER timing is not affected by the Z80 or DMA writing to the program RAM. Program RAM is write-only.

The contents of the 2k program RAM are preserved during a hard/soft reset.

Register $61 holds the lower 8 bits of the index. Register $62 holds the upper 3 bits of the index as well as two control bits which set the COPPER operating mode.

Table 6.10: Register Bit Definitions

| Reg | 7-0 | Description |
|---|---|---|
| 0x60 | DDDDDDDD | BYTE data to write to COPPER program RAM |
| 0x61 | IIIIIIII | Program RAM index 7..0 |
| 0x62 | CC000III | Program RAM index 10..8 and control bits |

D 8 bit data
I 11 bit index
C 2 bit control

The COPPER has an internal 10 bit program counter (PC). Each instruction advances the program counter by one after completion. The program counter wraps to zero after the last instruction at location 1023. This causes the copper list to loop.

The program counter defaults to zero during a hard/soft reset.

The control bits require a change to update the operating mode. This feature preserves COPPER operation when setting the program RAM index address.

The program counter is preserved when stopping the COPPER. Two of the four control settings reset the internal PC to zero.

Table 5.11 describes the control bits:

Table 6.11: Control Mode Definitions

| Name | CC | Description |
|---|---|---|
| STOP | 00 | STOP COPPER |
| RESET | 01 | RESET PC and start COPPER |
| START | 10 | START COPPER |

* The control mode names used in this guide differ from the official names.

Here is a detailed description of the control bits:

**STOP** This is the default operating mode set during a hard/soft reset. The COPPER is idle in this state and will STOP if currently executing when entering this mode. It is safe to write to any location within the 2K program RAM when the COPPER is stopped.

Entering STOP mode preserves the internal program counter so that the COPPER may continue when restarted.

**RESET** The program counter is RESET to zero when entering this mode. The COPPER is started if idle otherwise entering this mode acts as a jump to location zero when the COPPER is running.

**START** Entering this mode causes an idle COPPER to start executing instructions from the current program counter. Entering this mode while the COPPER is running has no effect other than to disable FRAME mode if active.

**FRAME** The program counter is RESET to zero when entering this mode. The COPPER is started if idle otherwise entering this mode acts as a jump to location zero when the COPPER is running.

Entering this state enables FRAME mode. The program counter will be reset to zero each frame at 0,0.

## 6.4 Configuration

Hardware registers provide timing and configuration data allowing software to build and configure COPPER programs that function correctly across the various video modes and machine types. It is not essential to detect the machine type but it should be noted that software should not assume that it is running on a specific machine as the COPPER hardware is available across all four machine types.

Three registers can be read to determine the machine configuration for Ts per line, dot clocks, refresh rate, line count and maximum horizontal dot clock/pixel position compare.

**Refresh Rate** The refresh rate must be taken into account and can change real-time so should be monitored and auto-configured when the COPPER is active as the line count will change with the refresh rate. This could lead to the COPPER waiting for lines that never occur.

Peripheral 1 setting register $05 (5)

> bit 2 = 50/60 Hz mode
>
> > 0 = 50Hz
> >
> > 1 = 60Hz * Pentagon 60Hz is not supported in VGA mode so always 50Hz

**Video Modes** The video mode can only be changed during the boot process so one initial read is required of this register during software start up phase.

The machine timing is identical for the seven VGA modes although the physical refresh rate of the video output speeds up for each mode in turn by roughly 1Hz. The internal timing of the machine remains constant and as close to the original hardware as possible. VGA is a perfect Amstrad ZX Spectrum 128k +3 for example as far as timing is concerned across the seven VGA modes.

The effect of this speed up means that mode 0 will execute in one second of time whereas mode 6 will execute in a shorter time period. Mode 0 is as close to 50/60 Hz as possible where mode 6 is closer to 60/70 Hz. That would mean that one second of machine time for mode 6 will execute in 0.83 seconds of human time when running 50 frames per second at 60Hz.

The eighth mode (mode 7) is used for HDMI timing. Machine configuration is forced for this mode. Line counts, Ts and various other settings are set to meet the rigid HDMI timing specification. For mode 7, 50/60 Hz are rock solid but the original hardware timing loses Ts across all machines to meet HDMI display requirements.

Software that was previously written for specific hardware with hard-coded software timing loops may fail. This is one of the risks of coding timing loops counting Ts. We saw evidence of this with the release of the 1985 Sinclair ZX Spectrum 128k+ and the later Amstrad models as previous software written for the ZX Spectrum 48k/48k+ would fail when trying to display colour attribute and border effects as the number of Ts per line was changed from 224Ts (1982 original 48k) to 228Ts (128k models). The ZX Spectrum Next runs slower in HDMI mode. Demos may fail to display correctly and games may slow down although setting the Z80 to 7Mhz can solve the game slow down, demos should be run in VGA mode for maximum compatibility.

Video timing also affects audio output as the sample rate can vary depending on the output timing method.

The following undocumented register allows software to read the video timing mode:

Video timing register $11 (17)

bits 2-0 = Timing:
- 000 = Mode 0 (VGA)
- 001 = Mode 1 (VGA)
- 010 = Mode 2 (VGA)
- 011 = Mode 3 (VGA)
- 100 = Mode 4 (VGA)
- 101 = Mode 5 (VGA)
- 110 = Mode 6 (VGA)
- 111 = Mode 7 (HDMI) * Timing is forced to 216Ts 50Hz / 214.5Ts 60Hz

**Machine Type** The machine type register can be used to provide the number of Ts per line, line count, dot clock and maximum horizontal COPPER wait.

The dot clock (DC) is the number of Ts per line * 4.

The maximum horizontal COPPER wait (H) is in multiples of 16 clocks.

Video mode 7 (HMDI) overrides the timing.

The following list shows the various parameters that can be gained from reading the machine register combined with the refresh register and video mode bits:

Machine type register $03 (3)

> bits 6-4 = Timing:
> VGA 50Hz
>> 000 = 224.0Ts per line (312 lines) (DC=896) (H=54) (48k)
>> 001 = 224.0Ts per line (312 lines) (DC=896) (H=54) (48k)
>> 010 = 228.0Ts per line (311 lines) (DC=912) (H=55) (128k)
>> 011 = 228.0Ts per line (311 lines) (DC=912) (H=55) (128k)
>> 100 = 224.0Ts per line (320 lines) (DC=896) (H=54) (Pentagon)
>> 101 = RESERVED
>> 110 = RESERVED
>> 111 = RESERVED
>
> VGA 60Hz
>> 000 = 224.0Ts per line (262 lines) (DC=896) (H=54) (48k)
>> 001 = 224.0Ts per line (262 lines) (DC=896) (H=54) (48k)
>> 010 = 228.0Ts per line (261 lines) (DC=912) (H=55) (128k)
>> 011 = 228.0Ts per line (261 lines) (DC=912) (H=55) (128k)
>> 100 = 224.0Ts per line (320 lines) (DC=896) (H=54) (Pentagon)*
>> 101 = RESERVED
>> 110 = RESERVED
>> 111 = RESERVED
>
> HDMI 50Hz
>> 000 = 216.0Ts per line (312 lines) (DC=864) (H=52) (48k)
>> 001 = 216.0Ts per line (312 lines) (DC=864) (H=52) (48k)
>> 010 = 216.0Ts per line (312 lines) (DC=864) (H=52) (128k)
>> 011 = 216.0Ts per line (312 lines) (DC=864) (H=52) (128k)
>> 100 = 216.0Ts per line (312 lines) (DC=864) (H=52) (Pentagon)
>> 101 = RESERVED
>> 110 = RESERVED
>> 111 = RESERVED
>
> HDMI 60Hz
>> 000 = 214.5Ts per line (262 lines) (DC=858) (H=52) (48k)
>> 001 = 214.5Ts per line (262 lines) (DC=858) (H=52) (48k)
>> 010 = 214.5Ts per line (262 lines) (DC=858) (H=52) (128k)
>> 011 = 214.5Ts per line (262 lines) (DC=858) (H=52) (128k)

100 = 214.5Ts per line (262 lines) (DC=858) (H=52) (Pentagon)
101 = RESERVED
110 = RESERVED
111 = RESERVED

* Pentagon 60Hz is not supported in VGA mode so always 50Hz

**Summary** Table 5.13 provides a full list of video timing configuration data:

Table 6.12: Summary of Video Modes

| Timing | Video | Refresh | T-States | Clocks | Lines | Width | HRZ | Max | Slack | Adjust |
|---|---|---|---|---|---|---|---|---|---|---|
| 0/1 48k | VGA | 50Hz | 224 | 896 | 312 | 256 | 448 | 54 | 32 | -3 |
| 0/1 48k | VGA | 50Hz | 224 | 896 | 312 | 512 | 448 | 54 | 32 | -2 |
| 0/1 48k | VGA | 60Hz | 224 | 896 | 262 | 256 | 448 | 54 | 32 | -3 |
| 0/1 48k | VGA | 60Hz | 224 | 896 | 262 | 512 | 448 | 54 | 32 | -2 |
| 2/3 128k | VGA | 50Hz | 228 | 912 | 311 | 256 | 456 | 55 | 32 | -3 |
| 2/3 128k | VGA | 50Hz | 228 | 912 | 311 | 512 | 456 | 55 | 32 | -2 |
| 2/3 128k | VGA | 60Hz | 228 | 912 | 261 | 256 | 456 | 55 | 32 | -3 |
| 2/3 128k | VGA | 60Hz | 228 | 912 | 261 | 512 | 456 | 55 | 32 | -2 |
| 4 Pentagon | VGA | 50Hz | 224 | 896 | 320 | 256 | 448 | 55 | 32 | -3 |
| 4 Pentagon | VGA | 50Hz | 224 | 896 | 320 | 512 | 448 | 55 | 32 | -2 |
| 0/1 48k | HDMI | 50Hz | 216 | 864 | 312 | 256 | 432 | 52 | 32 | -3 |
| 0/1 48k | HDMI | 50Hz | 216 | 864 | 312 | 512 | 432 | 52 | 32 | -2 |
| 0/1 48k | HDMI | 60Hz | 214.5 | 858 | 262 | 256 | 429 | 52 | 26 | -3 |
| 0/1 48k | HDMI | 60Hz | 214.5 | 858 | 262 | 512 | 429 | 52 | 26 | -2 |
| 2/3 128k | HDMI | 50Hz | 216 | 864 | 312 | 256 | 432 | 52 | 32 | -3 |
| 2/3 128k | HDMI | 50Hz | 216 | 864 | 312 | 512 | 432 | 52 | 32 | -2 |
| 2/3 128k | HDMI | 60Hz | 214.5 | 858 | 262 | 256 | 439 | 52 | 26 | -3 |
| 2/3 128k | HDMI | 60Hz | 214.5 | 858 | 262 | 512 | 439 | 52 | 26 | -2 |
| 4 Pentagon | HDMI | 50Hz | 216 | 864 | 312 | 256 | 432 | 52 | 32 | -3 |
| 4 Pentagon | HDMI | 50Hz | 216 | 864 | 312 | 512 | 432 | 52 | 32 | -2 |
| 4 Pentagon | HDMI | 60Hz | 214.5 | 858 | 262 | 256 | 439 | 52 | 26 | -3 |
| 4 Pentagon | HDMI | 60Hz | 214.5 | 858 | 262 | 512 | 439 | 52 | 26 | -2 |

Chapter 7

# Interrupts

CPU processing of interrupts is enabled using the EI instruction and disabled with the DI instruction. Interrupts can happen at any time and should preserve register contents. If none of your code uses the alternate registers the EXX and EX AF,AF' instructions can make this faster and easier. Upon completion of your interrupt routine you should call the RTI instruction and re-enable interrupts.

IM0 – When an interrupt is received by the CPU it disables interrupts and executes the instruction placed on the bus by the interrupting device and (no known use on the Next) It is enabled with the IM0 instruction and enabling interrupts (EI).

IM1 – When an interrupt is received, the CPU disables interrupts and jumps to an interrupt handler at $0038 (normally in ROM). The ROM interrupt handler updates the frame counter and scans the keyboard. This is the default interrupt handling method for the ZX Spectrum and is probably the method to use if you don't need the ROMs for anything. It is enabled using the IM1 instruction and enabling interrupts.

IM2 – When the CPU receives an interrupt it disables interrupts and jumps to an interrupt routine starting at the contents of the jump table at I. The start of the interrupt routine is the contents of I*$100+bus and I*$100+bus+1. Most devices that can supply interrupts on the ZX Spectrum leave the data bus in a floating state. As a result the interpreted state of the data bus while generally $FF is not entirely predictable. The solution to place your interrupt routine at an address where the MSB and LSB are the same ($0101, $0202, . . . $FFFF) then place 257 copies of that value in a block starting at I*$100 (you can set the value of the I register).

Code:

```
;; my program
org $8000
;; enable interrupt mode im2
ld i,$fe
im2
ei
;; program body
;; interrupt routine
handler:
;; preserve registers used
;; handle interrupt
;; restore registers
ei
rti
;; jump to interrupt routine
org $fdfd
jp handler
;; im2 jump table
org $fe00 ; not actually legal
defs $101,$fd
```

# Chapter 8

# Raspberry Pi0 Acceleration

The Spectrum Next is known to have an header (with male pins) which can attach a Raspberry Pi Zero.

The Raspberry Pi 0 has a Broadcom BCM2835 SoC with an ARMv6 core, a Videocore 4 GPU, and its own 512 MB memory and HDMI output. It has its own SD card from which it boots. While it can traditionally run Linux, it can also be programmed ”bare metal” directly in assembly language, which may be more likely for this implementation.

It is not yet known how the Raspberry Pi 0 will interface with the Spectrum Next. It was originally used as an HDMI converter, but the HDMI functionality was moved onto the main board, makes this unnecessary.

It is known that the Raspberry Pi versions supporting Wi-Fi will not provide that Wi-Fi to the Next.

# Chapter 9

# Storage

## 9.1 NextOS

NextOS is one of the Operating Systems that is integrated into the Next. It's based on the +3E DOS which is in turn an extension of the +3 DOS as found on the ZX Spectrum +3. It provides access to the extra memory via integrated Ram Disk as well as SD cards. It also supports CP/M Plus file structures and partitions on the SD card (although support of CP/M for the Next is still being developed), FAT16 and FAT32 partitions, the latter with full 255 character Long File Names (LFN). NextOS additionally supports NextBASIC's 128k full screen editor and does not need USR0 mode to operate.

## 9.2 EsxDOS

esxDos is one of the operating systems that is integrated into the Next. It provides a posix-like api to access the SD card as disk and it provides a familiar basic interface to the disk from basic. The current version of ESXDOS is 0.8.6; a new version 0.9.x is being written for the Next. esxDos currently only supports the so-called USR0 mode in BASIC (single keyword/48k Basic)

# Appendix A

# Ports

February 25, 2019 Phoebus Dokos

Table A.1: ZX Spectrum Ports

| R | W | 16————————-0 | Port(hex) | Description | Disable |
|---|---|---|---|---|---|
| * | * | XXXX XXXX XXXX XXX0 | $fe | ULA | |
| * | * | XXXX XXXX 1111 1111 | $ff | Timex video/floating bus | Nextreg $08 bit 2 |
| | * | 0XXX XXXX XXXX XX01 | $7ffd | ZX Spectrum 128 memory | Port $7ffd bit 5 |
| | * | 01XX XXXX XXXX XX01 | $7ffd | ZX Spectrum 128 memory +3 only | Port $7ffd bit 5 |
| | * | 1101 XXXX XXXX XX01 | $dffd | ZX Spectrum 128 memory (precedence over AY) | Port $7ffd bit 5 |
| | * | 0001 XXXX XXXX XX01 | $1ffd | ZX Spectrum +3 memory | Port $7ffd bit 5 |
| * | | 0000 XXXX XXXX XX01 | | ZX Spectrum +3 floating bus | Port $7ffd bit 5 |
| * | * | 0010 0100 0011 1011 | $243b | NextREG Register Select | |
| * | * | 0010 0101 0011 1011 | $253b | NextREG data/value | |
| | * | 0001 0000 0011 1011 | $103b | i2c SCL (rtc) | |
| * | * | 0001 0001 0011 1011 | $113b | i2c SDA (rtc) | |
| * | * | 0001 0010 0011 1011 | $123b | Layer 2 | |
| * | * | 0001 0011 0011 1011 | $133b | UART tx | |
| * | * | 0001 0100 0011 1011 | $143b | UART rx | |
| * | * | 0001 0101 0011 1011 | $153b | UART control | |
| * | * | XXXX XXXX 0110 1011 | $6b | zxnDMA | |
| * | * | 11XX XXXX XXXX X101 | $fffd | AY reg | Nextreg $06 bit 0 |
| | * | 10XX XXXX XXXX X101 | $bffd | AY dat | Nextreg $06 bit 0 |
| | * | XXXX XXXX 0000 1111 | $0f | DAC A | Nextreg $08 bit 3 |
| | * | XXXX XXXX 1111 0001 | $f1 | DAC A (precedence over XXFD) | Nextreg $08 bit 3 |
| | * | XXXX XXXX 0011 1111 | $3f | DAC A | Nextreg $08 bit 3 |
| | * | XXXX XXXX 1101 1111 | $df | DAC A/C specdrum | Nextreg $08 bit 3 |
| | * | XXXX XXXX 0001 1111 | $1f | DAC B | Nextreg $08 bit 3 |
| | * | XXXX XXXX 1111 0011 | $f3 | DAC B | Nextreg $08 bit 3 |
| | * | XXXX XXXX 0100 1111 | $4f | DAC C | Nextreg $08 bit 3 |
| | * | XXXX XXXX 1111 1001 | $f9 | DAC C (precedence over XXFD) | Nextreg $08 bit 3 |
| | * | XXXX XXXX 0101 1111 | $5f | DAC D | Nextreg $08 bit 3 |
| | * | XXXX XXXX 1111 1011 | $fb | DAC D | Nextreg $08 bit 3 |
| | * | XXXX XXXX 1110 0111 | $e7 | SPI /CS (sd card/flash/rpi) | Nextreg $09 bit 2 |
| * | * | XXXX XXXX 1110 1011 | $eb | SPI /DATA | Nextreg $09 bit 2 |
| * | * | XXXX XXXX 1110 0011 | $e3 | divMMC Control | Nextreg $09 bit 2 |
| * | | XXXX 1011 1101 1111 | $fbdf | Kempston mouse x | Nextreg $09 bit 3 |
| * | | XXXX 1111 1101 1111 | $ffdf | Kempston mouse y | Nextreg $09 bit 3 |
| * | | XXXX 1010 1101 1111 | $fadf | Kempston mouse wheel/buttons | Nextreg $09 bit 3 |
| * | | XXXX XXXX 0001 1111 | $1f | Kempston joy 1 | Nextreg $05 |
| * | | XXXX XXXX 0011 0111 | $37 | Kempston joy 2 | Nextreg $05 |
| * | * | XXXX XXXX 0001 1111 | $1f | Multiface 1 disable | |
| * | | XXXX XXXX 1001 1111 | $9f | Multiface 1 enable | |
| * | * | XXXX XXXX 0011 1111 | $3f | Multiface 128 disable | |
| * | | XXXX XXXX 1011 1111 | $bf | Multiface 128 enable | |
| * | * | XXXX XXXX 1011 1111 | $bf | Multiface +3 disable | |
| * | | XXXX XXXX 0011 1111 | $3f | Multiface +3 enable | |
| * | * | 0011 0000 0011 1011 | $303b | Sprite slot/flags | |
| | * | XXXX XXXX 0101 0111 | $57 | Sprite attributes | |
| | * | XXXX XXXX 0101 1011 | $5b | Sprite pattern | |
| | * | 1011 1111 0011 1011 | $bf3b | ULAPlus register | |
| * | * | 1111 1111 0011 1011 | $ff3b | ULAPlus data | |

## A.1 8-bit

Port $0f (15) DAC A
Disable with bit 3 of Nextreg $08

Port $1f (31) Kempston Joystick 1
Disable with Nextreg $05

Port $1f (31) DAC B
Disable with bit 3 of Nextreg $08

Port $1f (31) Multiface 1 Disable

Port $37 (55) Kempston Joystick 2

Disable with Nextreg $05

Port $3f (63) DAC A
Disable with bit 3 of Nextreg $08

Port $3f (63) Multiface 128 Disable

Port $3f (63) Multiface +3 Enable

Port $4f (79) DAC C
Disable with bit 3 of Nextreg $08

Port $57 (87) Sprite Attributes

Port $5b (91) Sprite Pattern

Port $5f (95) DAC D
Disable with bit 3 of Nextreg $08

Port $6b (107) zxnDMA

Port $9f (159) Multiface 1 Enable

Port $bf (191) Multiface 128 Enable

Port $bf (191) Multiface +3 Disable

Port $df (223) DAC A/C SpecDrum
Disable with bit 3 of Nextreg $08

Port $e3 (227) divMMC Control
Disable with bit 2 of Nextreg $09

Port $e7 (231) SPI /CS (SD card, flash, rpi)
Disable with bit 2 of Nextreg $09

Port $eb (235) SPI /DATA (SD card, flash, rpi)
Disable with bit 2 of Nextreg $09

Port $f1 (241) DAC A (precedence over $xxfd)
Disable with bit 3 of Nextreg $08

Port $f3 (243) DAC B
Disable with bit 3 of Nextreg $08

Port $f9 (249) DAC C (precedence over $xxfd)
Disable with bit 3 of Nextreg $08

Port $fb (251) DAC D
Disable with bit 3 of Nextreg $08

Port $fe (254) ULA

bits 5-7: unused
bit 4: enable ear output
bit 3: enable mic output
bits 0-2: border colour

Port $ff (255) Timex Sinclair/floating bus
Disable with bit 2 to Nextreg $08

bit 7: memory paging (not on Next)
bit 6: disables generation of interrupts
bits 3-5: set hi-res mode colour combination
bits 0-2: screen mode
  000=normal ULA mode
  001=alternate ULA address
  010=hi-colour
  110=hi-res

## A.2 16-bit

Port $103b (4155) i2c SCL (rtc)

Port $113b (4411) i2c SDA (rtc)

Port $123b (4667) Layer 2

bits 6-7: Video RAM bank select
  00=first 16k
  01=middle 16k
  10=last 16k
  11=full 48k (3.00)
bits 4-5: Reserved (00)
bit 3: Shadow layer 2 select
bit 2: Enable layer 2 read paging (3.00)
bit 1: Layer 2 visible
bit 0: Enable layer 2 write paging

Port $133b (4923) UART tx
Read: UART Status

bits 7-3: Reserved (0)

bit 2: UART full
bit 1: UART transmit busy
bit 0: UART receive has data

Write: UART Transmit

Port $143b (5179) UART rx
Read: UART Receive
Write: UART Prescalar

bit 7:
 0=Bits 6-0 of prescalar
 1=Bits 13-7 of prescalar
bits 6-0: Prescalar low bits

Port $153b (5435) UART control (3.0)

bit 7: Reserved (0)
bit 6: UART select
 0=ESP
 1=Pi
bit 5: Reserved (0)
bit 4: Prescalar valid in this write
bit 3: Reserved (0)
bits 2-0: Bits 16-14 of prescalar

Port $243b (9275) Next Register Select
Write-only and is used to set the registry number, listed below.

Port $253b (9531) Next Register Data/Value
Used to access the registry value, the registry being either only a few bits or all bits only read, write or read/write, depending on the registry number set.
The nextreg instruction can be used to control Spectrum Next registers more directly.

Port $303b (12347) Sprite Slot/Flags
Write: Sprite Slot Select
select sprite slot for Sprite Attribute and Sprite Pattern ports which independently auto-increment
Read: Sprite status

bits 2-7: reserved
bit 1: Max sprites per line
bit 0: Collision flag

Port $7ffd (32765) ZX Spectrum 128 Memory
Disable with bit 5 port $7ffd

bits 6-7: reserved
bit 5: Lock memory paging (unlocks only on reset)
bit 4: ROM Select (low bit of ROM select for +2/+3)
bit 3: Shadow screen toggle
bits 0-2: Bank number for slot 4

Port $7ffe (32766) Keyboard 8 (read only)

bit 0: 'B'
bit 1: 'N'
bit 2: 'M'
bit 3: Symbol Shift
bit 4: Space

Port $bf3b (48955) ULAPlus register port (write only) (3.00)

Port $bffd (49149) AY (TurboSound Next) Data
Writes to selected register of selected sound chip

Port $BFFE (49150) Keyboard 7 (read only)

bit 0: 'H'
bit 1: 'J'
bit 2: 'K'
bit 3: 'L'
bit 4: Enter

Port $dffd (57341) ZX Spectrum 128 Memory (precedence over AY)
Disable with bit 5 port $7ffd
See Port $7ffd

Port $dffe (57342) Keyboard 6 (read only)

bit 0: 'Y'
bit 1: 'U'
bit 2: 'I'
bit 3: 'O'
bit 4: 'P'

Port $effe (61438) Keyboard 5 (read only)

bit 0: ‘6’
bit 1: ‘7’

bit 2: ‘8’
bit 3: ‘9’
bit 4: ‘0’

Port $f7fe (63486) Keyboard 4 (read only)

bit 0: ‘5’
bit 1: ‘4’
bit 2: ‘3’
bit 3: ‘2’
bit 4: ‘1’

Port $fadf (64223) Kempston Mouse Wheel/Buttons
Disable with bit 2 of Nextreg %09

Port $fbdf (64479) Kempston Mouse X
Disable with bit 2 of Nextreg $09

Port $fbfe (64510) Keyboard 3 (read only)

bit 0: ‘T’
bit 1: ‘R’
bit 2: ‘E’
bit 3: ‘W’
bit 4: ‘Q’

Port $fdfe (65022) Keyboard 2 (read only)

bit 0: ‘G’
bit 1: ‘F’
bit 2: ‘D’
bit 3: ‘S’
bit 4: ‘A’

Port $fefe (65278) Keyboard 1 (read only)

bit 0: ‘V’
bit 1: ‘C’
bit 2: ‘X’
bit 3: ‘Z’
bit 4: Caps Shift

Port $fbdf (64479) Kempston Mouse Y
Disable with bit 2 of Nextreg $09
Kempston Mouse Y coordinate (0-191)

Port $ff3b (65339) ULAPlus data port (3.00)

Port $fffd (65533) AY (Turbo Sound Next) Register
Select active sound chip/sound chip register
Select Chip

- bit 7: 1=select chip
- bit 6: Left audio enable
- bit 5: Right audio enable
- bits 2-4: reserved=111
- bits 0-1: Select active chip
  - 01=AY3
  - 10=AY2
  - 11=AY1 (default)

Select Register

- bit 7: 0=select register
- bits 4-6: reserved=000
- bits 0-3: register number

# Appendix B

# Registers

## B.1 ZX Spectrum Next Registers

February 25, 2019 Phoebus Dokos

TBBlue stores configuration state in a field of registers. These registers are accessible via two I/O ports or via the special nextreg instructions.

Port $243B (9275) is used to set the register number, listed below.

Port $253B (9531) is used to access the register value.

Some registers are accessible only during the initialization process.

2.00.27

(R) $00 (00) ⇒ Machine ID

00000001 = DE1A  
00000010 = DE2A  
00000101 = FBLABS  
00000110 = VTRUCCO  
00000111 = WXEDA  
00001000 = EMULATORS *  
00001010 = ZX Spectrum Next *  
00001011 = Multicore  
11111010 = ZX Spectrum Next Anti-brick *

* = Relevant for ZX Next machines & software

(R) $01 (01) ⇒ Core Version

bits 7-4 = Major version number
bits 3-0 = Minor version number

(see register $0E for sub minor version number)

(R/W) $02 (02) ⇒ Reset:

bits 7-3 = Reserved, must be 0
bit 2 = (R) Power-on reset (PoR)
bit 1 = (R/W) Reading 1 indicates a Hard-reset.
If written 1 causes a Hard Reset.
bit 0 = (R/W) Reading 1 indicates a Soft-reset.
If written 1 causes a Soft Reset.

(R/W) $03 (03) ⇒ Set machine type
A write to this register disables the IPL in config mode
($0000-$3FFF is mapped to RAM instead of the internal ROM)

bit 7 = (W) lock timing
= (R) register $44 second byte indicator
bits 6-4 = Timing:
(always writable if bit 7 is set)
000 or 001 = ZX 48K
010 = ZX 128K/+2 (Grey)
011 = ZX +2A-B/+3e/Next Native
100 = Pentagon 128K
bit 3 = Reserved, must be 0
bits 2-0 = Machine type (writable in config mode only):
000 = Config mode
001 = ZX 48K
010 = ZX 128K/+2 (Grey)
011 = ZX +2A-B/+3e/Next Native
100 = Pentagon 128K

(W) $04 (04) ⇒ Set page RAM, only in config mode (no IPL):

bits 7-5 = Reserved, must be 0
bits 4-0 = RAM bank mapped to $0000-$3FFF

(64 x 16k pages = 1024K, 0 after a PoR or Hard-reset)

(R/W) $05 (05) ⇒ Peripheral 1 setting:

bits 7-6 = joystick 1 mode (LSB)
bits 5-4 = joystick 2 mode (LSB)

bit 3 = joystick 1 mode (MSB)
bit 2 = 50/60 Hz mode (0 = 50Hz, 1 = 60Hz)(0 after a PoR or Hard-reset)
bit 1 = joystick 2 mode (MSB)
bit 0 = Enable Scandoubler (1 = enabled)(1 after a PoR or Hard-reset)

Joystick modes:

000 = Sinclair 2 (67890)
001 = Kempston 1 (port $1F)
010 = Cursor (56780)
011 = Sinclair 1 (12345)
100 = Kempston 2 (port $37)
101 = MD 1 (3 or 6 button joystick port $1F) item[] 110 = MD 2 (3 or 6 button joystick port $37)

(R/W) $06 (06) ⇒ Peripheral 2 setting:

bit 7 = Enable turbo mode (0 = disabled, 1 = enabled)
(0 after a PoR or Hard-reset)
bit 6 = DMA mode (0 = zxn dma, 1 = z80 dma)
(Only ZX Next board, 0 after a PoR or Hard-reset)
bit 5 = Enable Lightpen (1 = enabled)(0 after a PoR or Hard-reset)
bit 4 = DivMMC automatic paging (1 = enabled)(0 after a PoR or Hard-reset)
bit 3 = Enable Multiface (1 = enabled)(0 after a PoR or Hard-reset)
bit 2 = PS/2 mode (0 = keyboard, 1 = mouse)
(exchanges the keyboard/mouse pins on the PS/2 connector)
(0 after a PoR or Hard-reset)
bits 1-0 = Audio chip mode (00 = YM, 01 = AY, 1X = Disabled)

(R/W) $07 (07) ⇒ Turbo mode:

bit 1-0 = Turbo (00 = 3.5MHz, 01 = 7MHz, 10 = 14MHz)

(00 after a PoR or Hard-reset)

(R/W) $08 (08) ⇒ Peripheral 3 setting:

bit 7 = 128K paging enable (inverse of port $7ffd, bit 5)
Use "1" to disable the locked paging.
bit 6 = "1" to disable RAM contention. (0 after a reset)
bit 5 = Stereo mode (0 = ABC, 1 = ACB)(0 after a PoR or Hard-reset)
bit 4 = Enable internal speaker (1 = enabled)(1 after a PoR or Hard-reset)

bit 3 = Enable Specdrum/Covox (1 = enabled)(0 after a PoR or Hard-reset)
bit 2 = Enable Timex modes (1 = enabled)(0 after a PoR or Hard-reset)
bit 1 = Enable TurboSound (1 = enabled)(0 after a PoR or Hard-reset)
bit 0 = Reserved, must be 0

(R/W) $09 (09) ⇒ Peripheral 4 setting:

bit 7 = Mono setting for AY 2 (1 = mono, 0 default)
bit 6 = Mono setting for AY 1 (1 = mono, 0 default)
bit 5 = Mono setting for AY 0 (1 = mono, 0 default)
bit 4 = Sprite id lockstep (1 = Nextreg $34 and IO Port $303B are in lockstep, 0 default)
bit 3 = Disables Kempston port ($DF) if set
bit 2 = Disables divMMC ports ($E3, $E7, $EB) if set
bits 1-0 = scanlines (0 after a PoR or Hard-reset)
00 = scanlines off
01 = scanlines 75%
10 = scanlines 50%
11 = scanlines 25%

(R) $0E (14) ⇒ Core Version (sub minor number)
(see register $01 for the major and minor version number)

(R) $10 (16) ⇒ Anti-brick system

bits 7-2 = Reserved, must be 0
bit 1 = Button DivMMC (1 = pressed)
bit 0 = Button Multiface (1 = pressed)

(W) $10 (16) ⇒ Core Boot

bit 7 = Start selected core (1 = start)
bits 6-5 = Reserved, must be 0
bits 4-0 = Core ID 0-31 (writable in config mode only, default is 2)

(R/W) $11 (17) ⇒ Video Timing (writable in config mode only)

bits 7-3 = Reserved, must be 0
bits 2-0 = Mode (VGA = 0..6, HDMI = 7)
000 = Base VGA timing, clk28 = 28000000
001 = VGA setting 1, clk28 = 28571429
010 = VGA setting 2, clk28 = 29464286
011 = VGA setting 3, clk28 = 30000000

100 = VGA setting 4, clk28 = 31000000
101 = VGA setting 5, clk28 = 32000000
110 = VGA setting 6, clk28 = 33000000
111 = HDMI, clk28 = 27000000

50/60Hz selection depends on bit 2 of register $05

(R/W) $12 (18) ⇒ Layer 2 RAM bank

bits 7-6 = Reserved, must be 0
bits 5-0 = RAM bank (point to bank 8 after a Reset, NextZXOS modifies to 9)

(R/W) $13 (19) ⇒ Layer 2 RAM shadow bank

bits 7-6 = Reserved, must be 0
bits 5-0 = RAM bank (point to bank 11 after a Reset, NextZXOS modifies to 12)

(R/W) $14 (20) ⇒ Global transparency color

bits 7-0 = Transparency color value ($E3 after a reset)

(Note: this value is 8-bit, so the transparency is compared against only by the MSB bits of the final 9-bit colour)
(Note2: this only affects Layer 2, ULA and LoRes. Sprites use register $4B for transparency and tilemap uses nextreg $4C)

(R/W) $15 (21) ⇒ Sprite and Layers system

bit 7 = LoRes mode, 128 x 96 x 256 colours (1 = enabled)
bit 6 = Sprite priority (1 = sprite 0 on top, 0 = sprite 127 on top)
bit 5 = Enable sprite clipping in over border mode (1 = enabled)
bits 4-2 = set layers priorities:
Reset default is 000, sprites over the Layer 2, over the ULA graphics
000 - S L U
001 - L S U
010 - S U L
011 - L U S
100 - U S L
101 - U L S
110 - S(U+L) ULA and Layer 2 combined, colours clamped to 7
111 - S(U+L-5) ULA and Layer 2 combined, colours clamped to [0,7]
bit 1 = Over border (1 = yes)(Back to 0 after a reset)

bit 0 = Sprites visible (1 = visible)(Back to 0 after a reset)

(R/W) $16 (22) ⇒ Layer2 Offset X

bits 7-0 = X Offset (0-255)(0 after a reset)

(R/W) $17 (23) ⇒ Layer2 Offset Y

bits 7-0 = Y Offset (0-191)(0 after a reset)

(R/W) $18 (24) ⇒ Clip Window Layer 2

bits 7-0 = Coords of the clip window
1st write - X1 position
2nd write - X2 position
3rd write - Y1 position
4rd write - Y2 position

Reads do not advance the clip position
The values are 0,255,0,191 after a Reset

(R/W) $19 (25) ⇒ Clip Window Sprites

bits 7-0 = Cood. of the clip window
1st write - X1 position
2nd write - X2 position
3rd write - Y1 position
4rd write - Y2 position

The values are 0,255,0,191 after a Reset
Reads do not advance the clip position

When the clip window is enabled for sprites in "over border" mode, the X coords are internally doubled and the clip window origin is moved to the sprite origin inside the border.

(R/W) $1A (26) ⇒ Clip Window ULA/LoRes

bits 7-0 = Coord. of the clip window
1st write = X1 position
2nd write = X2 position
3rd write = Y1 position
4rd write = Y2 position

The values are 0,255,0,191 after a Reset
Reads do not advance the clip position

(R/W) $1B (27) ⇒ Clip Window Tilemap

bits 7-0 = Coord. of the clip window
1st write = X1 position
2nd write = X2 position
3rd write = Y1 position
4rd write = Y2 position

The values are 0,159,0,255 after a Reset
Reads do not advance the clip position
The X coords are internally doubled.

(W) $1C (28) ⇒ Clip Window control

bits 7-4 = Reserved, must be 0
bit 3 - reset the tilemap clip index
bit 2 - reset the ULA/LoRes clip index.
bit 1 - reset the sprite clip index.
bit 0 - reset the Layer 2 clip index.

(R) $1C (28) ⇒ Clip Window control
(may change)

bits 7-6 = Tilemap clip index
bits 5-4 = Layer 2 clip index
bits 3-2 = Sprite clip index
bits 1-0 = ULA clip index

(R) $1E (30) ⇒ Active video line (MSB)

bits 7-1 = Reserved, always 0
bit 0 = Active line MSB (Reset to 0 after a reset)

(R) $1F (31) = Active video line (LSB)

bits 7-0 = Active line LSB (0-255)(Reset to 0 after a reset)

(R/W) $22 (34) ⇒ Line Interrupt control

bit 7 = (R) INT flag, 1=During INT
(even if the processor has interrupt disabled)
bits 6-3 = Reserved, must be 0
bit 2 = If 1 disables original ULA interrupt (Reset to 0 after a reset)
bit 1 = If 1 enables Line Interrupt (Reset to 0 after a reset)
bit 0 = MSB of Line Interrupt line value (Reset to 0 after a reset)

(R/W) $23 (35) ⇒ Line Interrupt value LSB

bits 7-0 = Line Interrupt line value LSB (0-255)(Reset to 0 after a reset)

(W) $28 (40) ⇒ High address of Keymap

bits 7-1 = Reserved, must be 0
bit 0 = MSB address

(W) $29 (41) ⇒ Low address of Keymap

bits 7-0 = LSB adress

(W) $2A (42) ⇒ High data to Keymap

bits 7-1 = Reserved, must be 0
bit 0 = MSB data

(W) $2B (43) ⇒ Low data to Keymap
(writing this register the address is auto-incremented)

bits 7-0 = LSB data

(W) $2D (45) ⇒ SpecDrum port $DF / DAC A+C mirror

bits 7-0 = Data to be written to mono DAC

(this port can be used to generate mono audio using the copper)

(R/W) $2F (47) ⇒ Tilemap Offset X MSB

bits 7-2 = Reserved, must be 0
bits 1-0 = MSB X Offset

Meaningful Range is 0-319 in 40 char mode, 0-639 in 80 char mode

(R/W) $30 (48) ⇒ Tilemap Offset X LSB

bits 7-0 = LSB X Offset

Meaningful range is 0-319 in 40 char mode, 0-639 in 80 char mode

(R/W) $31 (49) ⇒ Tilemap Offset Y

bits 7-0 = Y Offset (0-255)

(R/W) $32 (50) ⇒ ULA / LoRes Offset X

bits 7-0 = X Offset (0-255)(Reset to 0 after a reset)

ULA can only scroll in multiples of 8 pixels so the lowest 3 bits have no effect at this time.
v3.00 system allows single pixel scrolling
LoRes scrolls in "half-pixels" at the same resolution and smoothness as Layer

2.

(R/W) $33 (51) ⇒ ULA / LoRes Offset Y

bits 7-0 = Y Offset (0-191)(Reset to 0 after a reset)

LoRes scrolls in ”half-pixels” at the same resolution and smoothness as Layer 2.

(R/W) $34 (52) ⇒ Sprite Number

If the sprite number is in lockstep with io port $303B (nextreg $09 bit 4 is set)

bits 7 = Pattern address offset (Add 128 to pattern address)
bits 6-0 = Sprite number 0-127, Pattern number 0-63

Selects which sprite has its attributes connected to the following registers.
Effectively performs an out to port $303B with the same value
Otherwise

bit 7 = Ignored
bits 6-0 = Sprite number 0-127
Selects which sprite has its attributes connected to the following registers.
Bit 7 always reads back as zero.

(W) $35 (53) ⇒ Sprite Attribute 0
(W) $75 (117) ⇒ Sprite Attribute 0 with automatic post increment of Sprite Number

bits 7-0 = LSB of X coordinate

(W) $36 (54) ⇒ Sprite Attribute 1
(W) $76 (118) ⇒ Sprite Attribute 1 with automatic post increment of Sprite Number

bits 7-0 = LSB of Y coordinate

(W) $37 (55) ⇒ Sprite Attribute 2
(W) $77 (119) ⇒ Sprite Attribute 2 with automatic post increment of Sprite Number

bits 7-4 = Palette offset added to top 4 bits of sprite colour index
bit 3 = X mirror
bit 2 = Y mirror
bit 1 = Rotate

bit 0 = MSB of X coordinate (palette offset indicator for relative sprites)

(W) \$38 (56) ⇒ Sprite Attribute 3
(W) \$78 (120) ⇒ Sprite Attribute 3 with automatic post increment of Sprite Number

bit 7 = Visible flag (1 = displayed)
bit 6 = Extended attribute (1 = Sprite Attribute 4 is active)
bits 5-0 = Pattern used by sprite (0-63)

(W) \$39 (57) ⇒ Sprite Attribute 4
(W) \$79 (121) ⇒ Sprite Attribute 4 with automatic post increment of Sprite Number
4-bit Sprites

bit 7 = H (1 = sprite uses 4-bit patterns)
bit 6 = N6 (0 = use the first 128 bytes of the pattern else use the last 128 bytes)
bit 5 = 1 if relative sprites are composite, 0 if relative sprites are unified Scaling
bits 4-3 = X scaling (00 = 1x, 01 = 2x, 10 = 4x, 11 = 8x)
bits 2-1 = Y scaling (00 = 1x, 01 = 2x, 10 = 4x, 11 = 8x)
bit 0 = MSB of Y coordinate

A relative mode is enabled if H,N6 = 01 that changes this byte format. Documentation for sprites can be found at specnext.com.
If this attribute is not active, the sprite behaves as if this byte is zero.

(R/W) \$40 (64) ⇒ Palette Index

bits 7-0 = Select the palette index to change the associated colour.

For the ULA only, INKs are mapped to indices 0-7, Bright INKS to indices 8-15, PAPERs to indices 16-23 and Bright PAPERs to indices 24-31.
In ULANext mode, INKs come from a subset of indices 0-127 and PAPERs come from a subset of indices 128-255. The number of active indices depends on the number of attribute bits assigned to INK and PAPER out of the attribute byte. The ULA always takes border colour from paper.

(R/W) \$41 (65) ⇒ Palette Value (8 bit colour)

bits 7-0 = Colour for the palette index selected by the register \$40.

(Format is RRRGGGBB - the lower blue bit of the 9-bit colour will be a logical OR of blue bits 1 and 0 of this 8-bit value.)

After the write, the palette index is auto-incremented to the next index if the auto-increment is enabled at reg $43. Reads do not auto-increment.

(R/W) $42 (66) ⇒ ULANext Attribute Byte Format

> bits 7-0 = Mask indicating which bits of an attribute byte are used to represent INK. The rest will represent PAPER.

(15 on reset)
The mask can only indicate a solid sequence of bits on the right side of the attribute byte (1, 3, 7, 15, 31, 63, 127 or 255).
INKs are mapped to base index 0 in the palette and PAPERs and border are mapped to base index 128 in the palette.
The 255 value enables the full ink colour mode making all the palette entries INK. PAPER and border both take on the fallback colour (nextreg $4A) in this mode.

(R/W) $43 (67) ⇒ Palette Control

> bit 7 = '1' to disable palette write auto-increment.
> bits 6-4 = Select palette for reading or writing:
>> 000 = ULA first palette
>> 100 = ULA second palette
>> 001 = Layer 2 first palette
>> 101 = Layer 2 second palette
>> 010 = Sprites first palette
>> 110 = Sprites second palette
>> 011 = Tilemap first palette
>> 111 = Tilemap second palette
>
> bit 3 = Select Sprites palette (0 = first palette, 1 = second palette)
> bit 2 = Select Layer 2 palette (0 = first palette, 1 = second palette)
> bit 1 = Select ULA palette (0 = first palette, 1 = second palette)
> bit 0 = Enabe ULANext mode if 1. (0 after a reset)

(R/W) $44 (68) ⇒ Palette Value (9 bit colour)
Two consecutive writes are needed to write the 9 bit colour

> 1st write:
>> bits 7-0 = RRRGGGBB
>
> 2nd write. If writing a L2 palette
>
>> bit 7 = 1 for L2 priority colour, 0 for normal
>> Priority colour will always be on top even on an SLU priority

arrangement. If you need the exact same colour on priority and non priority locations you will need to program the same colour twice changing bit 7 to 0 for the second colour
bits 6-1 = Reserved, must be 0
bit 0 = lsb B

If writing another palette
bits 7-1 = Reserved, must be 0
bit 0 = lsb B

After the two consecutives writes the palette index is auto-incremented if the auto-increment is enabled by reg $43.
Reads only return the 2nd byte and do not auto-increment.

(R/W) $4A (74) ⇒ Transparency colour fallback

bits 7-0 = Set the 8 bit colour used if all layers are transparent.

(black on reset = 0)

(R/W) $4B (75) ⇒ Transparency index for sprites

bits 7-0 = Set the index value ($E3 after reset)

For 4-bit sprites only the bottom 4-bits are relevant.

(R/W) $4C (76) ⇒ Transparency index for the tilemap

bits 7-4 = Reserved, must be 0
bits 3-0 = Set the index value ($F after reset)

(R/W) $50 (80) ⇒ MMU slot 0

bits 7-0 = Set a Spectrum RAM page at position $0000 to $1fff

(255 after a reset)
Pages can be from 0 to 223 on a fully expanded Next.
A 255 value causes the ROM to become visible.

(R/W) $51 (81) ⇒ MMU slot 1

bits 7-0 = Set a Spectrum RAM page at position $2000 to $3fff

(255 after a reset)
Pages can be from 0 to 223 on a full expanded Next.
A 255 value causes the ROM to become visible.

(R/W) $52 (82) ⇒ MMU slot 2

bits 7-0 = Set a Spectrum RAM page at position $4000 to $5fff

(10 after a reset)
Pages can be from 0 to 223 on a full expanded Next.

(R/W) \$53 (83) ⇒ MMU slot 3

bits 7-0 = Set a Spectrum RAM page at position \$6000 to \$7FFF

(Reset to 11 after a reset)
Pages can be from 0 to 223 on a full expanded Next.

(R/W) \$54 (84) ⇒ MMU slot 4

bits 7-0 = Set a Spectrum RAM page at position \$8000 to \$9FFF

(4 after a reset)
Pages can be from 0 to 223 on a full expanded Next.

(R/W) \$55 (85) ⇒ MMU slot 5

bits 7-0 = Set a Spectrum RAM page at position \$A000 to \$BFFF

(Reset to 5 after a reset)
Pages can be from 0 to 223 on a full expanded Next.

(R/W) \$56 (86) ⇒ MMU slot 6

bits 7-0 = Set a Spectrum RAM page at position \$C000 to \$DFFF

(0 after a reset)
Pages can be from 0 to 223 on a full expanded Next.

(R/W) \$57 (87) ⇒ MMU slot 7

bits 7-0 = Set a Spectrum RAM page at position \$E000 to \$FFFF

(1 after a reset)
Pages can be from 0 to 223 on a full expanded Next.

Writing to ports \$1FFD, \$7FFD and \$DFFD writes 255 to MMU0 and MMU1 and writes appropriate values to MMU6 and MMU7 to map in the selected 16k bank.
+3 special modes override the MMUs if used.

(W) \$60 (96) ⇒ Copper data

bits 7-0 = Byte to write to copper instruction memory.

Note that each copper instruction is two bytes long.
After a write, the index is auto-incremented to the next memory position.

(W) \$61 (97) ⇒ Copper control LO bit

bits 7-0 = Copper instruction memory address LSB.

(Index is set to 0 after a reset)

(W) $62 (98) ⇒ Copper control HI bit

bits 7-6 = Start control
00 = Copper fully stopped
01 = Copper start, execute the list from index 0, and loop to the start
10 = Copper start, execute the list from last point, and loop to the start
11 = Copper start, execute the list from index 0, and restart the list when the raster reaches position (0,0)
bits 2-0 = Copper instruction memory address MSB

(R/W) $68 (104) ⇒ ULA Control

bit 7 = 1 to disable ULA output
bit 6 = 0 to select the ULA colour for blending in SLU modes 6 & 7
= 1 to select the ULA/tilemap mix for blending in SLU modes 6 & 7
bits 5-1 = Reserved must be 0
bit 0 = 1 to enable stencil mode when both the ULA and tilemap are enabled

(if either are transparent the result is transparent otherwise the result is a logical AND of both colours)

(R/W) $6A (106) ⇒ LoRes Radistan Control (v3.00)

bits 7-6 = reserved
bit 5 = 1 Enable Radistan (16-colour)
bit 4 = Half of LoRes area to use for Radistan, xor with bit 0 of port $ff

- bits 3-0 = Radistsan palette offset

(R/W) $6B (107) ⇒ Tilemap Control

bit 7 = 1 to enable the tilemap
bit 6 = 0 for 40x32, 1 for 80x32
bit 5 = 1 to eliminate the attribute entry in the tilemap
bit 4 = palette select
bits 3-2 = Reserved set to 0
bit 1 = 1 to activate 512 tile mode
bit 0 = 1 to force tilemap on top of ULA

(R/W) \$6C (108) ⇒ Default Tilemap Attribute

> bits 7-4 = Palette Offset
> bit 3 = X mirror
> bit 2 = Y mirror
> bit 1 = Rotate
> bit 0 = ULA over tilemap

(bit 8 of the tile number if 512 tile mode is enabled)
Active tile attribute if bit 5 of nextreg \$6B is set.

(R/W) \$6E (110) ⇒ Tilemap Base Address

> bits 7-6 = Read back as zero, write values ignored
> bits 5-0 = MSB of address of the tilemap in Bank 5

The value written is an offset into Bank 5 allowing the tilemap to be placed at any multiple of 256 bytes.
Writing a physical MSB address in \$40-\$7f or \$c0-\$ff range is permitted.
The value read back should be treated as having a fully significant 8-bit value.

(R/W) \$6F (111) ⇒ Tile Definitions Base Address

> bits 7-6 = Read back as zero, write values ignored
> bits 5-0 = MSB of address of tile definitions in Bank 5

The value written is an offset into Bank 5 allowing tile definitions to be placed at any multiple of 256 bytes.
Writing a physical MSB address in \$40-\$7f or \$c0-\$ff range is permitted.
The value read back should be treated as having a fully significant 8-bit value.

(W) \$75 (117) ⇒ Sprite Attribute 0 with automatic post increment of Sprite Number
See nextreg \$35

(W) \$76 (118) ⇒ Sprite Attribute 1 with automatic post increment of Sprite Number
See nextreg \$36

(W) \$77 (119) ⇒ Sprite Attribute 2 with automatic post increment of Sprite Number
See nextreg \$37

(W) \$78 (120) ⇒ Sprite Attribute 3 with automatic post increment of Sprite

Number
See nextreg $38

(W) $79 (121) ⇒ Sprite Attribute 4 with automatic post increment of Sprite Number
See nextreg $39

(R/W) $90 (144) ⇒ Pi GPIO output enable 0 (3.00)
Control output enable for Pi GPIO pins 0-7

(R/W) $91 (145) ⇒ Pi GPIO output enable 1 (3.00)
Control output enable for Pi GPIO pins 8-15

(R/W) $92 (146) ⇒ Pi GPIO output enable 2 (3.00)
Control output enable for Pi GPIO pins 16-23

(R/W) $93 (147) ⇒ Pi GPIO output enable 3 (3.00)
Control output enable for Pi GPIO pins 24-27

(R/W) $98 (152) ⇒ Pi GPIO data 0 (3.00)
Set value of GPIO pin if enabled Pi GPIO pins 0-7

(R/W) $99 (153) ⇒ Pi GPIO data 1 (3.00)
Set value of GPIO pin if enabled Pi GPIO pins 8-15

(R/W) $9a (154) ⇒ Pi GPIO data 2 (3.00)
Set value of GPIO pin if enabled Pi GPIO pins 16-23

(R/W) $9b (155) ⇒ Pi GPIO data 3 (3.00)
Set value of GPIO pin if enabled Pi GPIO pins 24-27

(R/W) $a0 (160) ⇒ Pi peripheral enable (3.00)

bits 7-6 = Reserved (00)
bit 5 = RXTX (0=Pi hat, 1=Pi)
bit 4 = UART (0=14/15 GPIO, 1=14/15 UART)
bit 3 = I2C1 (0=2/3 GPIO, 1=2/3 I2C)
bits 2-1 = Reserved (00)
bit 0 = SPIO (0=7-11 GPIO, 1=7-11 SPI)

Control Pi GPIO function overlay

(W) $FF (255) ⇒ Debug LEDs (DE-1, DE-2 am Multicore only)

## B.2 AY-3-8192

## B.3 zxDMA

# Appendix C

# Extended Opcodes to Mnemonics

## C.1 Single Byte Opcodes

Table C.1: $00-$1F

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $00 | nop | nop | 1 | 4 | $10 | djnz x | – | 2 | 13/8 |
| $01 | ld bc,xx | lxi b,xx | 3 | 10 | $11 | ld de,xx | lxi d,xx | 3 | 10 |
| $02 | ld (bc),a | stax b | 1 | 7 | $12 | ld (de),a | stax d | 1 | 7 |
| $03 | inc bc | inx b | 1 | 6 | $13 | inc de | inx d | 1 | 6 |
| $04 | inc b | inr b | 1 | 4 | $14 | inc d | inr d | 1 | 4 |
| $05 | dec b | dcr b | 1 | 4 | $15 | dec d | dcr d | 1 | 4 |
| $06 | ld b,x | mvi b,x | 2 | 7 | $16 | ld d,x | mvi d,x | 2 | 7 |
| $07 | rlca | rlc | 1 | 4 | $17 | rla | ral | 1 | 4 |
| $08 | ex af,af' | – | 1 | 4 | $18 | jr x | – | 2 | 12 |
| $09 | add hl,bc | dad b | 1 | 11 | $19 | add hl,de | dad d | 1 | 11 |
| $0A | ld a,(bc) | ldax b | 1 | 7 | $1A | ld a,(de) | ldax d | 1 | 7 |
| $0B | dec bc | dcx b | 1 | 6 | $1B | dec de | dcx d | 1 | 6 |
| $0C | inc c | icr c | 1 | 4 | $1C | inc e | icr e | 1 | 4 |
| $0D | dec c | dcr c | 1 | 4 | $1D | dec e | dcr e | 1 | 4 |
| $0E | ld c,x | mvi c,x | 2 | 7 | $1E | ld e,x | mvi e,x | 2 | 7 |
| $0F | rrca | rrc | 1 | 4 | $1F | rra | rar | 1 | 4 |

Table C.2: $20-$3F

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $20 | jr nz,x | – | 2 | 12/7 | $30 | jr nc,x | – | 2 | 12/7 |
| $21 | ld hl,xx | lxi h,xx | 3 | 10 | $31 | ld sp,xx | lxi sp,xx | 3 | 10 |
| $22 | ld (xx),hl | shld xx | 3 | 16 | $32 | ld (xx),a | sta xx | 3 | 13 |
| $23 | inc hl | inx h | 1 | 6 | $33 | inc sp | inx sp | 1 | 6 |
| $24 | inc h | inr h | 1 | 4 | $34 | inc (hl) | inr m | 1 | 11 |
| $25 | dec h | dcr h | 1 | 4 | $35 | dec (hl) | dcr m | 1 | 11 |
| $26 | ld h,x | mvi h,x | 2 | 7 | $36 | ld (hl),x | mvi m,x | 2 | 10 |
| $27 | daa | daa | 1 | 4 | $37 | scf | stc | 1 | 4 |
| $28 | jr z,x | – | 2 | 12/7 | $38 | jr c,x | – | 2 | 12/7 |
| $29 | add hl,hl | dad h | 1 | 11 | $39 | add hl,sp | dad sp | 1 | 11 |
| $2A | ld hl,(xx) | lhld xx | 3 | 16 | $3A | ld a,(xx) | lda xx | 3 | 13 |
| $2B | dec hl | dcx h | 1 | 6 | $3B | dec sp | dcx sp | 1 | 6 |
| $2C | inc l | inr l | 1 | 4 | $3C | inc a | inr a | 1 | 4 |
| $2D | dec l | dcr l | 1 | 4 | $3D | dec a | dcr a | 1 | 4 |
| $2E | ld l,x | mvi l,x | 2 | 7 | $3E | ld a,x | mvi a,x | 2 | 7 |
| $2F | cpl | cma | 1 | 4 | $3F | ccf | cmc | 1 | 4 |

Table C.3: $40-$5F

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $40 | ld b,b | mov b,b | 1 | 4 | $50 | ld d,b | mov d,b | 1 | 4 |
| $41 | ld b,c | mov b,c | 1 | 4 | $51 | ld d,c | mov d,c | 1 | 4 |
| $42 | ld b,d | mov b,d | 1 | 4 | $52 | ld d,d | mov d,d | 1 | 4 |
| $43 | ld b,e | mov b,e | 1 | 4 | $53 | ld d,e | mov d,e | 1 | 4 |
| $44 | ld b,h | mov b,h | 1 | 4 | $54 | ld d,h | mov d,h | 1 | 4 |
| $45 | ld b,l | mov b,l | 1 | 4 | $55 | ld d,l | mov d,l | 1 | 4 |
| $46 | ld b,(hl) | mov b,m | 1 | 7 | $56 | ld d,(hl) | mov d,m | 1 | 7 |
| $47 | ld b,a | mov b,a | 1 | 4 | $57 | ld d,a | mov d,a | 1 | 4 |
| $48 | ld c,b | mov c,b | 1 | 4 | $58 | ld e,b | mov e,b | 1 | 4 |
| $49 | ld c,c | mov c,c | 1 | 4 | $59 | ld e,c | mov e,c | 1 | 4 |
| $4A | ld c,d | mov c,d | 1 | 4 | $5A | ld e,d | mov e,d | 1 | 4 |
| $4B | ld c,e | mov c,e | 1 | 4 | $5B | ld e,e | mov e,e | 1 | 4 |
| $4C | ld c,h | mov c,h | 1 | 4 | $5C | ld e,h | mov e,h | 1 | 4 |
| $4D | ld c,l | mov c,l | 1 | 4 | $5D | ld e,l | mov e,l | 1 | 4 |
| $4E | ld c,(hl) | mov c,m | 1 | 7 | $5E | ld e,(hl) | mov e,m | 1 | 7 |
| $4F | ld c,a | mov c,a | 1 | 4 | $5F | ld e,a | mov e,a | 1 | 4 |

Table C.4: $60-$7F

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $60 | ld h,b | mov h,b | 1 | 4 | $70 | ld (hl),b | mov m,b | 1 | 4 |
| $61 | ld h,c | mov h,c | 1 | 4 | $71 | ld (hl),c | mov m,c | 1 | 4 |
| $62 | ld h,d | mov h,d | 1 | 4 | $72 | ld (hl),d | mov m,d | 1 | 4 |
| $63 | ld h,e | mov h,e | 1 | 4 | $73 | ld (hl),e | mov m,e | 1 | 4 |
| $64 | ld h,h | mov h,h | 1 | 4 | $74 | ld (hl),h | mov m,h | 1 | 4 |
| $65 | ld h,l | mov h,l | 1 | 4 | $75 | ld (hl),l | mov m,l | 1 | 4 |
| $66 | ld h,(hl) | mov h,m | 1 | 7 | $76 | halt | halt | 1 | 4+ |
| $67 | ld h,a | mov h,a | 1 | 4 | $77 | ld (hl),a | mov m,a | 1 | 7 |
| $68 | ld l,b | mov l,b | 1 | 4 | $78 | ld a,b | mov a,b | 1 | 4 |
| $69 | ld l,c | mov l,c | 1 | 4 | $79 | ld a,c | mov a,c | 1 | 4 |
| $6A | ld l,d | mov l,d | 1 | 4 | $7A | ld a,d | mov a,d | 1 | 4 |
| $6B | ld l,e | mov l,e | 1 | 4 | $7B | ld a,e | mov a,e | 1 | 4 |
| $6C | ld l,h | mov l,h | 1 | 4 | $7C | ld a,h | mov a,h | 1 | 4 |
| $6D | ld l,l | mov l,l | 1 | 4 | $7D | ld a,l | mov a,l | 1 | 4 |
| $6E | ld l,(hl) | mov l,m | 1 | 7 | $7E | ld a,(hl) | mov a,m | 1 | 7 |
| $6F | ld l,a | mov l,a | 1 | 4 | $7F | ld a,a | mov a,a | 1 | 4 |

Table C.5: $80-$9F

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $80 | add a,b | add b | 1 | 4 | $90 | sub b | sub b | 1 | 4 |
| $81 | add a,c | add c | 1 | 4 | $91 | sub c | sub c | 1 | 4 |
| $82 | add a,d | add d | 1 | 4 | $92 | sub d | sub d | 1 | 4 |
| $83 | add a,e | add e | 1 | 4 | $93 | sub e | sub e | 1 | 4 |
| $84 | add a,h | add h | 1 | 4 | $94 | sub h | sub h | 1 | 4 |
| $85 | add a,l | add l | 1 | 4 | $95 | sub l | sub l | 1 | 4 |
| $86 | add a,(hl) | add m | 1 | 7 | $96 | sub (hl) | sub m | 1 | 7 |
| $87 | add a,a | add a | 1 | 4 | $97 | sub a | sub a | 1 | 4 |
| $88 | adc a,b | adc b | 1 | 4 | $98 | sbc a,b | sbb b | 1 | 4 |
| $89 | adc a,c | adc c | 1 | 4 | $99 | sbc a,c | sbb c | 1 | 4 |
| $8A | adc a,d | adc d | 1 | 4 | $9A | sbc a,d | sbb d | 1 | 4 |
| $8B | adc a,e | adc e | 1 | 4 | $9B | sbc a,e | sbb e | 1 | 4 |
| $8C | adc a,h | adc h | 1 | 4 | $9C | sbc a,h | sbb h | 1 | 4 |
| $8D | adc a,l | adc l | 1 | 4 | $9D | sbc a,l | sbb l | 1 | 4 |
| $8E | adc a,(hl) | adc m | 1 | 7 | $9E | sbc a,(hl) | sbb m | 1 | 7 |
| $8F | adc a,a | adc a | 1 | 4 | $9F | sbc a,a | sbb a | 1 | 4 |

Table C.6: $A0-$BF

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $A0 | and b | ana b | 1 | 4 | $B0 | or b | ora b | 1 | 4 |
| $A1 | and c | ana c | 1 | 4 | $B1 | or c | ora c | 1 | 4 |
| $A2 | and d | ana d | 1 | 4 | $B2 | or d | ora d | 1 | 4 |
| $A3 | and e | ana e | 1 | 4 | $B3 | or e | ora e | 1 | 4 |
| $A4 | and h | ana h | 1 | 4 | $B4 | or h | ora h | 1 | 4 |
| $A5 | and l | ana l | 1 | 4 | $B5 | or l | ora l | 1 | 4 |
| $A6 | and (hl) | ana m | 1 | 7 | $B6 | or (hl) | ora m | 1 | 7 |
| $A7 | and a | ana a | 1 | 4 | $B7 | or a | ora a | 1 | 4 |
| $A8 | xor b | xra b | 1 | 4 | $B8 | cp b | cmp b | 1 | 4 |
| $A9 | xor c | xra c | 1 | 4 | $B9 | cp c | cmp c | 1 | 4 |
| $AA | xor d | xra d | 1 | 4 | $BA | cp d | cmp d | 1 | 4 |
| $AB | xor e | xra e | 1 | 4 | $BB | cp e | cmp e | 1 | 4 |
| $AC | xor h | xra h | 1 | 4 | $BC | cp h | cmp h | 1 | 4 |
| $AD | xor l | xra l | 1 | 4 | $BD | cp l | cmp l | 1 | 4 |
| $AE | xor (hl) | xra m | 1 | 7 | $BE | cp (hl) | cmp m | 1 | 7 |
| $AF | xor a | xra a | 1 | 4 | $BF | cp a | cmp a | 1 | 4 |

Table C.7: $C0-$DF

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $C0 | ret nz | rnz | 1 | 11/5 | $D0 | ret nc | rnc | 1 | 11/5 |
| $C1 | pop bc | pop b | 1 | 10 | $D1 | pop de | pop d | 1 | 10 |
| $C2 | jp nz,xx | jnz xx | 3 | 10 | $D2 | jp nc,xx | jnc xx | 3 | 10 |
| $C3 | jp xx | jmp xx | 3 | 10 | $D3 | out (x),a | out x | 2 | 11 |
| $C4 | call nz,xx | cnz xx | 3 | 17/10 | $D4 | call nc,xx | cnc xx | 3 | 17/10 |
| $C5 | push bc | push b | 1 | 11 | $D5 | push de | push d | 1 | 11 |
| $C6 | add a,x | adi x | 2 | 7 | $D6 | sub x | sui x | 2 | 7 |
| $C7 | rst 00h | rst 0 | 1 | 11 | $D7 | rst 10h | rst 2 | 1 | 11 |
| $C8 | ret z | rz | 1 | 11/5 | $D8 | ret c | rc | 1 | 11/5 |
| $C9 | ret | ret | 1 | 10 | $D9 | exx | – | 1 | 4 |
| $CA | jp z,xx | jz xx | 3 | 10 | $DA | jp c,xx | jc xx | 3 | 10 |
| $CB | xxBITxx | – | +1 | – | $DB | in a,(x) | in x | 2 | 11 |
| $CC | call z,xx | cz xx | 3 | 17/10 | $DC | call c,xx | cc xx | 3 | 17/11 |
| $CD | call xx | call xx | 3 | 17 | $DD | xxIXxx | – | +1 | – |
| $CE | adc a,x | aci x | 2 | 7 | $DE | sbc a,x | sbi x | 2 | 7 |
| $CF | rst 08h | rst 1 | 1 | 11 | $DF | rst 18h | rst 3 | 1 | 11 |

Table C.8: $E0-$FF

| Op | Z80 | 8080 | Sz | T | Op | Z80 | 8080 | Sz | T |
|---|---|---|---|---|---|---|---|---|---|
| $E0 | ret po | rpo | 1 | 11/5 | $F0 | ret p | rp | 1 | 11/5 |
| $E1 | pop hl | pop h | 1 | 10 | $F1 | pop af | pop psw | 1 | 10 |
| $E2 | jp po,xx | jpo xx | 3 | 10 | $F2 | jp p,xx | jp xx | 3 | 10 |
| $E3 | ex (sp),hl | xthl | 1 | 19 | $F3 | di | di | 1 | 4 |
| $E4 | call po,xx | cpo xx | 3 | 17/10 | $F4 | call p,xx | cp xx | 3 | 17/10 |
| $E5 | push hl | push h | 1 | 11 | $F5 | push af | push psw | 1 | 11 |
| $E6 | and x | ani x | 2 | 7 | $F6 | or x | ori x | 2 | 7 |
| $E7 | rst 20h | rst 4 | 1 | 11 | $F7 | rst 30h | rst 6 | 1 | 11 |
| $E8 | ret pe | rpe | 1 | 11/5 | $F8 | ret m | rm | 1 | 11/5 |
| $E9 | jp (hl) | pchl | 1 | 4 | $F9 | ld sp,hl | sphl | 1 | 6 |
| $EA | jp pe,xx | jpe xx | 3 | 10 | $FA | jp m,xx | jm xx | 3 | 10 |
| $EB | ex de,hl | xchg | 1 | 4 | $FB | ei | ei | 1 | 4 |
| $EC | call pe,xx | cpe | 3 | 17/10 | $FC | call m,xx | cm xx | 3 | 17/10 |
| $ED | xx80xx | – | +1 | – | $FD | xxIYxx | – | +1 | – |
| $EE | xor x | xri x | 2 | 7 | $FE | cp x | cpi x | 2 | 7 |
| $EF | rst 28h | rst 5 | 1 | 11 | $FF | rst 38h | rst 7 | 1 | 11 |

## C.2 $CBxx Bit Operations

Table C.9: $CB00-$CB1F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CB00 | rlc b | 2 | 8 | $CB10 | rl b | 2 | 8 |
| $CB01 | rlc c | 2 | 8 | $CB11 | rl c | 2 | 8 |
| $CB02 | rlc d | 2 | 8 | $CB12 | rl d | 2 | 8 |
| $CB03 | rlc e | 2 | 8 | $CB13 | rl e | 2 | 8 |
| $CB04 | rlc h | 2 | 8 | $CB14 | rl h | 2 | 8 |
| $CB05 | rlc l | 2 | 8 | $CB15 | rl l | 2 | 8 |
| $CB06 | rlc (hl) | 2 | 15 | $CB16 | rl (hl) | 2 | 15 |
| $CB07 | rlc a | 2 | 8 | $CB17 | rl a | 2 | 8 |
| $CB08 | rrc b | 2 | 8 | $CB18 | rr b | 2 | 8 |
| $CB09 | rrc c | 2 | 8 | $CB19 | rr c | 2 | 8 |
| $CB0A | rrc d | 2 | 8 | $CB1A | rr d | 2 | 8 |
| $CB0B | rrc e | 2 | 8 | $CB1B | rr e | 2 | 8 |
| $CB0C | rrc h | 2 | 8 | $CB1C | rr h | 2 | 8 |
| $CB0D | rrc l | 2 | 8 | $CB1D | rr l | 2 | 8 |
| $CB0E | rrc (hl) | 2 | 15 | $CB1E | rr (hl) | 2 | 15 |
| $CB0F | rrc a | 2 | 8 | $CB1F | rr a | 2 | 8 |

Table C.10: $CB20-$CB3F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CB20 | sla b | 2 | 8 | $CB30 | sll b | 2 | 8 |
| $CB21 | sla c | 2 | 8 | $CB31 | sll c | 2 | 8 |
| $CB22 | sla d | 2 | 8 | $CB32 | sll d | 2 | 8 |
| $CB23 | sla e | 2 | 8 | $CB33 | sll e | 2 | 8 |
| $CB24 | sla h | 2 | 8 | $CB34 | sll h | 2 | 8 |
| $CB25 | sla l | 2 | 8 | $CB35 | sll l | 2 | 8 |
| $CB26 | sla (hl) | 2 | 15 | $CB36 | sll (hl) | 2 | 15 |
| $CB27 | sla a | 2 | 8 | $CB37 | sll a | 2 | 8 |
| $CB28 | sra b | 2 | 8 | $CB38 | srl b | 2 | 8 |
| $CB29 | sra c | 2 | 8 | $CB39 | srl c | 2 | 8 |
| $CB2A | sra d | 2 | 8 | $CB3A | srl d | 2 | 8 |
| $CB2B | sra e | 2 | 8 | $CB3B | srl e | 2 | 8 |
| $CB2C | sra h | 2 | 8 | $CB3C | srl h | 2 | 8 |
| $CB2D | sra l | 2 | 8 | $CB3D | srl l | 2 | 8 |
| $CB2E | sra (hl) | 2 | 15 | $CB3E | srl (hl) | 2 | 15 |
| $CB2F | sra a | 2 | 8 | $CB3F | srl a | 2 | 8 |

Table C.11: $CB40-$CB5F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CB40 | bit 0,b | 2 | 8 | $CB50 | bit 2,b | 2 | 8 |
| $CB41 | bit 0,c | 2 | 8 | $CB51 | bit 2,c | 2 | 8 |
| $CB42 | bit 0,d | 2 | 8 | $CB52 | bit 2,d | 2 | 8 |
| $CB43 | bit 0,e | 2 | 8 | $CB53 | bit 2,e | 2 | 8 |
| $CB44 | bit 0,h | 2 | 8 | $CB54 | bit 2,h | 2 | 8 |
| $CB45 | bit 0,l | 2 | 8 | $CB55 | bit 2,l | 2 | 8 |
| $CB46 | bit 0,(hl) | 2 | 12 | $CB56 | bit 2,(hl) | 2 | 12 |
| $CB47 | bit 0,a | 2 | 8 | $CB57 | bit 2,a | 2 | 8 |
| $CB48 | bit 1,b | 2 | 8 | $CB58 | bit 3,b | 2 | 8 |
| $CB49 | bit 1,c | 2 | 8 | $CB59 | bit 3,c | 2 | 8 |
| $CB4A | bit 1,d | 2 | 8 | $CB5A | bit 3,d | 2 | 8 |
| $CB4B | bit 1,e | 2 | 8 | $CB5B | bit 3,e | 2 | 8 |
| $CB4C | bit 1,h | 2 | 8 | $CB5C | bit 3,h | 2 | 8 |
| $CB4D | bit 1,l | 2 | 8 | $CB5D | bit 3,l | 2 | 8 |
| $CB4E | bit 1,(hl) | 2 | 12 | $CB5E | bit 3,(hl) | 2 | 12 |
| $CB4F | bit 1,a | 2 | 8 | $CB5F | bit 3,a | 2 | 8 |

Table C.12: $CB60-$CB7F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CB60 | bit 4,b | 2 | 8 | $CB70 | bit 6,b | 2 | 8 |
| $CB61 | bit 4,c | 2 | 8 | $CB71 | bit 6,c | 2 | 8 |
| $CB62 | bit 4,d | 2 | 8 | $CB72 | bit 6,d | 2 | 8 |
| $CB63 | bit 4,e | 2 | 8 | $CB73 | bit 6,e | 2 | 8 |
| $CB64 | bit 4,h | 2 | 8 | $CB74 | bit 6,h | 2 | 8 |
| $CB65 | bit 4,l | 2 | 8 | $CB75 | bit 6,l | 2 | 8 |
| $CB66 | bit 4,(hl) | 2 | 12 | $CB76 | bit 6,(hl) | 2 | 12 |
| $CB67 | bit 4,a | 2 | 8 | $CB77 | bit 6,a | 2 | 8 |
| $CB68 | bit 5,b | 2 | 8 | $CB78 | bit 7,b | 2 | 8 |
| $CB69 | bit 5,c | 2 | 8 | $CB79 | bit 7,c | 2 | 8 |
| $CB6A | bit 5,d | 2 | 8 | $CB7A | bit 7,d | 2 | 8 |
| $CB6B | bit 5,e | 2 | 8 | $CB7B | bit 7,e | 2 | 8 |
| $CB6C | bit 5,h | 2 | 8 | $CB7C | bit 7,h | 2 | 8 |
| $CB6D | bit 5,l | 2 | 8 | $CB7D | bit 7,l | 2 | 8 |
| $CB6E | bit 5,(hl) | 2 | 12 | $CB7E | bit 7,(hl) | 2 | 12 |
| $CB6F | bit 5,a | 2 | 8 | $CB7F | bit 7,a | 2 | 8 |

Table C.13: $CB80-$CB9F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CB80 | res 0,b | 2 | 8 | $CB90 | res 2,b | 2 | 8 |
| $CB81 | res 0,c | 2 | 8 | $CB91 | res 2,c | 2 | 8 |
| $CB82 | res 0,d | 2 | 8 | $CB92 | res 2,d | 2 | 8 |
| $CB83 | res 0,e | 2 | 8 | $CB93 | res 2,e | 2 | 8 |
| $CB84 | res 0,h | 2 | 8 | $CB94 | res 2,h | 2 | 8 |
| $CB85 | res 0,l | 2 | 8 | $CB95 | res 2,l | 2 | 8 |
| $CB86 | res 0,(hl) | 2 | 15 | $CB96 | res 2,(hl) | 2 | 15 |
| $CB87 | res 0,a | 2 | 8 | $CB97 | res 2,a | 2 | 8 |
| $CB88 | res 1,b | 2 | 8 | $CB98 | res 3,b | 2 | 8 |
| $CB89 | res 1,c | 2 | 8 | $CB99 | res 3,c | 2 | 8 |
| $CB8A | res 1,d | 2 | 8 | $CB9A | res 3,d | 2 | 8 |
| $CB8B | res 1,e | 2 | 8 | $CB9B | res 3,e | 2 | 8 |
| $CB8C | res 1,h | 2 | 8 | $CB9C | res 3,h | 2 | 8 |
| $CB8D | res 1,l | 2 | 8 | $CB9D | res 3,l | 2 | 8 |
| $CB8E | res 1,(hl) | 2 | 15 | $CB9E | res 3,(hl) | 2 | 15 |
| $CB8F | res 1,a | 2 | 8 | $CB9F | res 3,a | 2 | 8 |

Table C.14: $CBA0-$CBBF

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CBA0 | res 4,b | 2 | 8 | $CBB0 | res 6,b | 2 | 8 |
| $CBA1 | res 4,c | 2 | 8 | $CBB1 | res 6,c | 2 | 8 |
| $CBA2 | res 4,d | 2 | 8 | $CBB2 | res 6,d | 2 | 8 |
| $CBA3 | res 4,e | 2 | 8 | $CBB3 | res 6,e | 2 | 8 |
| $CBA4 | res 4,h | 2 | 8 | $CBB4 | res 6,h | 2 | 8 |
| $CBA5 | res 4,l | 2 | 8 | $CBB5 | res 6,l | 2 | 8 |
| $CBA6 | res 4,(hl) | 2 | 15 | $CBB6 | res 6,(hl) | 2 | 15 |
| $CBA7 | res 4,a | 2 | 8 | $CBB7 | res 6,a | 2 | 8 |
| $CBA8 | res 5,b | 2 | 8 | $CBB8 | res 7,b | 2 | 8 |
| $CBA9 | res 5,c | 2 | 8 | $CBB9 | res 7,c | 2 | 8 |
| $CBAA | res 5,d | 2 | 8 | $CBBA | res 7,d | 2 | 8 |
| $CBAB | res 5,e | 2 | 8 | $CBBB | res 7,e | 2 | 8 |
| $CBAC | res 5,h | 2 | 8 | $CBBC | res 7,h | 2 | 8 |
| $CBAD | res 5,l | 2 | 8 | $CBBD | res 7,l | 2 | 8 |
| $CBAE | res 5,(hl) | 2 | 15 | $CBBE | res 7,(hl) | 2 | 15 |
| $CBAF | res 5,a | 2 | 8 | $CBBF | res 7,a | 2 | 8 |

Table C.15: $CBC0-$CBDF

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CBC0 | set 0,b | 2 | 8 | $CBD0 | set 2,b | 2 | 8 |
| $CBC1 | set 0,c | 2 | 8 | $CBD1 | set 2,c | 2 | 8 |
| $CBC2 | set 0,d | 2 | 8 | $CBD2 | set 2,d | 2 | 8 |
| $CBC3 | set 0,e | 2 | 8 | $CBD3 | set 2,e | 2 | 8 |
| $CBC4 | set 0,h | 2 | 8 | $CBD4 | set 2,h | 2 | 8 |
| $CBC5 | set 0,l | 2 | 8 | $CBD5 | set 2,l | 2 | 8 |
| $CBC6 | set 0,(hl) | 2 | 15 | $CBD6 | set 2,(hl) | 2 | 15 |
| $CBC7 | set 0,a | 2 | 8 | $CBD7 | set 2,a | 2 | 8 |
| $CBC8 | set 1,b | 2 | 8 | $CBD8 | set 3,b | 2 | 8 |
| $CBC9 | set 1,c | 2 | 8 | $CBD9 | set 3,c | 2 | 8 |
| $CBCA | set 1,d | 2 | 8 | $CBDA | set 3,d | 2 | 8 |
| $CBCB | set 1,e | 2 | 8 | $CBDB | set 3,e | 2 | 8 |
| $CBCC | set 1,h | 2 | 8 | $CBDC | set 3,h | 2 | 8 |
| $CBCD | set 1,l | 2 | 8 | $CBDD | set 3,l | 2 | 8 |
| $CBCE | set 1,(hl) | 2 | 15 | $CBDE | set 3,(hl) | 2 | 15 |
| $CBCF | set 1,a | 2 | 8 | $CBDF | set 3,a | 2 | 8 |

Table C.16: $CBE0-$CBFF

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $CBE0 | set 4,b | 2 | 8 | $CBF0 | set 6,b | 2 | 8 |
| $CBE1 | set 4,c | 2 | 8 | $CBF1 | set 6,c | 2 | 8 |
| $CBE2 | set 4,d | 2 | 8 | $CBF2 | set 6,d | 2 | 8 |
| $CBE3 | set 4,e | 2 | 8 | $CBF3 | set 6,e | 2 | 8 |
| $CBE4 | set 4,h | 2 | 8 | $CBF4 | set 6,h | 2 | 8 |
| $CBE5 | set 4,l | 2 | 8 | $CBF5 | set 6,l | 2 | 8 |
| $CBE6 | set 4,(hl) | 2 | 15 | $CBF6 | set 6,(hl) | 2 | 15 |
| $CBE7 | set 4,a | 2 | 8 | $CBF7 | set 6,a | 2 | 8 |
| $CBE8 | set 5,b | 2 | 8 | $CBF8 | set 7,b | 2 | 8 |
| $CBE9 | set 5,c | 2 | 8 | $CBF9 | set 7,c | 2 | 8 |
| $CBEA | set 5,d | 2 | 8 | $CBFA | set 7,d | 2 | 8 |
| $CBEB | set 5,e | 2 | 8 | $CBFB | set 7,e | 2 | 8 |
| $CBEC | set 5,h | 2 | 8 | $CBFC | set 7,h | 2 | 8 |
| $CBED | set 5,l | 2 | 8 | $CBFD | set 7,l | 2 | 8 |
| $CBEE | set 5,(hl) | 2 | 15 | $CBFE | set 7,(hl) | 2 | 15 |
| $CBEF | set 5,a | 2 | 8 | $CBFF | set 7,a | 2 | 8 |

## C.3 $DDxx IX

Table C.17: $DD00-$DD5F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $DD09 | add ix,bc | 2 | 15 | $DD35 | dec (ix+x) | 3 | 23 |
| $DD19 | add ix,de | 2 | 15 | $DD36 | ld (ix+x),x | 5 | 19 |
| $DD21 | ld ix,xx | 4 | 14 | $DD39 | add ix,sp | 2 | 15 |
| $DD22 | ld (xx),ix | 4 | 20 | $DD44 | ld b,ixh | 2 | 8 |
| $DD23 | inc ix | 2 | 10 | $DD45 | ld b,ixl | 2 | 8 |
| $DD24 | inc ixh | 2 | 8 | $DD46 | ld b,(ix+x) | 2 | 19 |
| $DD25 | dec ixh | 2 | 8 | $DD4C | ld c,ixh | 2 | 8 |
| $DD26 | ld ixh,x | 3 | 11 | $DD4D | ld c,ixl | 2 | 8 |
| $DD29 | add ix,ix | 2 | 15 | $DD4E | ld c,(ix+x) | 3 | 19 |
| $DD2A | ld ix,(xx) | 4 | 20 | $DD54 | ld d,ixh | 2 | 8 |
| $DD2B | dec ix | 2 | 10 | $DD55 | ld d,ixl | 2 | 8 |
| $DD2C | inc ixl | 2 | 8 | $DD56 | ld d,(ix+x) | 3 | 19 |
| $DD2D | dec ixl | 2 | 8 | $DD5C | ld e,ixh | 2 | 8 |
| $DD2E | ld ixl,x | 4 | 11 | $DD5D | ld e,ixl | 2 | 8 |
| $DD34 | inc (ix+x) | 3 | 23 | $DD5E | ld e,(ix+x) | 3 | 19 |

Table C.18: $DD60-$DD8F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $DD60 | ld ixh,b | 2 | 8 | $DD70 | ld (ix+x),b | 3 | 19 |
| $DD61 | ld ixh,c | 2 | 8 | $DD71 | ld (ix+x),c | 3 | 19 |
| $DD62 | ld ixh,d | 2 | 8 | $DD72 | ld (ix+x),d | 3 | 19 |
| $DD63 | ld ixh,e | 2 | 8 | $DD73 | ld (ix+x),e | 3 | 19 |
| $DD64 | ld ixh,ixh | 2 | 8 | $DD74 | ld (ix+x),h | 3 | 19 |
| $DD65 | ld h,(ix+x) | 3 | 19 | $DD75 | ld (ix+x),l | 3 | 19 |
| $DD65 | ld ixh,ixl | 2 | 8 | $DD77 | ld (ix+x),a | 3 | 19 |
| $DD67 | ld ixh,a | 2 | 8 | $DD7C | ld a,ixh | 2 | 8 |
| $DD68 | ld ixl,b | 2 | 8 | $DD7D | ld a,ixl | 2 | 8 |
| $DD69 | ld ixl,c | 2 | 8 | $DD7E | ld a,(ix+x) | 3 | 19 |
| $DD6A | ld ixl,d | 2 | 8 | $DD84 | add a,ixh | 2 | 8 |
| $DD6B | ld ixl,e | 2 | 8 | $DD85 | add a,ixl | 2 | 8 |
| $DD6C | ld ixl,ixh | 2 | 2 | $DD86 | add a,(ix+x) | 3 | 19 |
| $DD6D | ld ixl,ixl | 2 | 2 | $DD8C | adc a,ixh | 2 | 8 |
| $DD6E | ld l,(ix+x) | 3 | 19 | $DD8D | adc a,ixl | 2 | 8 |
| $DD6F | ld ixl,a | 2 | 8 | $DD8E | adc a,(ix+x) | 3 | 19 |

Table C.19: $DD90-$DDFF

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $DD94 | sub ixh | 2 | 8 | $DDB4 | or ixh | 2 | 8 |
| $DD95 | sub ixl | 2 | 8 | $DDB5 | or ixl | 2 | 8 |
| $DD96 | sub (ix+x) | 3 | 19 | $DDB6 | or (ix+x) | 3 | 19 |
| $DD9C | sbc a,ixh | 2 | 8 | $DDBC | cp ixh | 2 | 8 |
| $DD9D | sbc a,ixl | 2 | 8 | $DDBD | cp ixl | 2 | 8 |
| $DD9E | sbc a,(ix+x) | 3 | 1 | $DDBE | cp (ix+x) | 2 | 19 |
| $DDA4 | and ixh | 2 | 8 | $DDCB | xBIT+IXx | +1 | – |
| $DDA5 | and ixl | 2 | 8 | $DDE1 | pop ix | 2 | 14 |
| $DDA6 | and (ix+x) | 3 | 19 | $DDE3 | ex (sp),ix | 2 | 23 |
| $DDAC | xor ixh | 2 | 8 | $DDE5 | push ix | 2 | 15 |
| $DDAD | xor ixl | 2 | 8 | $DDE9 | jp (ix) | 3 | 8 |
| $DDAE | xor (ix+x) | 3 | 19 | $DDF9 | ld sp,ix | 2 | 10 |

## C.4 $EDxx Block/Port

Table C.20: $ED00-$ED4F

| Opcode | Mnemonic | Bytes | Timing | Opcode | Mnemonic | Bytes | Timing |
|---|---|---|---|---|---|---|---|
| $ED23 | swapinb * | 2 | 8 | $ED40 | in b,(c) | 2 | 12 |
| $ED24 | mirror a * | 2 | 8 | $ED41 | out (c),b | 2 | 12 |
| $ED27 | test x * | 3 | 11 | $ED42 | sbc hl,bc | 2 | 15 |
| $ED28 | bsla de,b * | 2 | 8 | $ED43 | ld (xx),bc | 4 | 20 |
| $ED29 | bsra de,b * | 2 | 8 | $ED44 | neg | 2 | 8 |
| $ED2A | bsrl de,b * | 2 | 8 | $ED45 | retn | 2 | 14 |
| $ED2B | bsrf de,b * | 2 | 8 | $ED46 | im 0 | 2 | 8 |
| $ED2C | brlc de,b * | 2 | 8 | $ED47 | ld i,a | 2 | 9 |
| $ED30 | mul d,e * | 2 | 8 | $ED48 | in c,(c) | 2 | 12 |
| $ED31 | add hl,a * | 2 | 8 | $ED49 | out (c),c | 2 | 12 |
| $ED32 | add de,a * | 2 | 8 | $ED4A | adc hl,bc | 2 | 15 |
| $ED33 | add bc,a * | 2 | 8 | $ED4B | ld bc,(xx) | 4 | 20 |
| $ED34 | add hl,xx * | 4 | 16 | $ED4D | reti | 2 | 14 |
| $ED35 | add de,xx * | 4 | 16 | $ED4F | ld r,a | 2 | 9 |
| $ED36 | add bc,xx * | 4 | 16 | | | | |

* ZX Spectrum Next extension

Table C.21: $ED50-$ED8F

| Opcode | Mnemonic | Bytes | Timing | Opcode | Mnemonic | Bytes | Timing |
|---|---|---|---|---|---|---|---|
| $ED50 | in d,(c) | 2 | 12 | $ED67 | rrd | 2 | 18 |
| $ED51 | out (c),d | 2 | 12 | $ED68 | in l,(c) | 2 | 12 |
| $ED52 | sbc hl,de | 2 | 15 | $ED69 | out (c),l | 2 | 12 |
| $ED53 | ld (xx),de | 4 | 20 | $ED6A | adc hl,hl | 2 | 15 |
| $ED56 | im 1 | 2 | 8 | $ED6B | ld hl,(xx) | 4 | 20 |
| $ED57 | ld a,i | 2 | 9 | $ED6F | rld | 2 | 18 |
| $ED58 | in e,(c) | 2 | 12 | $ED70 | in f,(c) | 2 | 12 |
| $ED59 | out (c),e | 2 | 12 | $ED71 | out (c),f | 2 | 12 |
| $ED5A | adc hl,de | 2 | 15 | $ED72 | sbc hl,sp | 2 | 15 |
| $ED5B | ld de,(xx) | 4 | 20 | $ED73 | ld (xx),sp | 4 | 20 |
| $ED5E | im 2 | 2 | 8 | $ED78 | in a,(c) | 2 | 12 |
| $ED5F | ld a,r | 2 | 9 | $ED79 | out (c),a | 2 | 12 |
| $ED60 | in h,(c) | 2 | 12 | $ED7A | adc hl,sp | 2 | 15 |
| $ED61 | out (c),h | 2 | 12 | $ED7B | ld sp,(xx) | 4 | 20 |
| $ED62 | sbc hl,hl | 2 | 15 | $ED8A | push xx | 4 | * |
| $ED63 | ld (xx),hl | 4 | | | | | |

Table C.22: $ED90-$EDFF

| Opcode | Mnemonic | Bytes | Timing | Opcode | Mnemonic | Bytes | Timing |
|---|---|---|---|---|---|---|---|
| $ED90 | outinb * | 2 | 16 | $EDAA | ind | 2 | 16 |
| $ED91 | nextreg r,v * | 4 | 20 | $EDAB | outd | 2 | 16 |
| $ED92 | nextreg r,a * | 3 | 17 | $EDAC | lddx * | 2 | 16 |
| $ED93 | pixeldn * | 2 | 8 | $EDB0 | ldir | 2 | 21/16 |
| $ED94 | pixelad * | 2 | 8 | $EDB1 | cpir | 2 | 21/16 |
| $ED95 | setae * | 2 | 8 | $EDB2 | inir | 2 | 21/16 |
| $ED98 | jp (c) * | 2 | 13 | $EDB3 | otir | 2 | 21/16 |
| $EDA0 | ldi | 2 | 16 | $EDB4 | ldirx * | 2 | 21/16 |
| $EDA1 | cpi | 2 | 16 | $EDB7 | ldpirx * | 2 | 21/16 |
| $EDA2 | ini | 2 | 16 | $EDB8 | lddr | 2 | 21/16 |
| $EDA3 | outi | 2 | 16 | $EDB9 | cpdr | 2 | 21/16 |
| $EDA4 | ldix * | 2 | 16 | $EDBA | indr | 2 | 21/16 |
| $EDA5 | ldws * | 2 | 14 | $EDBB | otdr | 2 | 12/16 |
| $EDA8 | ldd | 2 | 16 | $EDBC | lddrx * | 2 | 21/16 |
| $EDA9 | cpd | 2 | 16 | | | | |

* ZX Spectrum Next extension

## C.5 $FDxx IY

Table C.23: $FD00-$FD5F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $FD09 | add iy,bc | 2 | 15 | $FD35 | dec (iy+x) | 3 | 23 |
| $FD19 | add iy,de | 2 | 15 | $FD36 | ld (iy+x),x | 5 | 19 |
| $FD21 | ld iy,xx | 4 | 14 | $FD39 | add iy,sp | 2 | 15 |
| $FD22 | ld (xx),iy | 4 | 20 | $FD44 | ld b,iyh | 2 | 8 |
| $FD23 | inc iy | 2 | 10 | $FD45 | ld b,iyl | 2 | 8 |
| $FD24 | inc iyh | 2 | 8 | $FD46 | ld b,(iy+x) | 2 | 19 |
| $FD25 | dec iyh | 2 | 8 | $FD4C | ld c,iyh | 2 | 8 |
| $FD26 | ld iyh,x | 3 | 11 | $FD4D | ld c,iyl | 2 | 8 |
| $FD29 | add iy,iy | 2 | 15 | $FD4E | ld c,(iy+x) | 3 | 19 |
| $FD2A | ld iy,(xx) | 4 | 20 | $FD54 | ld d,iyh | 2 | 8 |
| $FD2B | dec iy | 2 | 10 | $FD55 | ld d,iyl | 2 | 8 |
| $FD2C | inc iyl | 2 | 8 | $FD56 | ld d,(iy+x) | 3 | 19 |
| $FD2D | dec iyl | 2 | 8 | $FD5C | ld e,iyh | 2 | 8 |
| $FD2E | ld iyl,x | 4 | 11 | $FD5D | ld e,iyl | 2 | 8 |
| $FD34 | inc (iy+x) | 3 | 23 | $FD5E | ld e,(iy+x) | 3 | 19 |

Table C.24: $FD60-$FD8F

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $FD60 | ld iyh,b | 2 | 8 | $FD70 | ld (iy+x),b | 3 | 19 |
| $FD61 | ld iyh,c | 2 | 8 | $FD71 | ld (iy+x),c | 3 | 19 |
| $FD62 | ld iyh,d | 2 | 8 | $FD72 | ld (iy+x),d | 3 | 19 |
| $FD63 | ld iyh,e | 2 | 8 | $FD73 | ld (iy+x),e | 3 | 19 |
| $FD64 | ld iyh,iyh | 2 | 8 | $FD74 | ld (iy+x),h | 3 | 19 |
| $FD65 | ld h,(iy+x) | 3 | 19 | $FD75 | ld (iy+x),l | 3 | 19 |
| $FD65 | ld iyh,iyl | 2 | 8 | $FD77 | ld (iy+x),a | 3 | 19 |
| $FD67 | ld iyh,a | 2 | 8 | $FD7C | ld a,iyh | 2 | 8 |
| $FD68 | ld iyl,b | 2 | 8 | $FD7D | ld a,iyl | 2 | 8 |
| $FD69 | ld iyl,c | 2 | 8 | $FD7E | ld a,(iy+x) | 3 | 19 |
| $FD6A | ld iyl,d | 2 | 8 | $FD84 | add a,iyh | 2 | 8 |
| $FD6B | ld iyl,e | 2 | 8 | $FD85 | add a,iyl | 2 | 8 |
| $FD6C | ld iyl,iyh | 2 | 2 | $FD86 | add a,(iy+x) | 3 | 19 |
| $FD6D | ld iyl,iyl | 2 | 2 | $FD8C | adc a,iyh | 2 | 8 |
| $FD6E | ld l,(iy+x) | 3 | 19 | $FD8D | adc a,iyl | 2 | 8 |
| $FD6F | ld iyl,a | 2 | 8 | $FD8E | adc a,(iy+x) | 3 | 19 |

Table C.25: $FD90-$FDFF

| Opcode | Mnemonic | Sz | T | Opcode | Mnemonic | Sz | T |
|---|---|---|---|---|---|---|---|
| $FD94 | sub iyh | 2 | 8 | $FDB4 | or iyh | 2 | 8 |
| $FD95 | sub iyl | 2 | 8 | $FDB5 | or iyl | 2 | 8 |
| $FD96 | sub (iy+x) | 3 | 19 | $FDB6 | or (iy+x) | 3 | 19 |
| $FD9C | sbc a,iyh | 2 | 8 | $FDBC | cp iyh | 2 | 8 |
| $FD9D | sbc a,iyl | 2 | 8 | $FDBD | cp iyl | 2 | 8 |
| $FD9E | sbc a,(iy+x) | 3 | 1 | $FDBE | cp (iy+x) | 2 | 19 |
| $FDA4 | and iyh | 2 | 8 | $FDCB | xBIT+IYx | +1 | – |
| $FDA5 | and iyl | 2 | 8 | $FDE1 | pop iy | 2 | 14 |
| $FDA6 | and (iy+x) | 3 | 19 | $FDE3 | ex (sp),iy | 2 | 23 |
| $FDAC | xor iyh | 2 | 8 | $FDE5 | push iy | 2 | 15 |
| $FDAD | xor iyl | 2 | 8 | $FDE9 | jp (iy) | 3 | 8 |
| $FDAE | xor (iy+x) | 3 | 19 | $FDF9 | ld sp,iy | 2 | 10 |

## C.6 $DDCBxx IY Bit Operations

## C.7 $FDCBxx IX Bit Operations

# Appendix D

# Opcodes to Extended Mnemonics

# Appendix E

# Memory Map

## E.1 Global Memory Map

## E.2 Z80 Visible Memory Map

Appendix F

# File Formats

## F.1 NEX

## F.2 SCR

## F.3 SHC

## F.4 SHR

## F.5 SL2

## F.6 SLR

## F.7 SPR

# List of Figures

# List of Tables

# Index