ZX Spectrum Next Programming Notes

Theodore (Alex) Evans

May 31, 2020

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 This document started when much of the documentation on the ZX Spectrum Next site (https://www.specnext.com/) was out of date and/or difficult to figure out.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: Z80n normal and turbo modes
- Memory: 1024Kb RAM (expandable to 2048Kb on board)
- Video: Multilayer video implementing classic ZX Spectrum, Timex Hi-Resolution, Timex Hi-Colour, LoRes, Layer 2, and Tilemap video modes with Hardware sprites

- Video Output: RGB, VGA, HDMI
- Storage: SD Card slot, with DivMMC-compatible protocol
- Audio: ZX Spectrum 1-bit audio, Turbo Sound Next (3x AY-3-8912 audio chips with stereo output), stereo PCM audio, and ability to use Pi accelerator as a sound source
- 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): Pi Zero with 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 (layer 1 (ULA/Timex/LoRes), layer 2, layer 3 (tilemap) 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 General Features

There are a number of control features for the various video modes that are done in a unified fashion. These features are layering and transparency, palettes, scrolling, and clipping. For the sake of convenience we will occasionally talk about a global coordinate system for graphics on the ZX Next. This coordinate system has (0, 0) at the upper left corner of the usable display area and (319, 255) at the lower right corner. Individual pixels generally correspond to integer locations in this grid, but some modes may either double or halve this grid. This will be discussed in the sections for each of the video layers.

2.1.1 Video Layering and Transparency

Video for the ZX Next is composed of a number of features and layers each of which may have its own set of video modes. Not all of these features are mandatory.

By composing together the border colour and transparency fallback color, layer 1 (ULA, Timex modes, or LoRes), layer 2 ($256 \times 192 \times 256$, $320 \times 256 \times 256$, or $640 \times 256 \times 6$), layer 3 (16 or 2 colour tiles), and sprites; we generate the full video display.

The border/transparency fallback is the bottom with the ordering of the layers controlled by a combination of the video layering register (Next register $15 (21) bits 4-2), the interaction of layers 1 and 3 (Next register $6B (107) bit 0), and whether or not a pixel in layer 2 is set as a priority colour.

Register (R/W) $15 (21) ⇒ Sprite and Layer System Setup

- bit 7 = LoRes mode (0 on reset)
- bit 6 = Sprite priority (1 = sprite 0 on top, 0 = sprite 127 on top) (0 on reset)
- bit 5 = Enable sprite clipping in over border mode (0 on reset)
- bits 4-2 = set layers priorities (000 on reset)
  - 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 = Enable Sprites Over border (0 on reset)
- bit 0 = Enable Sprites (0 on reset)

Transparency for Layer 2, Layer 1, and 1-bit Tilemaps are controlled by Next register $14 (20) and defaults to $E3. Sprites and 4-bit Tilemaps have their own registers ($4B and $4C respectively) for setting their transparency index (not colour). 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.

Register (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)

Register (R/W) $4A (74) ⇒ Fallback Colour Value

- bits 7-0 = 8-bit colour if all layers are transparent ($E3 on reset)

(black on reset = 0)

Register (R/W) $4B (75) ⇒ Sprite Transparency Index

- bits 7-0 = Index value ($E3 if reset)

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

Register (R/W) $4C (76) ⇒ Level 3 Transparency Index

- bits 7-4 = Reserved, must be 0
- bits 3-0 = Index value ($0F on reset)

2.1.2 Palette

Next Colour Palettes Each video mode group has a pair of palettes assigned to it a primary and an alternate palette. Each palette entry is actually a 9-bit value (RRRGGGBBB) and can be set by selecting a palette using nextreg $43 (palette control), the entry using nextreg $40 (palette index), then writing the value into nextreg $44 (palette value, 9-bit) using pairs of consecutive writes for each palette value or nextreg $41 (palette value, 8-bit). Once a palette index has been selected writes automatically increment the palette index number so it is possible to efficiently write the values for a collection of palette entries.

Register (R/W) $40 (64) ⇒ Palette Index Select

- bits 7-0 = Palette Index Number

Selects the palette index to change the associated colour

For ULA only, INKs are mapped to indices 0 through 7, BRIGHT INKs to indices 8 through 15, PAPERs to indices 16 through 23 and BRIGHT PAPERs to indices 24 through 31. In EnhancedULA mode, INKs come from a subset of indices from 0 through 127 and PAPERs from a subset of indices from 128 through 255.

The number of active indices depends on the number of attribute bits assigned to INK and PAPER out of the attribute byte.

In ULAplus mode, the last 64 entries (indices 192 to 255) hold the ULAplus palette. The ULA always takes border colour from PAPER for standard ULA and Enhanced ULA

Register (R/W) $41 (65) ⇒ 8-bit Palette Data

- bits 7-0 = Colour Entry in RRRGGGBB format

The lower blue bit of the 9-bit internal colour will be the logical or of bits 0 and 1 of the 8-bit entry. After each write, the palette index auto-increments if aut-increment has been enabled (NextReg $43 bit 7), Reads do not auto-increment.

Register (R/W) $43 (67) ⇒ Palette Control

- bit 7 = 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 = Sprite first palette
  - 110 = Sprite second palette
  - 011 = Layer 3 first palette
  - 111 = Layer 3 second palette
- bit 3 = Select Sprite 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 EnhancedULA mode if 1. (0 after a reset)

Register (R/W) $44 (68) ⇒ 9-bit Palette Data
Non Level 2

1st write

- bits 7-0 = MSB (RRRGGGBB)

2nd write

- bits 7-1 = Reserved, must be 0
- bit 0 = LSB (B)

Level 2

1st write
- bits 7-0 = MSB (RRRGGGBB)

2nd write
- bit 7 = Priority
- bits 6-1 = Reserved, must be 0
- bit 0 = LSB (B)

9-bit Palette Data is entered in two consecutive writes; the second write autoincrements the palette index if auto-increment is enabled in NextREG $43 bit 7

If writing an L2 palette, the second write's D7 holds the L2 priority bit which if set (1) brings the colour defined at that index on top of all other layers. If you also need the same colour in regular priority (for example: for enviromental masking) you will have to set it up again, this time with no priority.

Reads return the second byte and do not autoincrement.

2.1.3 Scrolling

The ZX Spectrum Next has four sets of scrolling registers to independently contol the display offsets of various video modes (Layer2, ULA, Tilemap, and LoRes). When the video is offset, the portion that is pushed off the screen (to the left and or top) then becomes visible on the opposite side of the screen so that the video offset values are effectively the coordinates of the origin in a toroidal universe.

2.1.4 Clipping

The ZX Spectrum Next has four clipping registers create a window of the layer that is visible. Clipping is managed by a set of four successive writes to the clipping register applicable for the video mode. If a section is masked off by clipping, it is as if the area were the transparency colour and the video lyers behind it become visible.

2.2 Layer 1

The Layer 1 consists of ZX Spectrum ULA video, Timex video modes, and the Spectrum Next's lores video modes 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, ULANext mode, or an emulation of ULA+. In the Spectrum and Timex modes all colours are either Paper (foreground), paper (background), or border colours.

2.2.1 Colour Attributes

The ZX Spectrum Next has three major modes for colour attributes: the ZX Spectrum attribute mapping, which is augmented by using the ZX Spectrum Next's palette; ULA Next, which allows the user to how many foreground and how many background colous are to be selected by the attribute bytes; and an emulation of ULA+.

ULA Colour In ULA colour 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. This is the default state for interpreting ULA palettes.

Table 2.1: ULA Colour

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

ULA Next The ULA next modes use a varying number of bits from the attribute byte to determine the ink colours 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. ULA next is enabled

using bit 0 of nextreg $43 (palette control) and controlled with nextreg $42 (ULA Next attribute byte format)

Table 2.2: 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

ULA+ The ZX Next emulates ULAPlus using the last 64 (192-255) entries of the ULA palette. ULA+ is controlled using two ports: $BF3B (register port) and $FF3B (data port)

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 7-6: 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 5-0: Select the register sub-group
  Mode group
- Bits 5-3: 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
- Bits 2-0: 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)

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

2.2.2 Layer 1 Scrolling

Layer 1 has two sets of scrolling registers. One for the the legacy modes (ZX Spectrum, Alternate Page, Timex Hi-Resoulution, and Timex Hi-colour) and a second set for the two ZX Spextrum Next specific LoRes modes. All modes scroll as if they were 256×192 screens located at global coordinates (32, 32) to (287, 223), The registers for the legacy modes are $26 and $27 and the registers for the LoRes modes are $32 and $33.

Register (R/W) $26 (38) ⇒ ULA Horizontal Scroll Control

- bits 7-0 = ULA X Offset (0-255) (0 on reset)

Register (R/W) $27 (39) ⇒ ULA Vertical Scroll Control

- bits 7-0 = ULA Y Offset (0-191) (0 on reset)

Register (R/W) $32 (50) ⇒ Layer 1,0 (LoRes) Horizontal Scroll Control)

- bits 7-0 = X Offset (0-255) ($00 on reset)

Layer 1,0 (LoRes) scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

Register (R/W) $33 (51) ⇒ Layer 1,0 (LoRes) Vertical Scroll Control)

- bits 7-0 = Y Offset (0-191) ($00 on reset)

Layer 1,0 (LoRes) scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

2.2.3 Layer 1 Clipping

All of the modes in the Layer 1 share a single clipping register, $1A. The clip index may alternately be set using register $1C. This is expecially useful for reading the current clipping coordinates as reads on the clipping register do not change the index. Note that clipping coordinates are based on a full display area for the mode of 256×192 resolution even though not all modes have that resolution.

Register (R/W) $1A (26) ⇒ Layer 0 (ULA/LoRes) Clip Window Definition

- 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

Register (R/W) $1C (28) ⇒ Clip Window Control
Read

- bits 7-6 = Layer 3 Clip Index
- bits 5-4 = Layer 0/1 Clip Index
- bits 3-2 = Sprite clip index
- bits 1-0 = Layer 2 Clip Index

Write

- bits 7-4 = Reserved, must be 0
- bit 3 - reset Layer 3 clip index

- bit 2 - reset Layer 0/1 clip index
- bit 1 - reset sprite clip index.
- bit 0 - reset Layer 2 clip index.

2.2.4 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 located at global coordinates (32, 32) to (287, 223) 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
```

2.2.5 Alternate Page Mode

Timex mode 1

This mode is the same as ZX Spectrum mode except it is at an 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
```

2.2.6 Timex Hi-Colour Mode

Timex mode 2

This mode is a 256×192 video mode located at global coordinates (32, 32) to (287, 223) 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
```

2.2.7 Timex Hi-Resolution Mode

Timex mode 6

This is a monochrome 512×192 video mode located at global coordinates (32, 32) to (287, 223) with each pixel being half width. 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 2.3: 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
```

2.2.8 Lo-Resolution Mode

This is a Spectrum Next specific video mode with a resolution of 128×96 located at global coordinates (32, 32) to (287, 223) with each pixel being double height and double width replacing the old Radistan mode. It can either allow for 16 colours, in which case it uses either the bitmap 1 area or the bitmap 2 area, or 256 colours using both bitmap 1 and bitmap 2. The colour of each pixel can be selected independently with data ordered linearly in a row major fashion. In the case of 16 colour mode, the nybbles describing the colours are X major (MSN LSN). Scrolling is by half pixels and uses different registers ($32 and $33) from the rest of the ULA group modes. LoRes mode is enabled by writing $100xxxxx$ to Next register $15 (the sprites and layers register) with Next register $6A used to decide whether it is 16 or 256 colours.

Register (R/W) $15 (21) ⇒ Sprite and Layer System Setup

- bit 7 = LoRes mode (0 on reset)
- bit 6 = Sprite priority (1 = sprite 0 on top, 0 = sprite 127 on top) (0 on reset)
- bit 5 = Enable sprite clipping in over border mode (0 on reset)
- bits 4-2 = set layers priorities (000 on reset)
  - 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 = Enable Sprites Over border (0 on reset)
- bit 0 = Enable Sprites (0 on reset)

Register (R/W) $32 (50) ⇒ Layer 1,0 (LoRes) Horizontal Scroll Control)

- bits 7-0 = X Offset (0-255) ($00 on reset)

Layer 1,0 (LoRes) scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

Register (R/W) $33 (51) ⇒ Layer 1,0 (LoRes) Vertical Scroll Control)

- bits 7-0 = Y Offset (0-191) ($00 on reset)

Layer 1,0 (LoRes) scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

Register (R/W) $6A (106) ⇒ Layer 1,0 (LoRes) Control

- bits 7-6 = reserved, must be 0
- bit 5 = Enable Radistan (16-colour) (0 on reset)
- bit 4 = Radistan DFILE switch (xor with bit 0 of port $ff) (0 on reset)
- bits 3-0 = Radistsan palette offset (0 on reset)
- bits 1-0 = ULAplus palette offset (0 on reset)

Code: 256 colour

```
;; enable LoRes mode:
nextreg $15,$80
;; 256-colour mode
ld bc,$243B ; next register select port
ld a,$6A
out (c),a
ld bc,$253B ; next register r/w port
in a,(c)
and $EF ; lores radistan control
out (c),a
```

Code: 16 colour

```
;; enable LoRes mode:
nextreg $15,$80
;; 16-colour mode
nextreg $6A,$10
```

2.3 Layer 2

Layer 2 is a linearly mapped, row-wise, upper left to lower right, bit-map graphics area. It supports modes with $256 \times 192 \times 256$ resolution at global

coordinates (32, 32) to (287, 223), 320 × 256 × 256 resolution at global coordinates (0, 0) to (318, 255), and 640 × 256 × 16 resolution at global coordinates (0, 0) to (319, 255) with half width pixels. It can be mapped starting at any 16k memory blocks. The 256 × 192 × 256 mode requires 3 consecutive blocks (48k) while the others use 5 consecutive blocks (80k).

2.3.1 Configuration

Layer 2 is enabled using port $123B or register $69. The mode is selected using register $70. How layer 2 memory is overlaid on main memory is controled by port $123B and register $70. The location in memory is controlled by register $12 with a shadow area pointed to by register $13 for double buffering. Finally port $123B is used to select either the main RAM area or the shadow RAM area for rendering the layer.

Port $123B (4667) Layer 2
Bit 4 = 0

bits 7-6 = Video RAM bank select
00 = first 16k
01 = second 16k
10 = third 16k
11 = first 48k
bit 5 = Reserved, must be 0
bit 4 = 0
bit 3 = Shadow layer 2 select
bit 2 = Enable layer 2 read paging
bit 1 = Layer 2 visible (mirrored in register $69)
bit 0 = Enable layer 2 write paging

Bit 4 = 1

bits 7-5 = Reserved, must be 0
bit 4 = 1
bit 3 = Reserved, must be 0
bit 2-0 = 16k bank relative offset

Register (R/W) $12 (18) ⇒ Layer 2 Active 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)

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

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

Register (R/W) $69 (105) ⇒ Display Control 1

- bit 7 = Layer 2 Enable (Port $123B bit 1 alias)
- bit 6 = ULA Shadow display enable (Port $7FFD bit 3 alias)
- bits 5-0 = Timex alias (Port $FF alias)

Register (R/W) $70 (112) ⇒ Layer 2 Control

- bits 7-6 = Reserved, must be 0
- bits 5-4 = Resolution (00 on soft reset)
  - 00 = 256 × 192 × 256
  - 01 = 320 × 256 × 256
  - 10 = 640 × 256 × 16
  - 11 = Do not use
- bits 3-0 = Palette offset ($0 on soft reset)

Layer 2 256 × 192**, Write only overlaid on ROM**

```
p_layer2: defl $123b
start:
  ld bc,p_layer2
  ld a,$03       ; enable, wo, 1st 16k
  out (c),a
  call wrtpage
  ld bc,p_layer2
  ld a,$43       ; enable, wo, 2nd 16k
  out (c),a
  call wrtpage
  ld bc,p_layer2
  ld a,$83       ; enable, wo, 3rd 16k
  out (c),a
  call wrtpage
  ret
wrtpage:
  ld hl,$0000
  ld bc,$0040    ; 40*256 writes
```

```
loop:
  ld (hl),b
  inc hl
  djnz loop
  dec c
  jr nz,loop
```

Layer 2 256×192 **resolution**

```
r_mmu_7:  defl $57
r_disp1:  defl $69
r_layer2: defl $70
start:
  nextreg r_disp1,$80  ; enable layer 2
  nextreg r_layer2,$00 ; 256x192x256
  ld a,$12             ; page 18=bank 9
loop1:
  nextreg r_mmu_7,a    ; map page into slot 7
  ld bc,$0020          ; 20*256 = 8k
  ld hl,$E000          ; address of slot 7
loop2:
  ld (hl),b
  inc hl
  djnz loop2
  dec c
  jp NZ,loop2
  inc a
  cp $18               ; stop at page 24
  jp NZ,loop1
```

Layer 2 320×256 **resolution**

```
r_mmu_7:  defl $57
r_disp1:  defl $69
r_layer2: defl $70
start:
  nextreg r_disp1,$80  ; enable layer 2
  nextreg r_layer2,$10 ; 320x256x256
  ld a,$12             ; page 18=bank 9
```

```
loop1:
  nextreg r_mmu_7,a    ; map page into slot 7
  ld bc,$0020          ; 20*256 = 8k
  ld hl,$E000          ; start of slot 7
loop2:
  ld (hl),b
  inc hl
  djnz loop2
  dec c
  jp NZ,loop2
  inc a
  cp $1C               ; stop at page 28
  jp NZ,loop1
```

Layer 2 640×256 **resolution**

```
r_mmu_7:  defl $57
r_disp1:  defl $69
r_layer2:  defl $70
start:
  nextreg r_disp1, $80   ; enable layer 2
  nextreg r_layer2, $20  ; 640x256x16
  ld a, $12    ; page 18=bank 9
loop1:
  nextreg r_mmu_7, a  ; map page into slot 7
  ld bc, $0020    ; 20*256 = 8k
  ld hl, $E000    ; start address for slot 7
loop2:
  ld (hl), b
  inc hl
  djnz loop2
  dec c
  jp NZ, loop2
  inc a
  cp $1C      ; stop at page 28
  jp NZ, loop1
```

2.3.2 Scrolling

Scrolling Layer 2 is controlled by registers $16 and $17. (Is there a third scrolling register for layer 2?)

Register (R/W) $16 (22) ⇒ Layer 2 Horizontal Scroll Control

- bits 7-0 = X Offset (0-255)(0 on reset)

Register (R/W) $17 (23) ⇒ Layer 2 Vertical Scroll Control

- bits 7-0 = Y Offset (0-191)(0 on reset)

2.3.3 Clipping

The Clip area for is based on the local coordinate system for the mode in question and is set using register $18 with the option of selection which write in active using register $1C.

Register (R/W) $18 (24) ⇒ Layer 2 Clip Window Definition

- 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

Register (R/W) $1C (28) ⇒ Clip Window Control
Read

- bits 7-6 = Layer 3 Clip Index
- bits 5-4 = Layer 0/1 Clip Index
- bits 3-2 = Sprite clip index
- bits 1-0 = Layer 2 Clip Index

Write

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

2.4 Layer 3 (Tilemap) Mode

Started with documentation by Phoebus Dokos, February 25, 2019. Partially rewritten for clarity and to add core 3.00.00 features.

2.4.1 General Description

The tilemap is a hardware character oriented display. It uses a set of user defined 4-bit, 16-colour, or 1-bit, 2-colour 8×8 tiles. The tiles can be dispplayed in two resolutions: 40×32 tiles (320×256 pixels) and 80×32 tiles (640×256 pixels).

The display area on screen is the same as the sprite layer, meaning it overlaps the standard 256×192 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 40×30 or 80×30, 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 such as have historically been used in many games.

The tilemap is defined by two data structures and configured using four NextRegs. The NextRegs are $6b (107), Tilemap Control; $6c (108), Default Tilemap Attribute, $6c (110); Tilemap Base Address; and $6d (111) Tile Definitions Base Address.

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 either one or two bytes.

If entries are two bytes each, the first byte for each entry is bits 0-7 of the tile number, while the second byte is an attribute byte which is interpreted acctording to the mode set in the tilemap control register ($6b). For 40×32 resolution, a full size tilemap will occupy 2560 bytes, and for 80×32 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 consists of a tile number byte (bits 0-7 of the tile number) followed an attribute byte:

Tilemap Attribute Byte 4-bit

bits 7-4 : most significant 4-bits of palette entry
bit 3 : x mirror
bit 2 : y mirror
bit 1 : rotate
bit 0 : ULA over tilemap (in 512 tile mode, bit 8 of the tile number)

Tilemap Attribute Byte 1-bit

bits 7-1 : most significan 7-bits of palette entry
bit 0 : ULA over tilemap (in 512 tile mode, bit 8 of the 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 ($6b).

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.

When using 1-byte tilemap entries, the map consists of the tile numbers for tile in the map with the tilemap attribute byte for every tile coming from the default tilemap attribute register ($6c).

Tile Definitions The second data structure is the tile definitions themselves. To find the difinition for a specific tile you would look at (base address) + (tile number) * (definition size).

For 4-bit, 16-colour, tiles, each 8×8 tile takes up 32 bytes. Each pixel uses four bits to select one of 16 colours. A tile 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. Ane of the 16 colours for each tile is the transparency color.

For 1-bit, 2-colour, tiles, each 8×8 tile takes up 8 bytes. Each pixel uses one bit to select one of two colours. A tile is defined in X major order with packing in the X direction. The 1-bit colour of each pixel is augmented by the 7-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. Transparency for each tile is according to the global transparency colour.

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

Register (R/W) $6B (107) $\Rightarrow$ Layer 3 (Tilemap) Control

- bit 7 = Layer 3 Enable (0 on reset)
- bit 6 = Layer 3 Size control (0 on reset)
  - 0 = 40x32
  - 1 = 80x32
- bit 5 = Disable Arrtibute Entry (0 on reset)
- bit 4 = palette select (0 on reset)
- bit 3 = Enable Text mode (1-bit tilemap) (0 on reset)
- bit 2 = Reserved, must be 0
- bit 1 = Activate 512 tile mode (0 on reset)
- bit 0 = Enable Layer 3 on top of ULA (0 on reset)

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 5 determines whether the attribute byte for each tile come from the tilemap (0) or from the default tile attribute register (1).

Bit 4 selects either the primary tilemap palette (0) or the secondary tilemap palette (1).

Bit 3 selects whether to use 4-bit, 16-colour, or 1-bit 2-colour tiles.

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.

Register (R/W) $6C (108) ⇒ Default Layer 3 Attribute*

- bits 7-4 = Palette Offset ($00 on reset)
- bit 3 = X mirror (0 on reset)
- bit 2 = Y mirror (0 on reset)
- bit 1 = Rotate (0 on reset)
- bit 0 = Bit 8 of the tile number (512 tile mode) (0 on reset)
- bit 0 = ULA over tilemap (256 tile mode) (0 on reset)

*Active tile attribute if bit 5 of nextreg $6B is set.

If bit 5 of nextreg $6B is set, the tilemap 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.

Register (R/W) $6E (110) ⇒ Layer 3 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 ($2C on reset)

Soft Reset default $2C - This is because the address is $6C00 so the MSB is $6C. But the stored value is only the lower 6 bits so it's an offset into the 16k Bank 5. To calculate therefore subtract $40 leaving you with $2C.

The value written is an offset into the 16k 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.

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 40×32 or 80×32 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.

Register (R/W) $6F (111) ⇒ Layer 3 Tile Definitions Base Address

- bits 7-6 = Read back as zero, write values ignored
- bits 5-0 = MSB of address of the tilemap in Bank 5 ($0C on reset)

Soft Reset default $0C - This is because the address is $4C00 so the MSB is $4C. But the stored value is only the lower 6 bits so it's an offset into the 16k Bank 5. To calculate therefore subtract $40 leaving you with $0C.

The value written is an offset into the 16k 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.

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)

Register (R/W) $4C (76) ⇒ Level 3 Transparency Index

- bits 7-4 = Reserved, must be 0
- bits 3-0 = Index value ($0F on reset)

Defines the transparent colour index for 4-bit tiles. The 4-bit pixels of a tile definition are compared to this value to determine if they are transparent. In the case of 1-bit tiles transparency is determined by comparing the final pixel colour against the global transparency colour.

For palette information see palette section.

Register (R/W) $1B (27) ⇒ Layer 3 (Tilemap) Clip Window Definition

- 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.

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.

Register (R/W) $2F (47) ⇒ Layer 3 (Tilemap) Horizontal Scroll Control MSB

- bits 7-2 = Reserved, must be 0
- bits 1-0 = X Offset MSB ($00 on reset)

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

Register (R/W) $30 (48) ⇒ Layer 3 (Tilemap) Horizontal Scroll Control LSB

- bits 7-0 = X Offset LSB ($00 on reset)

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

Register (R/W) $31 (49) ⇒ Layer 3 (Tilemap) Vertical Scroll Control

- bits 7-0 = Y Offset (0-255))$00 on reset)

Register (R/W) $68 (104) ⇒ ULA Control

- bit 7 = Disable ULA output (0 on reset)
- bit 6-5 = Color blending control for layering modes 6 & 7
  - 00 = ULA as blend colour
  - 01 = No blending
  - 10 = ULA/Tilemap mix result as blend colour
  - 11 = Tilemap as blend colour
- bit 4 = Cancel entries in 8x5 matrix for extended keys (3.01.04)
- bit 3 = Enable ULAplus (0 on reset)
- bit 2 = Enable ULA half pixel scroll (0 on reset)
  may change
- bit 1 = Reserved (must be 0)
- bit 0 = Enable stencil mode (0 on reset)
  When ULA and Layer 3 are enabled, if either are transparent, the result is transparent, otherwise the result is the 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.
3. 1-bit tilemaps. In 3.00.00, a number of modifications were made to accomidate 1-bit tilemaps.

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 32×24 or 64×24 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 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 320×256 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 16×16 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 × 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×, 4×, and 8× 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.

2.5.1 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

Figure 2.1: Pattern Example

```
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:

```
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 16×16 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
3333333443334A4
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.

2.5.2 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.

2.5.3 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

```
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

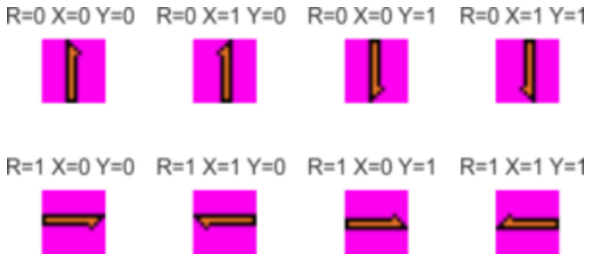

Figure 2.2: All Rotate and Mirror Flags

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.

2.5.4 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 16×16 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 16×16 in size with the anchor controlling the look of the whole.
A unified sprite is like a big version of an individual 16×16 sprite controlled by the anchor.

2.5.5 Programming Sprites

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

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

```
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 (87) Sprite Attributes
Byte 1

bits 7-0 = LSB of X coordinate (bit 8 is in byte 3)

Byte 2

bits 7-0 = LSB of Y coordinate (bit 8 is in byte 5)

Byte 3

bits 7-4 = Palette Offset
bit 3 = Enable X Mirror
bit 2 = Enable Y Mirror
bit 1 = Enable Roration
bit 0 = By Sprite Type
Anchor = MSB of X coordinate
Relative = Enable relative palette offset

Byte 4

bit 7 = Enable visibility
bit 6 = Enable Byte 5
bit 5-0 = Pattern Index (“name”)

Byte 5 (optional)
Anchor

bit 7-6 = type and pattern
00 = 8-bit color
01 = relative
10 = 4-bit color, lower half of pattern (bytes 0-127)
11 = 4-bit color, upper half of pattern (byets 128-255)
bit 5 = Attached relative sprite type
0 = composite
1 = big sprite
bit 4-3 = X-axis scale factor

00 = 1×
01 = 2×
10 = 4×
11 = 8×
bit 2-1 = Y-axis scale factor
bit 0 = MSB of Y coordinate

Composite Relative

bits 7-6 = 01
bit 5 = N6
8-bit
Reserved, must be 0
4-bit
0 = lower half of pattern (bytes 0-127)
1 = upper half of pattern (bytes 128-255)
bit 4-3 = X-axis scale factor
bit 2-1 = Y-axis scale factor
bit 0 = Enable relative pattern offset

Big-sprite Relative

bits 7-6 = 01
bit 5 = N6
8-bit
Reserved, must be 0
4-bit
0 = lower half of pattern (bytes 0-127)
1 = upper half of pattern (bytes 128-255)
bit 4-1 = Reserved, must be 0
bit 0 = Enable relative pattern offset

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 (91) Sprite Pattern
Load data into sprite pattern memory auto-incrementing. Port $303B can be used to set the starting sprite pattern number.

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.

Register (R/W) $34 (52) ⇒ Sprite Number
Lockstep (NextReg $09 bit 4 set)

- bit 7 = Pattern address offset (Add 128 to pattern address)
- bits 6-0 = Sprite number 0-127, Pattern number 0-63
  effectively performs an out to port $303B

No Lockstep (NextReg $09 bit 4 clear)

- bit 7 = Reserved, must be 0
- bits 6-0 = Sprite number 0-127

This register selects which sprite has its attributes connected to the sprite attribute registers

Register (W) $35 (53) ⇒ Sprite Attribute 0

- bits 7-0 = Sprite X coordinate LSB (MSB in NextReg $37)

Register (W) $75 (117) ⇒ Sprite Attribute 0 (Auto-incrementing)
See nextreg $35

Register (W) $36 (54) ⇒ Sprite Attribute 1

- bits 7-0 = Sprite Y coordinate LSB (MSB in NextReg $39)

Register (W) $76 (118) ⇒ Sprite Attribute 1 (Auto-incrementing)
See nextreg $36

Register (W) $37 (55) ⇒ Sprite Attribute 2

- bits 7-4 = 4-bit Palette offset
- bit 3 = Enable horizontal mirror (reverse)
- bit 2 = Enable vertical mirror (reverse)
- bit 1 = Enable 90^O Clockwise Rotation

Normal Sprites

- bit 0 = X coordinate MSB

Relative Sprites

- bit 0 = Palette offset is relative to anchor sprite

Rotation is applied before mirroring

Register (W) $77 (119) ⇒ Sprite Attribute 2 (Auto-incrementing)
See nextreg $37

Register (W) $38 (56) ⇒ Sprite Attribute 3

- bit 7 = Enable Visiblity
- bit 6 = Enable Attribute 4 (0 = Attribute 4 effectively $00)
- bits 5-0 = Sprite Pattern Number

Register (W) $78 (120) ⇒ Sprite Attribute 3 (Auto-incrementing)
See nextreg $38

Register (W) $39 (57) ⇒ Sprite Attribute 4
Normal Sprites

- bit 7 = 4-bit pattern switch (0 = 8-bit sprite, 1 = 4-bit sprite)
- bit 6 = Pattern number bit-7 for 4-bit, 0 for 8-bit
- bit 5 = Type of attached relative sprites (0 = Composite, 1 = Unified)
- 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

Relative, Composite Sprites

- bit 7-6 = 01
- bit 5 = Pattern number bit-7 for 4-bit, 0 for 8-bit
- 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 = Pattern number is relative to anchor

Relative, Unified Sprites

- bit 7-6 = 01
- bit 5 = Pattern number bit-7 for 4-bit, 0 for 8-bit
- bits 4-1 = 0000
- bit 0 = Pattern number is relative to anchor

Register (W) $79 (121) ⇒ Sprite Attribute 4 (Auto-incrementing)
See nextreg $39

2.5.6 Global Control of Sprites

The following nextreg are also of interest for sprites.

Register (R/W) $09 (9) ⇒ Peripheral 4 setting:

- bit 7 = PSG 2 Mono Enable (0 on hard reset)
- bit 6 = PSG 1 Mono Enable (0 on hard reset)
- bit 5 = PSG 0 Mono Enable (0 on hard reset)
- bit 4 = Sprite ID lockstep enable (1 = Nextreg $34 and IO Port $303B are in lockstep, 0 on reset)
- bit 3 = divMMC mapRAM bit Control (reset bit 7 of port $E3)
- bit 2 = HDMI audio mute (0 on hard reset)
- bits 1-0 = scanlines
  - 00 = scanlines off
  - 01 = scanlines 12.5%
  - 10 = scanlines 25%
  - 11 = scanlines 50%

  In Sprite lockstep, NextREG $34 and Port $303B are in lockstep

Register (R/W) $15 (21) ⇒ Sprite and Layer System Setup

- bit 7 = LoRes mode (0 on reset)
- bit 6 = Sprite priority (1 = sprite 0 on top, 0 = sprite 127 on top) (0 on reset)
- bit 5 = Enable sprite clipping in over border mode (0 on reset)
- bits 4-2 = set layers priorities (000 on reset)
  - 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 = Enable Sprites Over border (0 on reset)
- bit 0 = Enable Sprites (0 on reset)

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.

Register (R/W) $19 (25) ⇒ Sprite Clip Window Definition

- 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.

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.

Register (R/W) $1C (28) ⇒ Clip Window Control
Read

- bits 7-6 = Layer 3 Clip Index
- bits 5-4 = Layer 0/1 Clip Index
- bits 3-2 = Sprite clip index

- bits 1-0 = Layer 2 Clip Index

Write

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

Register (R/W) $4B (75) ⇒ Sprite Transparency Index

- bits 7-0 = Index value ($E3 if reset)

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

Chapter 3

Audio

3.1 ZX Spectrum 1-bit

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 Sampled 8-bit

The ZX Next has four 8-bit D/A audio channels connected to provide sampled stereo sound. Channels A and B are the left channels, while C and D are the right channels. In order use 8-bit sound, it must first be enabled by

setting bit 3 on nextreg $08. In order to emulate legacy hardware there are a number of ports that can be used to control the four channels additionally these are mirrored to three nextregs to enable driving audio using the copper. Channel A is mapped to ports $0f, $3f, and $f1; channel B to ports $1f and $f3 and nextreg $2C; channel C to ports $4f, and $f9 and nextreg $2E; and channel D to: $5f and $fb; with port $df connected to both channel A and C and nextreg $2D connected to both channel A and D.

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).

Register (R/W) $00 (0) ⇒ Channel A fine tune

Register (R/W) $01 (1) ⇒ Channel A coarse tune (4 bits)

Register (R/W) $02 (2) ⇒ Channel B fine tune

Register (R/W) $03 (3) ⇒ Channel B coarse tune (4 bits)

Register (R/W) $04 (4) ⇒ Channel C fine tune

Register (R/W) $05 (5) ⇒ Channel C coarse tune (4 bits)

Register (R/W) $06 (6) ⇒ Noise period (5 bits)

Register (R/W) $07 (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)

Register (R/W) $08 (8) ⇒ Channel A amplitude

- bit 4 = enable fixed amplitude
  - 0 = fixed amplitude
  - 1 = use envelope generator (bits 0-3 ignored)
- bits 3-0 = value of fixed amplitude

Register (R/W) $09 (9) ⇒ Channel B amplitude

- bit 4 = enable fixed amplitude
  - 0 = fixed amplitude
  - 1 = use envelope generator (bits 0-3 ignored)
- bits 3-0 = value of fixed amplitude

Register (R/W) $0A (10) ⇒ Channel C amplitude

- bit 4 = enable fixed amplitude
  - 0 = fixed amplitude
  - 1 = use envelope generator (bits 0-3 ignored)
- bits 3-0 = value of fixed amplitude

Register (R/W) $0B (11) ⇒ Envelope period fine

Register (R/W) $0C (12) ⇒ Envelope period coarse

Register (R/W) $0D (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

3.3.1 Pi Audio

If connected the Pi Zero is configured to use the ZX Next as a soundcard over an I^2S interface making the Raspberry Pi a fully configurable audio source for the ZX Spectrum Next.

Chapter 4

Memory

The ZX Spectrum Next commonly has with either 1MB or 2MB SRAM memory. This is more the 64kB directly addressable by its Z80N CPU. It is therefore necessary to use some form of memory paging to address all of the memory. This is accomplished using 8k pages or 16k banks. 256k of the total memory is used by the ROM images and device specific RAM leaving either 768k (pages 0-95/banks 0-47) or 1792k (pages 0-223/banks 0-111) that can be paged in as RAM. Pages 10, 11, and 14 are a little odd in that rather than coming from the normal SRAM, they come from BRAM internal to the FPGA.

4.1 Memory Management

There are a number of different systems for controling memory papping into the 64k memory space of the Z80N CPU in the ZX Next: ZX Next native memory paging, ZX Spectrum 128, ZX Spectrum +3, divMMC, and Multiface.

4.1.1 Default Layout

The default mapping of memory banks is the same as on 128k Spectrum models with a ROM0 (128k editor and menu system) mapped in at $0000-$3FFF, bank 5 at $4000-$7FFF, bank 2 at $8000-$BFFF, and bank 0 at $C000-$FFFF.

4.1.2 RAM

ZX Spectrum Next Native Registers $50 to $57 control the which SRAM pages are in each of the eight memory slots. Registers $50 and $51 support the special value $FF which indicates that the currently selected ROM is to be mapped into slots 0 and/or 1 ($0000-$3FFF).

Register (R/W) $50 (80) ⇒ MMU Slot 0 Control

- bits 7-0 = 8k RAM page at position $0000 to $1FFF ($ff on reset)

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

Register (R/W) $51 (81) ⇒ MMU Slot 1 Control

- bits 7-0 = 8k RAM page at position $2000 to $3FFF ($ff on reset)

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

Register (R/W) $52 (82) ⇒ MMU Slot 2 Control

- bits 7-0 = 8k RAM page at position $4000 to $5FFF ($0A on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $53 (83) ⇒ MMU Slot 3 Control

- bits 7-0 = 8k RAM page at position $6000 to $7FFF ($0B on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $54 (84) ⇒ MMU Slot 4 Control

- bits 7-0 = 8k RAM page at position $8000 to $9FFF ($04 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $55 (85) ⇒ MMU Slot 5 Control

- bits 7-0 = 8k RAM page at position $A000 to $BFFF ($05 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $56 (86) ⇒ MMU Slot 6 Control

- bits 7-0 = 8k RAM page at position $C000 to $DFFF ($00 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $57 (87) ⇒ MMU Slot 7 Control

- bits 7-0 = 8k RAM page at position $E000 to $FFFF ($01 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Writing to ports $1FFD, $7FFD and $DFFD writes $FF 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.

In addition the ZX Next has special controls which allow the data area for Layer 2 to be overlaied on memory in a fashion that permits selective read or write access. For details see the section on Layer 2 video.

ZX Spectrum 128 In addition to the native memory management, the ZX Next supports a memory management system that is an expanded, and backward compatible, version of the the system used on earlier ZX Spectrum models. This system uses registers $1FFD, $7FFD, and $DFFD.

Port $1FFD (8189) Plus 3 Memory Paging Control

bits 7-3 = Unused, nust be 0
bit 2 = High bit of ROM selection (low bit is in Port $7FFD)
  00 = ROM0 = 128k editor and menu system
  01 = ROM1 = 128k syntax checker
  10 = ROM2 = +3DOS
  11 = ROM3 = 48k BASIC
bit 1 = Special mode: Low bit of memory configuration number
bit 0 = Paging mode
  0 = Normal
  1 = Special

You should echo writes to $5B67

Port $7FFD (32765) Memory Paging Control

bits 6-7 = reserved
bit 5 = Lock memory paging
bit 4 = low bit of ROM Select (high bit is in Port $1FFD)
  00 = ROM0 = 128k editor and menu system
  01 = ROM1 = 128k syntax checker
  10 = ROM2 = +3DOS
  11 = ROM3 = 48k BASIC
bit 3 = Shadow screen toggle

bits 2-0 = LSB of Bank number for slot 4 (MSB is in Port $DFFD)

Disable with bit 5 port $7FFD

Port $DFFD (57341) Next Memory Bank Select

bits 7-4 = Reserved, must be 0
bits 3-0 = MSB of bank number for slot 4 (LSB is in Port $7FFD)

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

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.

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.

Spectrum 128 Special Paging "Special paging mode" (also called "All-Ram 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

4.1.3 ROM

The ZX Spectrum Next had several ROMS: ROM0 (16k) - 128k editor and menu system, ROM1 (16k) - 128k syntax checker, ROM2 (16k) - +3DOS, ROM3 (16k) - 48k BASIC, divMMC/esxDOS ROM (8k), divMMC RAM (128k), Multiface ROM (8k) and Alternate ROM (16k).

ZX Next native Slots 0 and 1 select use by ROM by selecting page $FF. Which what ROM is mapped in is determined by the other memory management system. If the rest of the system selected the 48k ROM, Nextreg $8C determines whether the actual 48k ROM, or the ZX Next Alternate ROM is in use. In addition, it is possible to enable writing to the Alternate ROM.

Register (R/W) $8C (140) ⇒ Alternate ROM
Immediate

- bit 7 = Alt ROM Enable (0 on hard reset)
- bit 6 = Alt ROM visible ONLY during writes (0 on hard reset)
- bit 5 = Reserved, must be 0
- bit 4 = 48k ROM Lock (0 on hard reset)

After Soft Reset (copied into bits 7-4)

- bit 3 = Alt ROM Enable (0 on hard reset)
- bit 2 = Alt ROM visible ONLY during writes (0 on hard reset)
- bit 1 = Reserved, must be 0
- bit 0 = 48k ROM Lock (0 on hard reset)

ZX Spectrum 128k

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.

divMMC The divMMC ROM mapping can take priority when it is enabled by port $E3 or, when automapping has been enabled by nextreg $06, when it has been automapped due to reading one of the appropriate addresses. Port $E3 also controls whether the divMMC maps the esxDOS ROM or divMMC RAM page 3 into slot 0 and which divMMC RAM page is mapped into slot 1.

Port $E3 (227) divMMC Control
Disable with bit 2 of Nextreg $09

bit 7 = conmem, enable divMMC memory
bit 6 = mapram, enable divMMC allRAM mode
bits 3-0 = bank, selected divMMC ram bank for $2000-$3FFF region

- conmem can be used to manually control divMMC mapping. When enabled
  $0000-$1FFF contains esxDOS ROM or esxDOS page 3
  $2000-$3FFF contains esxDOS RAM page selected by bits 3-0
- divMMC automatically maps itself in when instruction fetches hit specific addresses in the ROM. When this happens, the esxDOS ROM (or divMMC bank 3 if mapram is set) appears in $0000-$1FFF and the selected divMMC bank appears as RAM in $2000-$3FFF
- bit 6 can only be set, once set only a power cycle can reset it. nextreg $09 bit 3 can be set to reset this bit.

divMMC automapping is normally disabled by NextZXOS see nextreg $06 bit 4.
Register (R/W) $06 (6) ⇒ Peripheral 2 Settings

- bit 7 = F8 CPU Speed Hotkey Enable (1 on reset)
- bit 6 = Enable classic audio mode (beep and tape to internal speaker, other audio to ear and HDMI, 3.01.02)

- bit 5 = F3 50Hz/60Hz Hotkey Enable (1 on reset)
- bit 4 = divMMC Automap/NMI Enable (0 on hard reset)
- bit 3 = NMI Button Enable (0 on hard reset)
- bit 2 = PS/2 Mode (0 = keyboard, 1 = mouse)
- bits 1-0 = PSG Mode (00 = YM, 01 = AY, 11 = hold all PSGs in Reset)

Multiface Need to find useful docs on the Multiface memory.

9f 1-In, 128-In2
1f 1-Out
bf 128-In, 3-Out
3f 128-Out, 3-In, 3-button
7f3f 3-7ffd
1f3f 3-1ffd

4.2 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.

4.3 Memory Map

4.3.1 Global Memory Map

Physical Address	Size	Description
$000000-$00FFFF	64k	ZX Spectrum ROM (ROM0-3)
$010000-$011FFF	8k	EsxDOS ROM
$012000-$013FFF	8k	Multiface ROM
$014000-$017FFF	16k	EsxDOS Extra ROM
$018000-$01BFFF	16k	Alternate ROM
$01C000-$01FFFF	16k	Multiface RAM
$020000-$03FFFF	128k	DivMMC RAM
$040000-$0FFFFF	768k	Standard RAM
$100000-$1FFFFF	1024k	Expanded RAM

Normal RAM is divided into 8k pages or 16k banks which may be mapped into the 64k memory space by the memory management hardware of the Next. Some of these pages have special properties.

Pages 10, 11 and 14 are used by Layer 1/0 (ULA) video modes with 10 used by standard Spectrum ULA video, 10 and 11 used by Timex Hi-res and Hi-colour modes, 11 used by Timex alternate video and page 14 used by the ULA shadow mode. Pages 10 an 11 are usable by Layer 3 (Tilemap) video.

4.3.2 Z80 Visible Memory Map

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,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 256×192 which can be doubled to 512×192 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 256×192 and Timex HIRES 512×192. Details of LORES 128×96 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 256×192 and 512×192 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 256×192 resolution the duration of one pixel is two dot clocks. In Timex HIRES 512×192 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 256×192 resolution is line 191 and horizontal dot clocks 510+511.

The bottom right pixel of the frame buffer in Timex HIRES 512×192 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 256×192 resolution and every sixteen pixels in Timex HIRES 512×192 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 × 96 resolution

16 dot clocks = 8 pixels in standard 256 × 192 resolution

16 dot clocks = 16 pixels in high 512 × 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.

Register (R/W) $05 (5) ⇒ Peripheral 1 Settings

- bits 7-6 = joystick 1 mode (MSB)
- bits 5-4 = joystick 2 mode (MSB)
- bit 3 = joystick 1 mode (LSB)
- bit 2 = 50/60 Hz mode (0 = 50Hz, 1 = 60Hz)
- bit 1 = joystick 2 mode (LSB)
- bit 0 = Enable Scandoubler

Joystick modes

- 000 = Sinclair 2 (67890)
- 001 = Kempston 2 (port $37)
- 010 = Kempston 1 (port $1F)
- 011 = Megadrive 1 (port $1F)
- 100 = Cursor
- 101 = Megadrive 2 (port $37)
- 110 = Sinclair 1 (12345)
- 111 = I/O Mode Both joysticks are places in I/O Mode if either is set to I/O Mode. The underlying joystick type is not changed and reads of this register will continue to return the last joystick type. Ehether the joystick is in io mode or not is invisible but this state can be cleared either through reset or by re-writing the gegister with joystick type not equal to 111. Recovery time for a normal joystick read after leaving

I/O Mode is at most 64 scan lines.

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 register allows software to read the video timing mode:

Register (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
  Only writable in config mode

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:

Register (R/W) $03 (3) ⇒ Machine Type
A write to this register disables the boot rom in config mode
bits 2-0 select machine type when in config mode

- bit 7 = (W) Display Timing change enable (allow changes to bits 6-4) (0 on hard reset)
- bits 6-4 = Display Timing
- bit 3 = Display Timing user lock control
  Read
  - 0 = No user lock on display timing
  - 1 = User lock on display timing

  Write
  - 1 = Apply user lock on display timing (0 on hard reset)
- bits 2-1 = Machine Type
  Machine Types/Display Timings

- 000 or 001 = ZX 48K
- 010 = ZX 128K/+2 (Grey)
- 011 = ZX +2A-B/+3e/Next Native
- 100 = Pentagon 128K

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 has a header (with male pins) which can be attached to a Raspberry Pi Zero. There is a modified version of DietPi called NextPi which is the standard distro for the Raspberry Pi0 accelerator. Software for the general public should be written assuming that it will be interfacing with a Pi0 running this distro.

If you are more adventurous, you may choose to use another distro, or even another accelerator that uses the Raspberry Pi style (40 pin) expansion bus. Chief concers when doing this is that you have a console presented on the UART that defaults to 115,200 bps, you don't need to login, the machine is configured with a driver to treat the I^2S interface as a sound card, and the presence of the nextpi scripts.

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. For this application the Pi 0 ships with a 1GB microSD card containing NextPi a customized version of DietPi.

The Pi Zero, if installed, is a smart peripheral for the ZX Spectrum Next. Available interfaces are: low level access to the GPIO pins, higher level access to standardized I/O interfaces, and use of the Pi Zero as a sound card.

When using the low level GPIO interface Pi Zero GPIO pins 2-27 can be configured as either inputs or outputs using nextregs $90-$93. If they are outputs, the output state can be set by writing to nextregs $98-$9b. The current status of the GPIO pins can be read from nextregs $98-$9b whether

it is the state driven by the ZX Spectrum Next or the state drive by some other peripherial attached to the bus (normally the Raspberry Pi Zero).

Standardized I/O access with the Pi Zero can use the I^2C, SPI, or UART interfaces and is configured using nextreg $a0. Any enabled port will disable low level (write) access to the corresponding GPIO pins.

The I^2C interface is controlled using ports $103b (SCL) and $113b (SDA). This is the same I^2C interface that is used for the optional Real Time Clock. Interfacing with the Pi Zero over I^2C is complicated by the fact that it is a master/slave interface, but both the ZX Spectrum Next and Pi Zero are configured to be bus masters.

The SPI interface is controlled using ports $e7 (/CS) and $eb (/DATA). The SPI interface is shared between the SD card(s), the flash memory, and the Pi Zero. Interfacing with the Pi Zero over SPI is complicated by the fact it is a master/slave interface and both the ZX Spectrum Next and Pi Zero are configured to be bus masters.

The UART interface is controlled by ports $133b (TX), $143b (RX), and $153b (control). This is the default means of bidirectional communication between the ZX Spectrum Next and Pi Zero. To use the UART you must first enable the UART on the GPIO and connecting it to the Pi Zero (not hats) by setting nextreg $a0 bits 5 and 4 to 1 and selecting Pi for the UART by setting port $153b bit 6 to 1. Assuming that the serial parameters match (by default both ends are set to 115,200 bps with 8N1 and no flow control) this will give you access to the serial console on the Pi Zero over the UART.

```
;; enable UART connection with Pi Zero
   ld c,$3b
   ld b,$15 ; UART control
;; select Pi on UART control
   in a,(c)
   or $40
   out (c),a
   ld b,$24 ; Next Register Select
   ld a,$a0
   out (c),a
   inc b ; Next Register Data
;; Enable UART on GPIO and select Pi
   in a,(c)
   or $30
```

```
out (c),a
```

The I^2S sound interface between the ZX Spectrum Next and the Pi Zero is controlled by nextregs $a2 and $a3. Normally, one would control the Pi through some other channel such as the UART recieve audio from the Pi to either use as a fulloy programmable sound card or to allow loading of tape files on the ZX Spectrum Next.

(R/W) $90 (144) ⇒ PI GPIO Output Enable 0

- bit 7 = Pi GPIO 7
- bit 6 = Pi GPIO 6
- bit 5 = Pi GPIO 5
- bit 4 = Pi GPIO 4
- bit 3 = Pi GPIO 3
- bit 2 = Pi GPIO 2
- bit 1 = Pi GPIO 1 (cannot be enabled)
- bit 0 = Pi GPIO 0 (cannot be enabled)
  $00 on soft reset

(R/W) $91 (145) ⇒ PI GPIO Output Enable 1

- bit 7 = Pi GPIO 15
- bit 6 = Pi GPIO 14
- bit 5 = Pi GPIO 13
- bit 4 = Pi GPIO 12
- bit 3 = Pi GPIO 11
- bit 2 = Pi GPIO 10
- bit 1 = Pi GPIO 9
- bit 0 = Pi GPIO 8
  $00 on soft reset

(R/W) $92 (146) ⇒ PI GPIO Output Enable 2

- bit 7 = Pi GPIO 23
- bit 6 = Pi GPIO 22
- bit 5 = Pi GPIO 21
- bit 4 = Pi GPIO 20
- bit 3 = Pi GPIO 19
- bit 2 = Pi GPIO 18
- bit 1 = Pi GPIO 17
- bit 0 = Pi GPIO 16
  $00 on soft reset

(R/W) $93 (147) ⇒ PI GPIO Output Enable 3

- bits 7-4 = Reserved, must be 0
- bit 3 = Pi GPIO 27
- bit 2 = Pi GPIO 26
- bit 1 = Pi GPIO 25
- bit 0 = Pi GPIO 24

  $00 on soft reset

(R/W) $98 (152) ⇒ PI GPIO 0

- bit 7 = Pi GPIO 7
- bit 6 = Pi GPIO 6
- bit 5 = Pi GPIO 5
- bit 4 = Pi GPIO 4
- bit 3 = Pi GPIO 3
- bit 2 = Pi GPIO 2
- bit 1 = Pi GPIO 1 (read only)
- bit 0 = Pi GPIO 0 (read only)

  $ff on soft reset

(R/W) $99 (153) ⇒ PI GPIO 1

- bit 7 = Pi GPIO 15
- bit 6 = Pi GPIO 14
- bit 5 = Pi GPIO 13
- bit 4 = Pi GPIO 12
- bit 3 = Pi GPIO 11
- bit 2 = Pi GPIO 10
- bit 1 = Pi GPIO 9
- bit 0 = Pi GPIO 8

  $01 on soft reset

(R/W) $9A (154) ⇒ PI GPIO 2

- bit 7 = Pi GPIO 23
- bit 6 = Pi GPIO 22
- bit 5 = Pi GPIO 21
- bit 4 = Pi GPIO 20
- bit 3 = Pi GPIO 19
- bit 2 = Pi GPIO 18
- bit 1 = Pi GPIO 17
- bit 0 = Pi GPIO 16

$00 on soft reset

(R/W) $9B (155) ⇒ PI GPIO 3

- bits 7-4 = Reserved, must be 0
- bit 3 = Pi GPIO 27
- bit 2 = Pi GPIO 26
- bit 1 = Pi GPIO 25
- bit 0 = Pi GPIO 24
  $00 on soft reset

(R/W) $A0 (160) ⇒ PI Peripheral Enable

- bits 7-6 = Reserved, must be 0
- bit 5 = Enable UART on GPIO 14,15 (overrides gpio) (soft reset = 0)
- bit 4 = 0 to connect Rx to GPIO 15, Tx to GPIO 14 (for comm with pi hats) (soft reset = 0) = 1 to connect Rx to GPIO 14, Tx to GPIO 15 (for comm with pi)
- bit 3 = Enable I^2C on GPIO 2,3 (override gpio) (soft reset = 0)
- bits 2-1 = Reserved, must be 0
- bit 0 = Enable SPI on GPIO 7,8,9,10,11 (overrides gpio) (soft reset = 0)

(R/W) $A2 (162) ⇒ PI I^2S Audio Control

- bits 7-6 = I^2S enable (soft reset = 00)
  - 00 = I^2S off
  - 01 = I^2S is mono source right
  - 10 = I^2S is mono source left
  - 11 = I^2S is stereo
- bit 5 = Reserved, must be 0
- bit 4 = 0 PCM_DOUT to pi, PCM_DIN from pi (hats) (soft reset = 0) = 1 PCM_DOUT from pi, PCM_DIN to pi (pi)
- bit 3 = Mute left side (soft reset = 0)
- bit 2 = Mute right side (soft reset = 0)
- bit 1 = Slave mode (PCM_CLK, PCM_FS supplied externally) (soft reset = 0)
- bit 0 = Direct I^2S audio to EAR on port $FE (soft reset = 0)

(R/W) $A3 (163) ⇒ PI I^2S Clock Divide (Master Mode)

- bits 7-0 = Clock divide sets sample rate when in master mode (soft reset = 11)
  clock divider = 538461 / SampleRateHz - 1

Chapter 9

System Software

9.1 CP/M

The ZX Spextrum Next has support for CP/M+ 3.0. CP/M was the most popular microcomputer coperating system prior to the advent of MS-DOS.

9.1.1 Utilities

From the Digital Research: CP/M 3 Command Reference Manual 1984

This section documents all standard CP/M+ 3 commands plus those extras included with the ZX Spectrum Next CP/M system.

COLOURS

Syntax: **COLOURS [RGB]** ***paper ink***

Function: Sets the screen colours

Parameters:

paper	Paper (background) colour
ink	Ink (foreground) colour

Options:

RGB	Causes ink and paper colours to be interpreted as 9-bit octal RGB numbers

Notes: Sets the screen colours using standard ZX colours or octal 9-bit

RGB numbers.

Examples:

colours 1 6

colours rgb 000 750

COPYSYS

Syntax: **COPYSYS**

Function: Copy CP/M system

Notes: COPYSYS copies the CP/M Plus system from a CP/M Plus system diskette to another diskette. The new diskette must have the same format as the original system diskette.

DATE

Syntax: **DATE**
DATE C
DATE CONTINUOUS
DATE *time-specification*
DATE SET

Function: The DATE command lets you display and set the date and time of day.

Parameters:

time-specification Time/date in the format MM/DD/YY HH:MM:SS

Options:

C Continuously show the date and time until a key is pressed

CONTINUOUS Continuously show the date and time until a key is pressed

SET Prompt the user for the current date and time

Notes: The DATE command is a transient utility that lets you display and set the date and time of day. When you start CP/M 3, the date and time are set to the creation date of your CP/M 3 system. Use DATE to change this initial value to the current date and time.

Examples:

DATE

DATE C

DATE CONTINUOUS

DATE 08/13/82 09:15:37

DATE SET

DEVICE

Syntax:

DEVICE
DEVICE NAMES
DEVICE VALUES
DEVICE logical-dev {XON|NOXON|baud-rate},
DEVICE physical-dev {XON|NOXON|baud-rate}
DEVICE logical-dev=physical-dev {option} {,physical-dev {option},...}
DEVICE logical-dev = NULL
DEVICE CONSOLE {PAGE}
DEVICE CONSOLE {COLUMNS=n, LINES=n}

DEVICE displays current logical device assignments and physical device names.

DIR (built-in)

Syntax:

DIR
DIR d:
DIR filespec
DIR d: options
DIR filespec,... filespec options

The DIR command displays the names of files catalogued in the directory of an online disk that belong to current user number and have the Directory (DIR) attribute. DIR accepts the * and ? wildcards in the file specification.

The DIR command with options displays the names of files and the characteristics associated with the files. DIR is a built-in utility. DIR with options is a transient utility and must be loaded into memory from the disk.

DIRSYS/DIRS (built-in)

Syntax:

DIRSYS
DIRSYS d:
DIRSYS filespec

The DIRSYS command lists the names of files in the current directory that have the system (SYS) attribute. DIRSYS accept the * and ? wildcards in the file specification. DIRSYS is a built-in utility.

DUMP

Syntax:
    DUMP filespec

DUMP displays the contents of a file in and ASCII format.

ECHO (ZX Spectrum Next)

Syntax:
    ECHO string

Echo characters to the terminal

The following special character sequences may be used

- \a alert (bell) (ASCII 7)
- \b backspace (ASCII 8)
- \e escape (ASCII 27)
- \n line feed (ASCII 10)
- \r carriage return (ASCII 13)
- \l interpret further characters as lower-case
- \u interpret further characters as upper-case
- \\backslash ('\')

Note that CP/M converts all your typed characters to upper-case before providing them to ECHO.COM. Therefore you will need to use \l and \u to specify the case of characters if it is important (in ESCape sequences, for example).

ED

Syntax:
    ED
    ED input-filespec
    ED input-filespec {d: | output-filespec}

Character file editor. To redirect or rename the new version of the file specify the destination drive or destination filespec.

ERASE/ERA (built-in)

Syntax:

ERASE
ERASE filespec
ERASE filespec [CONFIRM]

The ERASE command removes one or more files from the directory of a disk. Wildcard characters are accepted in the filespec. Directory and data space are automatically reclaimed for later use by another file. The ERASE command can be abbreviated to ERA.

[CONFIRM] option informs the system to prompt for verification before erasing each file that matches the filespec. CONFIRM can be abbreviated to C.

EXIT (ZX Spectrum Next)

Syntax:
EXIT

The EXIT command leaves CP/M (rebooting the ZX Spectrum Next)

EXPORT (ZX Spectrum Next)

Syntax:
EXPORT cpm-filespec nextzxos-filespec

NextZXOS file export utility

Export file to a NextZXOS drive.

GENCOM

Syntax:
GENCOM COM-Eilespec RSX-filespec... RSX-Eilespec {[LOADER | SCB=(Offset,value)]}
GENCOM RSX-filespec ... RSX-filespec {[NULL | SCB=(Offset,value)l}
GENCOM filename
GENCOM filename [SCB=(offset,value)]

The GENCOM command attaches RSX files to a COM file, or creates a dummy COM file containing only RSXS. It can also restore a previously GENCOMed file to the original COM file without the header and RSXS, add or replace RSXs in already GENCOMed files, and attach header records to COM files without RSXS.

GENCPM

Syntax:

GENCPM {AUTO|AUTO DISPLAY}

GENCPM creates a memory image CPM3.SYS file, containing the CP/M 3 BDOS and customized BIOS. The GENCPM utility performs late resolution of intermodule references between system modules. GENCPM can accept its command input interactively from the console or from a file GENCPM.DAT.

In the nonbanked system, GENCPM creates a CPM3.SYS file from the BDOS3.SPR and BIOS3.SPR files. In the banked system, GENCPM creates the CPM3.SYS file from the RESBDOS3.SPR, the BNKBDOS3.SPR and the BNKBIOS3.SPR files. Remember to back up your CPM3.SYS file before executing GENCPM, because GENCPM deletes any existing CPM3.SYS file before it generates a new system.

GET

Syntax:

GET {CONSOLE INPUT FROM} FILE filespec options
GET {CONSOLE INPUT FROM} CONSOLE

GET directs the system to take console input from a file for the next system comand or user program entered at the console.

Console input is taken from a file until the program terminates. If the file is exhausted before program input is terminated, the program looks for subsequent input from the console. If the program terminates before exhausting all its input, the system reverts back to the console for console input.

HELP

Syntax:

HELP
HELP topic
HELP topic subtopic
HELP topic [NOPAGE]
HELP topic subtopic1...subtopic8
HELP>topic
HELP>.subtopic

HELP displays a list of topics and provides summarized information for CP/M Plus commands.

Typing HELP topic displays information about that topic. Typing HELP topic subtopic displays information about that subtopics One or two letters

is enough to identify the topics. After HELP displays information for your topic, it displays the special prompt HELP> on your screen, followed by a list of subtopics.

- Enter ? to display list of main topics.
- Enter a period and subtopic name to access subtopics.
- Enter a period to redisplay what you just read.
- Press RETURN to return to the CP/M Plus system prompt.
- [NOPAGE] option disables the 24 lines per page console display.
- Press any key to exit a display and return to the HELP> prompt.

HEXCOM

Syntax:
    HEXCOM filename

The HEXCOM Command generates a command file (filetype COM) from a HEX input file. it names the output tile with the same filename as the input file but with filetype COM. HEXCOM always looks for a file with filetype HEX.

IMPORT (ZX Spectrum Next)

Syntax:
    IMPORT nextzxos-filespec
    IMPORT nextzxos-filespec cpm-filespec

NextZXOS file import utility

List or import files from a NextZXOS drive.

INITDIR (Not included)

Syntax:
    INITDIR d:

The INITDIR command initializes a disk directory to allow date and time stamping of files on that disk. INITDIR can also recover time/date directory space.

NEXTREG (ZX Spectrum Next)

Syntax:
    NEXTREG register {value}

NextReg Utility

Show or change a NextReg register (use at your own risk!)

LIB (Not included)

Syntax:

LIB filespec options
LIB filespec options=filespec <modifier> f,filespec<modifier>

A library is a file that contains a collection of object modules.

Use the LIB utility to create libraries, and to append, replace, select, or delete modules from an existing library. Use LIB to obtain information about the contents of library files. LIB creates and maintains library files that contain object modules in Microsoft REL file format. These modules are produced by the Digital Research relocatable macro-assembler program, RMAC, or other language translator that produces modules in Microsoft REL file format.

You can use LINK-80 to link the object modules contained in a library to other object files. LINK-80 automatically selects from the library only those modules needed by the program being linked, and then forms an executable file with a filetype of Com.

LINK (Not included)

Syntax:

LINK filespec [options]
LINK filespec [options],...filespec [options]
LINK filespec [options]=filespec [options],...

LINK combines relocatable object modules such as those produced by RMAC and PL/I- 80 into a COM file ready for execution. Relocatable files can contain external references and publics. Relocatable files can reference modules in library files. LINK searches the library files and includes the referenced modules in the output file. See the Programmer's Utilities Guide for the CP/M Family of Operating Systems for a complete description of LINK-80.

Use LINK option switches to control execution parameters. Link options follow the file specifications and are enclosed within square brackets. Multiple switches are separated by commas.

MAC (Not included)

Syntax:

MAC filename [$options]

MAC, the CP/M Plus macro assembler, reads assembly language statements

from a file of type ASM, assembles the statements, and produces three output files with the input filename and filetypes of HEX, PRN, and SYM. Filename.HEX contains Intel hexadecimal format object code. Filename.PRN contains an annotated source listing that you can print or examine at the console. Filename.SYM contains a sorted list of symbols defined in the program.

Use options to direct the input and output of MAC. Use a letter with the option to indicate the source and destination drives, and console, printer, or zero output. Valid drive names are A through 0. X, P, and Z specify console, printer, and zero output, respectively.

PATCH

Syntax:

PATCH filename.typ n

The PATCH command displays or installs patch number n to the CP/M Plus system or command files. The patch number n must be between 1 and 32 inclusive.

PIP

Syntax:

PIP Destination = Source
PIP d:[Gn]=filespec [options]
PIP filespec[Gn]=filespec [options]
PIP filespec[Gn]device=filespec [options] device

The file copy program PIP copies files, combines files, and transfers files between disks, printers, consoles, or other devices attached to your computer. The first filespec is the destination. The second filespec is the source. Use two or more source filespecs separated by commas to combine two or more files into one file. [options] is any combination of the available options. The [Gn] option in the destination filespec tells PIP to copy your file to that user number. PIP with no command tail displays an * prompt and awaits your series of commands, entered and processed one line at a time. The source or destination can optionally be any CP/M Plus logical device.

PUT

Syntax:

PUT CONSOLE {OUTPUT TO} FILE filespec {option}
PUT PRINTER {OUTPUT TO} FILE filespec {option}
PUT CONSOLE {OUTPUT TO} CONSOLE

PUT PRINTER {OUTPUT TO} PRINTER

PUT puts console or printer output to a file for the next command entered at the console, until the program terminates. Then console output reverts to the console. Printer output is directed to a file until the program terminates. Then printer output is put back to the printer.

PUT with the SYSTEM option directs all subsequent console/printer output to the specified file. This option terminates when you enter the PUT CONSOLE or PUT PRINTER command.

RENAME/REN (built-in)

Syntax:
RENAME
RENAME new-filespec=old-filespec

RENAME lets you change the name of a file in the directory of a disk. To change several filenames in one command use the * or ? wildcards in the file specifications. You can abbreviate the RENAME command to REN. REN prompts you for input.

RMAC (Not included)

Syntax:
RMAC filespec options

RMAC, a relocatable macro assembler, assembles ASM files into REL files that you can link to create COM files.

RMAC options specify the destination of the output files. Replace d with the destination drive letter for the output files.

SAVE

Syntax:
SAVE

SAVE copies the contents of memory to a file. To use SAVE, first issue the SAVE command, then run your program which reads a file into memory. Your program exits to the SAVE utility which prompts you for a filespec to which it copies the contents of memory, and the beginning and ending address of the memory to be SAVED.

SET

Syntax:

SET [options]
SET d: [options]
SET filespec [options]
SET [option = modifier]
SET filespec [option = modifier]

SET initiates password protection and time stamping of files. It also sets the file and drive attributes Read/Write, Read/Only, DIR and SYS. It lets you label a disk and passord protect the label. To enable time stamping of files, you must first run INITDIR to format the disk directory.

SET Default password operation:

Syntax:
SET [DEFAULT=password]

Instructs the system to use a default password if you do not enter a password for a password-protected file.

SET Time-stamp operations:

Syntax:
SET d: [CREATE=ON|OFF]
SET d: [ACCESS=ON|OFF]
SET d: [UPDATE=ON|OFF]

The above set commands allw YOU to keep a record of the time and date of file creation and update or of the last access update of your files.

SET Drive operations:

Syntax:
SET d: [RO]
SET d: [RW]

Adds or removes write protection from a drive.

SETDEF

Syntax:
SETDEF
SETDEF [TEMPORARY=d:]
SETDEF d:i,d:i,d:i,d:i
SETDEF [ORDER= (typ1, typn)]
SETDEF [DISPLAY | NO DISPLAY)
SETDEF [PAGE | NOPAGE]

SETDEF allows the user to display or define up to four drives for the program search order, the drive for temporary files, and the filetype search order. The SETDEF definitions affect only the loading of programs and/or execution of SUBMIT (SUB) files. SETDEF turns on/off the system Display and Console Page modes. When on, the system displays the location and name of programs loaded or SUBmit files executed, and stops after displaying one full console screen of information.

SHOW

Syntax:

SHOW
SHOW d:
SHOW d: [SPACE]
SHOW d: [LABEL]
SHOW d: [USERS]
SHOW d: [DIR]
SHOW d: [DRIVE]

The SHOW command displays the following disk drive information:

- access mode and the amount of free disk space
- disk label
- current user number
- number of files for each user number on the disk
- number of free directory entries for the disk
- drive characteristics

SID

Syntax:

SID [pgm-filespec],{sym-filespec}

The SID symbolic debugger allows you to monitor and test programs developed for the 8080 microprocessor. SID supports real-time breakpoints, fully monitored execution, symbolic disassembly, assembly, and memory display and fill functions. SID can dynamically load SID utility programs to provide traceback and histogram facilities.

SUBMIT

Syntax:

SUBMIT
SUBMIT filespec
SUBMIT filespec argument ... argument

The SUBMIT command lets you execute a group (batch) of commands from a SUBmit file (a file with filetype of SUB).

SUB files:

The SUB file can contain the following types of lines:

- any valid CP/M Plus command
- any valid CP/M Plus command with SUBMIT parameters ($0-$9)
- any data input line
- any program input line with parameters ($0 to $9)

The command line cannot exceed 135 characters.

TERMINFO (ZX Spectrum Next)

Syntax:
    TERMINFO

This program provides information on the terminal facilities provided by the BIOS on the ZX Spectrum Next.

TERMSIZE (ZX Spectrum Next)

Syntax:
    TERMSIZE top left height width

Terminal resize utility

Size can be up to 32x80 (defaults to 24x80, suitable for many programs). If setting a reduced size, the top and left parameters can be used to make the image more centered on your screen.

TYPE/TYP (built-in)

Syntax:
    TYPE
    TYPE filespec
    TYPE filespec [PAGE]
    TYPE filespec [NOPAGE]

The TYPE command displays the contents of an ASCII character file on your screen.

UPGRADE (ZX Spectrum Next)

Syntax:
    UPGRADE

UPGRADE CP/M from C:/NEXTZXOS/CPMBASE.P3D

USER/USE (built-in)

Syntax:
    USER
    USER n

The USER command sets the current user number. The disk directory can be divided into distinct groups according to a User Number. User numbers range from 0 through 15.

XREF (Not included)

Syntax:
    XREF {d:} filename {$P}

XREF provides a cross-reference summary of variable usage in a program. XREF requires the PRN and SYM files produced by MAC or RMAC for input to the program. The SYM and PRN files must have the same filename as the filename in the XREF command tail. XREF outputs a file of type XRF.

9.1.2 BDOS

From the CP/M 3 Programmers' Guide 1984

This section documents all BDOS system calls to include the parameters that must be passes to them and the values that are returned to the calling program.

BDOS function 0: **SYSTEM RESET**

Entry Parameters:

    C: $00

The System Reset function terminates the calling program and returns control to the CCP via a warm start sequence. Calling this function has the same effect as a jump to location $0000 of Page Zero.

Note that the disk subsystem is not reset by System Reset under CP/M 3. The calling program can pass a return code to the CCP by calling Function 108, Get/Set Program Return Code, prior to making a System Reset call or jumping to location $0000.

BDOS function 1: **CONSOLE INPUT**

Entry Parameters:

C: $01

Returned Value:

A: ASCII Character

The Console Input function reads the next character from the logical console, CONIN:, to register A. Graphic characters, along with carriage return, line-feed, and backspace, CTRL-H, are echoed to the console. Tab characters, CTR-L-1, are expanded in columns of 8 characters. CTRL-S, CTRL-Q, and CTRL-P are normally intercepted as described below. All other non-graphic characters are returned in register A but are not echoed to the console.

When the Console Mode is in the default state Function 1 intercepts the stop scroll, CTRL-S, start scroll, CTRL-Q, and start/stop printer echo, CTRL-P, characters. Any characters that are typed following a CTRL-S and preceding a CTRL-Q are also intercepted. However, if start/stop scroll has been disabled by the Console Mode, the CTRL-S, CTRL-Q, and CTRL-P characters are not intercepted. Instead, they are returned in register A, but are not echoed to the console.

If printer echo has been invoked, all characters that are echoed to the console are also sent to the list device, LST:. Function 1 does not return control to the calling program until a non-intercepted character is typed, thus suspending execution if a character is not ready.

BDOS function 2: **CONSOLE OUTPUT**

Entry Parameters:

C: $02
E: ASCII Character

The Console Output function sends the ASCII character from register E to the logical console device, CONOUT:. When the Console Mode is in the default state (see Section 2.2.1), Function 2 expands tab characters, CTRL-1, in columns of 8 characters, checks for stop scroll, CTRL-S, start scroll, CTRL-Q, and echoes characters to the logical list device, LST:, if printer echo, CTRL-P, has been invoked.

BDOS function 3: **AUXILIARY INPUT**

Entry Parameters:

C: $03

Returned Value:

A: ASCII Character

The Auxiliary Input function reads the next character from the logical auxiliary input device, AUXIN:, into register A. Control does not return to the calling program 'I the character is read. unti

BDOS function 4: **AUXILIARY OUTPUT**

Entry Parameters:

C: $04
E: ASCII Character

The Auxiliary Output function sends the ASCII character from register E to the logical auxiliary output device, AUXOUT:.

BDOS function 5: **LIST OUTPUT**

Entry Parameters:

C: $05
E: ASCII Character

The List Output function sends the ASCII character in register E to the logical list device, LST:.

BDOS function 6: **DIRECT CONSOLE I/O**

Entry Parameters:

C: $06
E: function/data (see description)

Returned Value:

A: char/status/no value (see description)

CP/M 3 supports direct I/O to the logical console, CONIN:, for those specialized applications where unadorned console input and output is required. Use Direct Console I/O carefully because it bypasses all the normal control character functions. Programs that perform direct I/O through the BIOS under previous releases of CP/M should be changed to use direct I/O so that they can be fully supported under future releases of MP/M and CP/M.

A program calls Function 6 by passing one of four different values in register E.

- $FF Console input/status command returns an input character; if no character is ready, a value of zero is returned.
- $FE Console status command (On return, register A contains 00 if no character is ready; otherwise it contains $FF.)
- $FD Console input command, returns an input character; this function will suspend the calling process until a character is ready.
- ASCII Function 6 assumes that register E contains a valid ASCII chararacter and sends it to the console.

BDOS function 7: **AUXILIARY INPUT STATUS**

Entry Parameters:

C: $07

Returned Value:

A: Auxiliary Input Status

The Auxiliary Input Status function returns the value $FF in register A if a character is ready for input from the logical auxiliary input device, AUXIN:. If no character is ready for input, the value $00 is returned.

BDOS function 8: **AUXILIARY OUTPUT STATUS**

Entry Parameters:

C: $08

Returned Value:

A: Auxiliary Output Status

The Auxiliary Output Status function returns the value $FF in register A if the logical auxiliary output device, AUXOUT:, is ready to accept a character for output. If the device is not ready for output, the value $00 is returned.

BDOS function 9: **PRINT STRING**

Entry Parameters:

C: $09
DE: String Address

The Print String function sends the character string addressed by register pair DE to the logical console, CONOUT:, until it encounters a delimiter in the string. Usually the delimiter is a dollar sign, $, but it can be changed to any other value by Function 110, Get/Set Output Delimiter. If the Console

Mode is in the default state, Function 9 expands tab characters, CTRL-I, in columns of 8 characters. It also checks for stop scroll, CTRL-S, start scroll, CTRL-Q, and echoes to the logical list device, LST:, if printer echo, CTRL-P, has been invoked.

BDOS function 10: **READ CONSOLE BUFFER**

Entry Parameters:

C: $0A
DE: Buffer Address

Returned Value:

Console Characters in Buffer

The Read Console Buffer function reads a line of edited console input from the logical console, CONIN:, to a buffer that register pair DE addresses. It terminates input and returns to the calling program when it encounters a return, CTRL-M, or a line feed, CTRL-J, character. Function 10 also discards all input characters after the input buffer is filled. In addition, it outputs a bell character, CTRL-G, to the console when it discards a character to signal the user that the buffer is full. The input buffer addressed by DE has the following format:

where mx is the maximum number of characters which the buffer holds, and nc is the number of characters placed in the buffer. The characters entered by the operator follow the nc value. The value mx must be set prior to making a Function 10 call and may range in value from 1 to 255. Setting mx to zero is equivalent to setting mx to one. The value nc is returned to the calling program and may range from zero to mx. If nc ¡ mx, then uninitialized positions follow the last character, denoted by ?? in the figure. Note that a terminating return or line feed character is not placed in the buffer and not included in the count nc.

If register pair DE is set to zero, Function 10 assumes that an initialized input buffer is located at the current DMA address (see Function 26, Set DMA Address). This allows a program to put a string on the screen for the user to edit. To initialize the input buffer, set characters c1 through cn to the initial value followed by a binary zero terminator.

When a program calls Function 10 with an initialized buffer, Function 10 operates as if the user had typed in the string. When Function 10 encounters the binary zero terminator, it accepts input from the console. At this point, the user can edit the initialized string or accept it as it is by pressing the

RETURN key. However, if the initialized string contains a return, CTRL-M, or a linefeed, CTRL-J, character, Function 10 returns to the calling program without giving the user the opportunity to edit the string.

The level of console editing supported by Function 10 differs for the banked and nonbanked versions of CP/M 3. Refer to the CPIM Plus (CPIM Version 3) Operating System User's Guide for a detailed description of console editing. In the nonbanked version, Function 10 recognizes the following edit control characters.

Nonbanked CP/M 3

- rub/del Removes and echoes the last character; GENCPM can change this function to CTRL-H
- CTRL-C Reboots when at the beginning of line; the Console Mode can disable this function
- CTRL-E Causes physical end of line
- CTRL-H Backspaces one character position; GENCPM can change this function to rub/del
- CTRL-J (Line-feed) terminates input line
- CTRL-M (Return) terminates input line
- CTRL-P Echoes console output to the list device
- CTRL-R Retypes the current line after new line
- CTRL-U Removes current line after new line
- CTRL-X Backspaces to beginning of current line

The banked version of CP/M 3 expands upon the editing provided in the non-banked version. The functionality of the two versions is similar when the cursor is positioned at the end of the line. However, in the banked version, the user can move the cursor anywhere in the current line, insert characters, delete characters, and perform other editing functions. In addition, the banked version saves the previous command line; it can be recalled when the current line is empty. In the banked version, Function 10 recognizes the following edit control characters.

Banked CP/M 3

- rub/del Removes and echoes the last character if at the end of the line; otherwise deletes the character to the left of the current cursor position; GENCPM can change this function to CTRL-H.
- CTRL-A Moves cursor one character to the left.
- CTRL-B Moves cursor to the beginning of the line when not at the beginning; otherwise moves cursor to the end of the line.

- CTRL-C Reboots when at the beginning of line; the Console Mode can disable this function.
- CTRL-E Causes physical end-of-line; if the cursor is positioned in the middle of a line, the characters at and to the right of the cursor are displayed on the next line.
- CTRL-F Moves cursor one character to the right.
- CTRL-G Deletes the character at the current cursor position when in the middle of the line; has no effect when the cursor is at the end of the line.
- CTRL-H Backspaces one character position when positioned at the end of the line,; otherwise deletes the character to the left of the cursor; GENCPM can change this function to rub/del.
- CTRL-J (Line-feed) terminates input; the cursor can be positioned anywhere in the line; the entire input line is accepted; sets the previous line buffer to the input line.
- CTRL-K Deletes all characters to the right of the cursor along with the character at the cursor.
- CTRL-M (Return) terminates input; the cursor can be positioned anywhere in the line; the entire input line is accepted; sets the previous line buffer to the input line.
- CTRL-P Echoes console output to the list device.
- CTRL-R Retypes the characters to the left of the cursor on the new line.
- CTRL-U Updates the previous line buffer to contain the characters to the left of the cursor; deletes current line, and advances to new line.
- CTRL-W Recalls previous line if current line is empty; otherwise moves cursor to end-of-line.
- CTRL-X Deletes all characters to the left of the cursor.

For banked systems, Function 10 uses the console width field defined in the System Control Block. If the console width is exceeded when the cursor is positioned at the end of the line, Function 10 automatically advances to the next line. The beginning of the line can be edited by entering a CTRL-R.

When a character is typed while the cursor is positioned in the middle of the line, the typed character is inserted into the line. Characters at and to the right of the cursor are shifted to the right. If the console width is exceeded, the characters disappear off the right of the screen. However, these characters are not lost. They reappear if characters are deleted out of the line, or if a CTRL-E is typed.

BDOS function 11: **GET CONSOLE STATUS**

Entry Parameters:

C: $0B

Returned Value:

A: Console Status

The Get Console Status function checks to see if a character has been typed at the logical console, CONIN:. If the Console Mode is in the default state, Function 11 returns the value $01 in register A when a character is ready. If a character is not ready, it returns a value of $00.

If the Console Mode is in CTRL-C Only Status mode, Function 11 returns the value $01 in register A only if a CTRL-C has been typed at the console.

BDOS function 12: **RETURN VERSION NUMBER**

Entry Parameters:

C: $0C

Returned Value:

HL: Version Number

The Return Version Number function provides information that allows version independent programming. It returns a two-byte value in register pair HL: H contains $00 for CP/M and L contains $31, the BDOS file system version number. Function 12 is useful for writing applications programs that must run on multiple versions of CP/M and MP/M.

BDOS function 13: **RESET DISK SYSTEM**

Entry Parameters:

C: $0D

The Reset Disk System function restores the file system to a reset state where all the disk drives are set to read-write (see Functions 28 and 29), the default disk is set to drive A, and the default DMA address is reset to $0080. This function can be used, for example, by an application program that requires disk changes during operation. Function 37, Reset Drive, can also be used for this purpose.

BDOS function 14: **SELECT DISK**

Entry Parameters:

C: $0E
E: Selected Disk

Returned Value:

A: Error Flag
H: Physical Error

The Select Disk function designates the disk drive named in register E as the default disk for subsequent BDOS file operations. Register E is set to 0 for drive A, 1 for drive B, and so on through 15 for drive P in a full 16-drive system. In addition, Function 14 logs in the designated drive if it is currently in the reset state. Logging-in a drive activates the drive's directory until the next disk system reset or drive reset operation.

FCBs that specify drive code zero (dr = $00) automatically reference the currently selected default drive. FCBs with drive code values between 1 and 16, however, gnore the selected default drive and directly reference drives A through P.

Upon return, register A contains a zero if the select operation was successful. If a physical error was encountered, the select function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console, and the calling program is terminated. Otherwise, the select function returns to the calling program with register A set to $FF and register H set to one of the following

physical error codes:

- 01 Disk I/O Error
- 04 Invalid drive

BDOS function 15: **OPEN FILE**

Entry Parameters:

C: $0F
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical or Extended Error

The Open File function activates the FCB for a file that exists in the disk directory under the currently active user number or user zero. The calling

program passes the address of the FCB in register pair DE, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and filetype, and byte 12 specifying the extent. Usually, byte 12 of the FCB is initialized to zero.

If the file is password protected in Read mode, the correct password must be placed in the first eight bytes of the current DMA, or have been previously established as the default password (see Function 106). If the current record field of the FCB, cr, is set to $FF, Function 15 returns the byte count of the last record of the file in the cr field. You can set the last record byte count for a file with Function 30, Set File Attributes. Note that the current record field of the FCB, cr, must be zeroed by the calling program before beginning read or write operations if the file is to be accessed sequentially from the first record.

If the current user is non-zero, and the file to be opened does not exist under the current user number, the open function searches user zero for the file. If the file exists under user zero, and has the system attribute, t2', set, the file is opened under user zero. Write operations are not supported for a file that is opened under user zero in this manner.

If the open operation is successful, the user's FCB is activated for read and write operations. The relevant directory information is copied from the matching directory FCB into bytes d0 through dn of the FCB. If the file is opened under user zero when the current user number is not zero, interface attribute f8' is set to one in the user's FCB. In addition, if the referenced file is password protected in Write mode, and the correct password was not passed in the DMA, or did not match the default password, interface attribute f7' is set to one. Write operations are not supported for an activated FCB if interface attribute f7' or f8' is true.

When the open operation is successful, the open function also makes an Access date and time stamp for the opened file when the following conditions are satisfied: the referenced drive has a directory label that requests Access date and time stamping, and the FCB extent number field is zero.

Upon return, the Open File function returns a directory code in register A with the value $00 if the open was successful, or $FF, 255 decimal, if the file was not found. Register H is set to zero in both of these cases. If a physical or extended error was encountered, the Open File function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode, a message identifying the error

is displayed at the console and the program is terminated. Otherwise, the Open File function returns to the calling program with register A set to $FF, and register H set to one of the following physical or extended error codes:

- 01 : Disk I/O Error
- 04 : Invalid drive error
- 07 : File password error
- 09 : ? in the FCB filename or filetype field

BDOS function 16: **CLOSE FILE**

Entry Parameters:

C: $10
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical or Extended Error

The Close File function performs the inverse of the Open File function. The calling program passes the address of an FCB in register pair DE. The referenced FCB must have been previously activated by a successful Open or Make function call (see Functions 15 and 22). Interface attribute f5' specifies how the file is to be closed as shown below:

- f5' = 0 - Permanent close (default mode)
- f5' = 1 - Partial close

A permanent close operation indicates that the program has completed file operations on the file. A partial close operation updates the directory, but indicates that the file is to be maintained in the open state.

If the referenced FCB contains new information because of write operations to the FCB, the close function permanently records the new information in the referenced disk directory. Note that the FCB does not contain new information, and the directory update step is bypassed if only read or update operations have been made to the referenced FCB.

Upon return, the close function returns a directory code in register A with the value $00 if the close was successful,, or $FF, 255 Decimal, if the file was not found. Register H is set to zero in both of these cases. If a physical or extended error is encountered, the close function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error

mode is in the default mode, a message identifying the error is displayed at the console, and the calling program is terminated. Otherwise, the close function returns to the calling program with register A set to $FF and register H set to one of the following physical error codes:

- 01 Disk I/O error
- 02 Read/only disk
- 04 Invalid drive error

BDOS function 17: **SEARCH FOR FIRST**

Entry Parameters:

C: $11
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical Error

The Search For First function scans the directory for a match with the FCB addressed by register pair DE. Two types of searches can be performed. For standard searches, the calling program initializes bytes 0 through 12 of the referenced FCB, with byte 0 specifying the drive directory to be searched, bytes 1 through 11 specifying the file or files to be searched for,, and byte 12 specifying the extent. Usually byte 12 is set to zero. An ASCII question mark, 63 decimal, 3F hex, in any of the bytes 1 through 12 matches all entries on the directory in the corresponding position. This facility, called ambiguous reference, can be used to search for multiple files on the directory. When called in the standard mode, the Search function scans for the first file entry in the specified directory that matches the FCB, and belongs to the current user number.

The Search For First function also initializes the Search For Next function. After the Search function has located the first directory entry matching the referenced FCB, the Search For Next function can be called repeatedly to locate all remaining matching entries. In terms of execution sequence, however, the Search For Next call must either follow a Search For First or Search For Next call with no other intervening BDOS disk-related function calls.

If byte 0 of the referenced FCB is set to a question mark, the Search function ignores the remainder of the referenced FCB, and locates the first directory

entry residing on the current default drive. All remaining directory entries can be located by making multiple Search For Next calls. This type of search operation is not usually made by application programs, but it does provide complete flexibility to scan all current directory values. Note that this type of search operation must be performed to access a drive's directory label.

Upon return, the Search function returns a Directory Code in register A with the value 0 to 3 if the search is successful, or $FF, 255 Decimal, if a matching directory entry is not found. Register H is set to zero in both of these cases. For successful searches, the current DMA is also filled with the directory record containing the matching entry, and the relative starting position is A * 32 (that is, rotate the A register left 5 bits, or ADD A five times). Although it is not usually required for application programs, the directory information can be extracted from the buffer at this position.

If the directory has been initialized for date and time stamping by INITDIR, then an SFCB resides in every fourth directory entry, and successful Directory Codes are restricted to the values 0 to 2. For successful searches, if the matching directory record is an extent zero entry, and if an SFCB resides at offset 96 within the current DMA, contents of (DMA Address + 96) = $21, the SFCB contains the date and time stamp information, and password mode for the file. This information is located at the relative starting position of 97 + (A * 10) within the current DMA in the following format:

- 0 - 3 Create or Access Date and Time Stamp Field
- 4 - 7 Update Date and Time Stamp Field
- 8 : Password Mode Field

If a physical error is encountered, the Search function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console, and the calling program is terminated. Otherwise, the Search function returns to the calling program with register A set to $FF, and register H set to one of the following physical error codes:

- 01 Disk I/O error
- 04 Invalid drive error

BDOS function 18: **SEARCH FOR NEXT**

Entry Parameters:

C: $12

Returned Value:

A: Directory Code
H: Physical Error

The Search For Next function is identical to the Search For First function, except that the directory scan continues from the last entry that was matched. Function 18 returns a Directory code in register A, analogous to Function 17.

Note: in execution sequence, a Function 18 call must follow either a Function 17 or another Function 18 call with no other intervening BDOS disk-related function calls.

BDOS function 19: **DELETE FILE**

Entry Parameters:

C: $13
DE: FCB Address

Returned Value:

A: Directory Code
H: Extended or Physical Error

The Delete File function removes files or XFCBs that match the FCB addressed in register pair DE. The filename and filetype can contain ambiguous references, that is, question marks in bytes f1' through t3', but the dr byte cannot be ambiguous, as it can in the Search and Search Next functions. Interface attribute f5' specifies the type of delete operation that is performed.

- f5' = 0 - Standard Delete (default mode)
- f5' = 1 - Delete only XFCBs

If any of the files that the referenced FCB specify are password protected, the correct password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see Function 106).

For standard delete operations, the Delete function removes all directory entries belonging to files that match the referenced FCB. All disk directory and data space owned by the deleted files is returned to free space, and becomes available for allocation to other files. Directory XFCBs that were owned by the deleted files are also removed from the directory. If interface attribute f5' of the FCB is set to 1, Function 19 deletes only the directory XFCBs that match the referenced FCB.

Note: if any of the files that match the input FCB specification fall the password check, or are Read-Only, then the Delete function does not delete any files or XFCBS. This applies to both types of delete operations.

In nonbanked systems, file passwords and XFCBs are not supported. Thus, if the Delete function is called with interface attribute f5' set to true, the Delete function performs no action but returns with register A set to zero.

Upon return, the Delete function returns a Directory Code in register A with the value 0 if the delete is successful, or $FF, 255 Decimal, if no file that matches the referenced FCB is found. Register H is set to zero in both of these cases. If a physical, or extended error is encountered, the Delete function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the calling program is terminated. Otherwise, the Delete function returns to the calling program with register A set to $FF and register H set to one of the following physical or extended error codes:

- 01 : Disk I/O error
- 02 : Read-Only disk
- 03 : Read-Only file
- 04 : Invalid drive error
- 07 : File password error

BDOS function 20: **READ SEQUENTIAL**

Entry Parameters:

C: $14
DE: FCB Address

Returned Value:

A: Error Code
H: Physical Error

The Read Sequential function reads the next 1 to 128 128-byte records from a file into memory beginning at the current DMA address. The BDOS Multi-Sector Count (see Function 44) determines the number of records to be read. The default is one record. The FCB addressed by register pair DE must have been previously activated by an Open or Make function call.

Function 20 reads each record from byte cr of the extent, then automatically increments the cr field to the next record position. If the cr field overflows,

then the function automatically opens the next logical extent and resets the cr field to 0 in preparation for the next read operation. The calling program must set the cr field to 0 following the Open call if the intent is to read sequentially from the beginning of the file. Upon return, the Read Sequential function sets register A to zero if the read operation is successful. Otherwise, register A contains an error code identifying the error as shown below:

- 01 Reading unwritten data (end-of-file)
- 09 Invalid FCB
- 10 Media change occurred
- 255 Physical Error; refer to register H

Error Code 01 is returned if no data exists at the next record position of the file. Usually, the no data situation is encountered at the end of a file. However, it can also occur if an attempt is made to read a data block that has not been previously written, or an extent which has not been created. These situations are usually restricted to files created or appended with the BDOS random write functions (see Functions 34 and 40).

Error Code 09 is returned if the FCB is invalidated by a previous BDOS close call that returns an error.

Error Code 10 is returned if a media change occurs on the drive after the referenced FCB is activated by a BDOS Open, or Make Call.

Error Code 255 is returned if a physical error is encountered and the BDOS error mode is Return Error mode, or Return and Display Error mode (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console, and the calling program is terminated. When a physical error is returned to the calling program, register H contains one of the following error codes:

- 01 Disk I/O error
- 04 Invalid drive error

On all error returns except for physical error returns, A = 255, Function 20 sets register H to the number of records successfully read before the error is encountered. This value can range from 0 to 127 depending on the current BDOS Multi-Sector Count. It is always set to zero when the Multi-Sector Count is equal to one.

BDOS function 21: **WRITE SEQUENTIAL**

Entry Parameters:

C: $15
DE: FCB Address

Returned Value:

A: Error Code
H: Physical Error

The Write Sequential function writes 1 to 128 128-byte data records, beginning at the current DMA address into the file named by the FCB addressed in register pair DE. The BDOS Multi-Sector Count (see Function 44) determines the number of 128 byte records that are written. The default is one record. The referenced FCB must have been previously activated by a BDOS Open or Make function call.

Function 21 places the record into the file at the position indicated by the cr byte of the FCB, and then automatically increments the cr byte to the next record position. If the cr field overflows, the function automatically opens, or creates the next logical extent, and resets the cr field to 0 in preparation for the next write operation. If Function 21 is used to write to an existing file, then the newly written records overlay those already existing in the file. The calling program must set the cr field to 0 following an Open or Make call if the intent is to write sequentially from the beginning of the file.

Function 21 makes an Update date and time for the file if the following conditions are satisfied: the referenced drive has a directory label that requests date and time stamping, and the file has not already been stamped for update by a previous Make or Write function call.

Upon return, the Write Sequential function sets register A to zero if the write operation is successful. Otherwise, register A contains an error code identifying the error as shown below:

- 01 No available directory space
- 02 No available data block
- 09 Invalid FCB
- 10 Media change occurred
- 255 Physical Error : refer to register H

Error Code 01 is returned when the write function attempts to create a new extent that requires a new directory entry, and no available directory entries exist on the selected disk drive.

Error Code 02 is returned when the write command attempts to allocate a new data block to the file, and no unallocated data blocks exist on the

selected disk drive.

Error Code 09 is returned if the FCB is invalidated by a previous BDOS close call that returns an error.

Error Code 10 is returned if a media change occurs on the drive after the referenced FCB is activated by a BDOS Open or Make call.

Error Code 255 is returned if a physical error is encountered and the BDOS error mode is Return Error mode, or Return and Display Error mode (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console, and the calling program is terminated. When a physical error is returned to the calling program, register H contains one of the following error codes:

- 01 Disk I/O error
- 02 Read-Only disk
- 03 Read-Only file or File open from user 0 when the current user number is non-zero or File password protected in Write mode
- 04 Invalid drive error

On all error returns, except for physical error returns, A = 255, Function 21 sets register H to the number of records successfully written before the error was encountered. This value can range from 0 to 127 depending on the current BDOS Multi-Sector Count. It is always set to zero when the Multi-Sector Count is set to one.

BDOS function 22: **MAKE FILE**

Entry Parameters:

C: $16
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical or Extended Error

The Make File function creates a new directory entry for a file under the current user number. It also creates an XFCB for the file if the referenced drive has a directory label that enables password protection on the drive, and the calling program assigns a password to the file.

The calling program passes the address of the FCB in register pair DE, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the

filename and filctype, and byte 12 set to the extent number. Usually, byte 12 is set to zero. Byte 32 of the FCB, the cr field, must be initialized to zero, before or after the Make call, if the intent is to write sequentially from the beginning of the file.

Interface attribute f6' specifies whether a password is to be assigned to the created file.

- f6' = 0 - Do not assign password (default)
- f6' = 1 - Assign password to created file

When attribute f6' is set to 1, the calling program must place the password in the first 8 bytes of the current DMA buffer, and set byte 9 of the DMA buffer to the password mode (see Function 102). Note that the Make function only interrogates interface attribute f6' if passwords are activated on the referenced drive. In nonbanked systems, file passwords are not supported, and attribute f6' is never interrogated.

The Make function returns with an error if the referenced FCB names a file that currently exists in the directory under the current user number.

If the Make function is successful, it activates the referenced FCB for file operations by opening the FCB, and initializes both the directory entry and the referenced FCB to an empty file. It also initializes all file attributes to zero. In addition, Function 22 makes a Creation date and time stamp for the file if the following conditions are satisfied: the referenced drive has a directory label that requests Creation date and time stamping and the FCB extent number field is equal to zero. Function 22 also makes an Update stamp if the directory label requests update stamping and the FCB extent field is equal to zero.

If the referenced drive contains a directory label that enables password protection, and if interface attribute f6' has been set to 1, the Make function creates an XFCB for the file. In addition, Function 22 also assigns the password, and password mode placed in the first nine bytes of the DMA, to the XFCB.

Upon return, the Make function returns a directory code in register A with the value 0 if the make operation is successful, or $FF, 255 decimal, if no directory space is available. Register H is set to zero in both of these cases. If a physical or extended error is encountered, the Make function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is

displayed at the console, and the calling program is terminated. Otherwise, the Make function returns to the calling program with register A set to $FF, and register H set to one of the following physical or extended error codes:

- 01 : Disk I/O error
- 02 : Read-Only disk
- 04 : Invalid drive error
- 08 : File already exists
- 09 : ? in filename or filetype field

BDOS function 23: **RENAME FILE**

Entry Parameters:

C: $17
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical or Extended Error

The Rename function uses the FCB, addressed by register pair DE, to change all directory entries of the file specified by the filename in the first 16 bytes of the FCB to the filename in the second 16 bytes. If the file specified by the first filename is password protected, the correct password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see Function 106). The calling program must also ensure that the filenames specified in the FCB are valid and unambiguous, and that the new filename does not already exist on the drive. Function 23 uses the dr code at byte 0 of the FCB to select the drive. The drive code at byte 16 of the FCB is ignored.

Upon return, the Rename function returns a Directory Code in register A with the value 0 if the rename is successful, or $0FF, 255 Decimal, if the file named by the first filename in the FCB is not found. Register H is set to zero in both of these cases. If a physical or extended error is encountered, the Rename function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the program is terminated. Otherwise, the Rename function returns to the calling program with register A set to $0FF and register H set to one of the following physical or extended error codes:

- 01 Disk I/O error

- 02 Read-Only disk
- 03 Read-Only file
- 04 Invalid drive error
- 07 File password error
- 08 File already exists
- 09 ? in filename or filetype field

BDOS function 24: **RETURN LOGIN VECTOR**

Entry Parameters:

C: $18

Returned Value:

HL: Login Vector

Function 24 returns the login vector in register pair HL. The login vector is a 16-bit value with the least significant bit of L corresponding to drive A, and the highorder bit of H corresponding to the 16th drive, labelled P. A 0 bit indicates that the drive is not on-line, while a 1 bit indicates the drive is active. A drive is made active by either an explicit BDOS Select Disk call, number 14, or an implicit selection when a BDOS file operation specifies a non-zero dr byte in the FCB. Function 24 maintains compatibilty with earlier releases since registers A and L contain the same values upon return.

BDOS function 25: **RETURN CURRENT DISK**

Entry Parameters:

C: $19

Returned Value:

A: Current Disk

Function 25 returns the currently selected default disk number in register A. The disk numbers range from 0 through 15 corresponding to drives A through P.

BDOS function 26: **SET DMA ADDRESS**

Entry Parameters:

C: $1A
DE: DMA Address

DMA is an acronym for Direct Memory Address, which is often used in connection with disk controllers that directly access the memory of the computer to transfer data to and from the disk subsystem. Under CP/M 3, the current DMA is usually defined as the buffer in memory where a record resides before a disk write, and after a disk read operation. If the BDOS Multi-Sector Count is equal to one (see Function 44), the size of the buffer is 128 bytes. However, if the BDOS Multi-Sector Count is greater than one, the size of the buffer must equal N * 128, where N equals the Multi-Sector Count.

Some BDOS functions also use the current DMA to pass parameters, and to return values. For example, BDOS functions that check and assign file passwords require that the password be placed in the current DMA. As another example, Function 46, Get Disk Free Space, returns its results in the first 3 bytes of the current DMA. When the current DMA is used in this context, the size of the buffer in memory is determined by the specific requirements of the called function.

When a transient program is initiated by the CCP, its DMA address is set to $0080. The BDOS Reset Disk System function, Function 13, also sets the DMA address to $0080. The Set DMA function can change this default value to another memory address. The DMA address is set to the value passed in the register pair DE. The DMA address remains at this value until it is changed by another Set DMA Address, or Reset Disk System call.

BDOS function 27: **GET ADDR(ALLOC)**

Entry Parameters:

C: $1B

Returned Value:

HL: ALLOC Address

CP/M 3 maintains an allocation vector in main memory for each active disk drive. Some programs use the information provided by the allocation vector to determine the amount of free data space on a drive. Note, however, that the allocation information might be inaccurate if the drive has been marked Read-Only.

Function 27 returns in register pair HL, the base address of the allocation vector for the currently selected drive. If a physical error is encountered when the BDOS error mode is one of the return modes (see Function 45), Function 27 returns the value $FFFF in the register pair HL.

In banked CP/M 3 systems, the allocation vector can be placed in bank zero. In this case, a transient program cannot access the allocation vector. However, the BDOS function, Get Disk Free Space (Function 46), can be used to directly return the number of free 128-byte records on a drive. The CP/M 3 utilities that display a drive's free space, DIR and SHOW, use Function 46 for that purpose.

BDOS function 28: **WRITE PROTECT DISK**

Entry Parameters:

C: $1C

The Write Protect Disk function provides temporary write protection for the currently selected disk by marking the drive as Read-Only, No program can write to a disk that is in the Read-Only state. A drive reset operation must be performed for a Read-Only drive to restore it to the Read-Write state (see Functions 13 and 37).

BDOS function 29: **GET READ-ONLY VECTOR**

Entry Parameters:

C: l$D

Returned Value:

HL: R/O Vector Value

Function 29 returns a bit vector in register pair HL that indicates which drives have the temporary Read-Only bit set. The Read-Only bit can be set only by a BDOS Write Protect Disk call.

The format of the bit vector is analogous to that of the login vector returned by Function 24. The least significant bit corresponds to drive A, while the most significant bit corresponds to drive P.

BDOS function 30: **SET FILE ATTRIBUTES**

Entry Parameters:

C: $1E
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical or Extended error

By calling the Set File Attributes function, a program can modify a file's attributes and set its last record byte count. Other BDOS functions can be called to interrogate these file parameters, but only Function 30 can change them. The file attributes that can be set or reset by Function 30 are fl' through f4', Read-Only, t1', System, t2', and Archive, t3'. The register pair DE addresses an FCB containing a filename with the appropriate attributes set or reset. The calling program must ensure that it does not specify an ambiguous filename. In addition, if the specified file is password totected, the correct password must be placed in the first eight bytes of the current DMA buffer or have been previously established as the default password (see Function 106).

Interface attribute f6' specifies whether the last record byte count of the specified file is to be set:

- f6' = 0 - Do not set byte count (default mode)
- f6' = 1 - Set byte count

If interface attribute f6' is set, the calling program must set the cr field of the referenced FCB to the byte count value. A program can access a file's byte count value with the BDOS Open, Search, or Search Next functions.

Function 30 searches the referenced directory for entries belonging to the current user number that matches the FCB specified name and type fields. The function then updates the directory to contain the selected indicators, and if interface attribute f6' is set, the specified byte count value. Note that the last record byte count is maintained in byte 13 of a file's directory FCBS.

File attributes t1', t2', and t3' are defined by CP/M 3. (They are described in Section 2.3.4.) Attributes fl' through f4' are not presently used, but can be useful for application programs, because they are not involved in the matching program used by the BDOS during Open File and Close File operations. Indicators f5' through f8' are reserved for use as interface attributes.

Upon return, Function 30 returns a Directory Code in register A with the value 0 if the function is successful, or $FF, 255 Decimal, if the file specified by the referenced FCB is not found. Register H is set to zero in both of these cases. If a physical or extended error is encountered', the Set File Attributes function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console, and the program is terminated. Otherwise, Function 30 returns to the calling program with reg'ls-Ler A set to $FF, and register H set to one of the following physical

or extended error codes:

- 01 Disk I/O error
- 02 Read-Only disk
- 04 Select error
- 07 File password error
- 09 ? in filename or filetype field

BDOS function 31: **GET ADDR(DPB PARMS)**

Entry Parameters:

C: $1F

Returned Value:

HL: DPB Address

Function 31 returns in register pair HL the address of the BIOS-resident Disk Parameter Block, DPB, for the currently selected drive. (Refer to the CP/M Plus (CP/M Version 3) Operating System System Guide for the format of the DPB). The calling program can use this address to extract the disk parameter values.

If a physical error is encountered when the BDOS error mode is one of the return modes (see Function 45), Function 31 returns the value $FFFF in the register pair HL.

BDOS function 32: **SET/GET USER CODE**

Entry Parameters:

C: $20

Returned Value:

E: $FF (get) or User Code (set)
A: Current Code or (no value)

A program can change, or interrogate the currently active user number by calling Function 32. If register E = $FF, then the value of the current user number is returned in register A, where the value is in the range of 0 to 15. If register E is not $FF, then the current user number is changed to the value of E, modulo 16.

BDOS function 33: **READ RANDOM**

Entry Parameters:

C: $21
DE: FCB Address

Returned Value:

A: Error Code
H: Physical Error

The Read Random function is similar to the Read Sequential function except that the read operation takes place at a particular random record number, selected by the 24-bit value constructed from the three byte, r0, r1, r2, field beginning at position 33 of the FCB. Note that the sequence of 24 bits is stored with the least significant byte first, rO, the middle byte next, rl, and the high byte last, r2. The random record number can range from 0 to 262,143. This corresponds to a maximum value of 3 in byte r2.

To read a file with Function 33, the calling program must first open the base extent, extent 0. This ensures that the FCB is properly initialized for subsequent random access operations. The base extent may or may not contain any allocated data. Function 33 reads the record specified by the random record field into the current DMA address. The function automatically sets the logical extent and current record values, but unlike the Read Sequential function, it does not advance the current record number. Thus, a subsequent Read Random call rereads the same record. After a random read operation, a file can be accessed sequentially, starting from the current randomly accessed position. However, the last randomly accessed record is reread or rewritten when switching from random to sequential mode.

If the BDOS Multi-Sector Count is greater than one (see Function 44), the Read Random function reads multiple consecutive records into memory beginning at the current DMA. The rO, rl, and r2 field of the FCB is automatically incremented to read each record. However, the FCBs random record number is restored to the first record's value upon return to the calling program.

Upon return, the Read Random function sets register A to zero if the read operation was successful. Otherwise, register A contains one of the following error codes:

- 01 Reading unwritten data (end-of-file)
- 03 Cannot close current extent
- 04 Seek to unwritten extent
- 06 Random record number out of range

- 10 Media change occurred
- 255 Physical Error : refer to register H

Error Code 01 is returned if no data exists at the next record position of the file. Usually, the no data situation is encountered at the end of a file. However, it can also occur if an attempt is made to read a data block that has not been previously written.

Error Code 03 is returned when the Read Random function cannot close the current extent prior to moving to a new extent.

Error Code 04 is returned when a read random operation accesses an extent that has not been created.

Error Code 06 is returned when byte 35, r2, of the referenced FCB is greater than 3.

Error Code 10 is returned if a media change occurs on the drive after the referenced FCB is activated by a BDOS Open or Make Call.

Error Code 255 is returned if a physical error is encountered, and the BDOS error mode is one of the return modes (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console, and the calling program is terminated. When a physical error is returned to the calling program, register H contains one of the following error codes:

- 01 Disk I/O error
- 04 Invalid drive error

On all error returns except for physical errors, A = 255, the Read Random function sets register H to the number of records successfully read before the error is encountered. This value can range from 0 to 127 depending on the current BDOS Multi-Sector Count. It is always set to zero when the Multi-Sector Count is equal to one.

BDOS function 34: **WRITE RANDOM**

Entry Parameters:

C: $22
DE: FCB Address

Returned Value:

A: Error Code
H: Physical Error

The Write Random function is analogous to the Read Random function, except that data is written to the disk from the current DMA address. If the disk extent or data block where the data is to be written is not already allocated, the BDOS automatically performs the allocation before the write operation continues.

To write to a file using the Write Random function, the calling program must first open the base extent, extent 0. This ensures that the FCB is properly initialized for subsequent random access operations. If the file is empty, the calling program must create the base extent with the Make File function before calling Function 34. The base extent might or might not contain any allocated data, but it does record the file in the directory, so that the file can be displayed by the DIR utility.

The Write Random function sets the logical extent and current record positions to correspond with the random record being written, but does not change the random record number. Thus, sequential read or write operations can follow a random write, with the current record being reread or rewritten as the calling program switches from random to sequential mode.

Function 34 makes an Update date and time stamp for the file if the following conditions are satisfied: the referenced drive has a directory label that requests Update date and time stamping if the file has not already been stamped for update by a previous BDOS Make or Write call.

If the BDOS Multi-Sector Count is greater than one (see Function 44), the Write Random function reads multiple consecutive records into memory beginning at the current DMA. The rO, rl, and r2 field of the FCB is automatically incremented to write each record. However, the FCB's random record number is restored to the first record's value when it returns to the calling program. Upon return, the Write Random function sets register A to zero if the write operation is successful. Otherwise, register A contains one of the following error codes:

- 02 No available data block
- 03 Cannot Close current extent
- 05 No available directory space
- 06 Random record number out of range
- 10 Media change occurred
- 255 Physical Error : refer to register H

Error Code 02 is returned when the write command attempts to allocate a new data block to the file and no unallocated data blocks exist on the

selected disk drive.

Error Code 03 is returned when the Write Random function cannot close the current extent prior to moving to a new extent.

Error Code 05 is returned when the write function attempts to create a new extent that requires a new directory entry and no available directory entries exist on the selected disk drive.

Error Code 06 is returned when byte 35, r2, of the referenced FCB is greater than 3.

Error Code 10 is returned if a media change occurs on the drive after the referenced FCB is activated by a BDOS Open or Make Call.

Error Code 255 is returned if a physical error is encountered and the BDOS error mode is one of the return modes (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console, and the calling program is terminated. When a physical error is returned to the calling program, it is identified by register H as shown below:

- 01 Disk I/O error
- 02 Read-Only disk
- 03 Read-Only file or File open from user 0 when the current user number is nonzero or File password protected in Write mode
- 04 Invalid drive error

On all error returns, except for physical errors, A = 255, the Write Random function sets register H to the number of records successfully written before the error is encountered. This value can range from 0 to 127 depending on the current BDOS Multi-Sector Count. It is always set to zero when the Multi-Sector Count is equal to one.

BDOS function 35: **COMPUTE FILE SIZE**

Entry Parameters:

C: $23
DE: FCB Address

Returned Value:

A: Error Flag
H: Physical or Extended error
Random Record Field Set

The Compute File Size function determines the virtual file size, which is, in effect, the address of the record immediately following the end of the file. The virtual size of a file corresponds to the physical size if the file is written sequentially. If the file is written in random mode, gaps might exist in the allocation, and the file might contain fewer records than the indicated size. For example, if a single record with record number 262,143, the CP/M 3 maximum is written to a file using the Write Random function, then the virtual size of the file is 262,144 records even though only 1 data block 'is actually allocated.

To compute file size, the calling program passes in register pair DE the address of an FCB in random mode format, bytes rO, rl and r2 present. Note that the FCB must contain an unambiguous filename and filetype. Function 35 sets the random record field of the FCB to the random record number + 1 of the last record in the file. If the r2 byte is set to 04, then the file contains the maximum record count 262,144.

A program can append data to the end of an existing file by calling Function 35 to set the random record position to the end of file, and then performing a sequence of random writes starting at the preset record address.

Note: the BDOS does not require that the file be open to use Function 35. However, if the file has been written to, it must be closed before calling Function 35. Otherwise, an incorrect file size might be returned.

Upon return, Function 35 returns a zero in register A if the file specified by the referenced FCB is found, or an $FF in register A if the file is not found. Register H is set to zero in both of these cases. If a physical error is encountered, Function 35 performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the program is terminated. Otherwise, Function 35 returns to the calling program with register A set to $FF, and register H set to one of the following physical errors:

- 01 Disk I/O error
- 04 Invalid drive error

BDOS function 36: **SET RANDOM RECORD**

Entry Parameters:

C: $24
DE: FCB Address

Returned Value:

Random Record Field Set

The Set Random Record function returns the random record number of the next record to be accessed from a file that has been read or written sequentially to a particular point. This value is returned in the random record field, bytes rO, rl, and r2, of the FCB addressed by the register pair DE. Function 36 can be useful in two ways,

First, it is often necessary to initially read and scan a sequential file to extract the positions of various key fields. As each key is encountered, Function 36 is called to compute the random record position for the data corresponding to this key. If the data unit size is 128 bytes, the resulting record number minus one is placed into a table with the key for later retrieval. After scanning the entire file and tabularizing the keys and their record numbers, you can move directly to a particular record by performing a random read using the corresponding random record number that you saved earlier. The scheme is easily generalized when variable record lengths are involved, because the program need only store the buffer-relative byte position along with the key and record number to find the exact starting position of the keyed data at a later time.

A second use of Function 36 occurs when switching from a sequential read or write over to random read or write. A file is sequentially accessed to a particular point in the file, then Function 36 is called to set the record number, and subsequent random read and write operations continue from the next record in the file.

BDOS function 37: **RESET DRIVE**

Entry Parameters:

C: $25
DE: Drive Vector

Returned Value:

A: $00

The Reset Drive function programmatically restores specified drives to the reset state. A reset drive is not logged-in and is in Read-Write status. The passed parameter in register pair DE is a 16-bit vector of drives to be reset, where the least significant bit corresponds to the first drive A, and the high-order bit corresponds to the sixteenth drive, labelled P. Bit values of 1

indicate that the specified drive is to be reset.

BDOS function 38: **ACCESS DRIVE**

Entry Parameters:

C: $26

This is an MP/M function that is not supported under CP/M 3. If called, the file system returns a zero In register A indicating that the access request is successful.

BDOS function 39: **FREE DRIVE**

Entry Parameters:

C: $27

This is an MP/M function that is not supported under CP/M 3. If called, the file system returns a zero In register A indicating that the free request is successful.

BDOS function 40: **WRITE RANDOM WITH ZERO FILL**

Entry Parameters:

C: $28
DE: FCB address

Returned Value:

A: Error Code
H: Physical Error

The Write Random With Zero Fill function is identical to the Write Random function (Function 34) with the exception that a previously unallocated data block is filled with zeros before the record is written. If this function has been used to create a file, records accessed by a read random operation that contain all zeros identify unwritten random record numbers. Unwritten random records in allocated data blocks of files created using the Write Random function (Function 34) contain uninitialized data.

BDOS function 41: **TEST AND WRITE RECORD**

Entry Parameters:

C: $29
DE: FCB Address

Returned Value:

A: Error Code
H: Physical Error

The Test and Write Record function is an MP/M function that is not supported under CP/M 3. If called, Function 41 returns with register A set to $FF and register H set to zero.

BDOS function 42: **LOCK RECORD**

Entry Parameters:

C: $2A
DE: FCB Address

Returned Value:

A: $00

The Lock Record function is an MP/M II function that is supported under CP/M 3 only to provide compatibility between CP/M 3 and MP/M. It is intended for use in situations where more than one running program has Read-Write access to a common file. Because CP/M 3 is a single-user operating system in which only one program can run at a time, this situation cannot occur. Thus, under CP/M 3, Function 42 performs no action except to return the value $00 in register A indicating that the record lock operation is successful.

BDOS function 43: **UNLOCK RECORD**

Entry Parameters:

C: $2B
DE: FCB Address

Returned Value:

A: $00

The Unlock Record function is an MP/M II function that is supported under CP/M 3 only to provide compatibility between CP/M 3 and MP/M. It is intended for use in situations where more than one running program has Read-Write access to a common file. Because CP/M 3 is a single-user operating system in which only one program can run at a time, this situation cannot occur. Thus, under CP/M 3, Function 43 performs no action except to return the value $00 in register A indicating that the record unlock operation is successful.

BDOS function 44: **SET MULTI-SECTOR COUNT**

Entry Parameters:

C: $2C
E: Number of Sectors

Returned Value:

A: Return Code

The Set Multi-Sector Count function provides logical record blocking under CP/M 3. It enables a program to read and write from 1 to 128 records of 128 bytes at a time during subsequent BDOS Read and Write functions.

Function 44 sets the Multi-Sector Count value for the calling program to the value passed in register E. Once set, the specified Multi-Sector Count remains in effect until the calling program makes another Set Multi-Sector Count function call and changes the value. Note that the CCP sets the Multi-Sector Count to one when it initiates a transient program.

The Multi-Sector Count affects BDOS error reporting for the BDOS Read and Write functions. If an error interrupts these functions when the Multi-Sector is greater than one, they return the number of records successfully read or written in register H for all errors except for physical errors (A = 255).

Upon return, register A is set to zero if the specified value is in the range of 1 to 128. Otherwise, register A is set to $FF.

BDOS function 45: **SET BDOS ERROR MODE**

Entry Parameters:

C: $2D
E: BDOS Error Mode

Returned Value:

None

Function 45 sets the BDOS error mode for the calling program to the mode specified in register E. If register E is set to $FF, 255 decimal, the error mode is set to Return Error mode. If register E is set to $FE, 254 decimal, the error mode is set to Return and Display mode. If register E is set to any other value, the error mode is set to the default mode.

The SET BDOS Error Mode function determines how physical and extended

errors (see Section 2.2.13) are handled for a program. The Error Mode can exist in three modes: the default mode, Return Error mode, and Return and Display Error mode. In the default mode, the BDOS displays a system message at the console that identifies the error and terminates the calling program. In the return modes, the BDOS sets register A to $FF, 255 decimal, places an error code that identifies the physical or extended error in register H and returns to the calling program. In Return and Display mode, the BDOS displays the system message before returning to the calling program. No system messages are displayed, however, when the BDOS is in Return Error mode.

BDOS function 46: **GET DISK FREE SPACE**

Entry Parameters:

C: $2E
E: Drive

Returned Value:

First 3 bytes of current DMA buffer
A: Error Flag
H: Physical Error

The Get Disk Free Space function determines the number of free sectors, 128 byte records, on the specified drive. The calling program passes the drive number in register E, with 0 for drive A, 1 for B, and so on, through 15 for drive P in a full 16drive system. Function 46 returns a binary number in the first 3 bytes of the current DMA buffer. This number is returned in the following format:

fso fsl fs2

Disk Free Space Field Format

fso = low byte
fsl = middle byte
fs2 = high byte

Note that the returned free space value might be inaccurate if the drive has been marked Read-Only.

Upon return, register A is set to zero if the function is successful. However, if the BDOS Error Mode is one of the return modes (see Function 45), and a physical error is encountered, register A is set to $FF, 255 decimal, and

register H is set to one of the following values:

- 01 - Disk I/O error
- 04 - Invalid drive error

BDOS function 47: **CHAIN TO PROGRAM**

Entry Parameters:

C: $2F
E: Chain Flag

The Chain To Program function provides a means of chaining from one program to the next without operator intervention. The calling program must place a command line terminated by a null byte, OOH, in the default DMA buffer. If register E is set to $FF, the CCP initializes the default drive and user number to the current program values when it passes control to the specified transient program. Otherwise, these parameters are set to the default CCP values. Note that Function 108, Get/Set Program Return Code, can be used to pass a two byte value to the chained program.

Function 47 does not return any values to the calling program and any encountered errors are handled by the CCP.

BDOS function 48: **FLUSH BUFFERS**

Entry Parameters:

C: $30

Returned Value:

A: Error Flag
H: Physical Error

E: Purge Flag

The Flush Buffers function forces the write of any write-pending records contained in internal blocking/deblocking buffers. If register E is set to $FF, this function also purges all active data buffers. Programs that provide write with read verify support need to purge internal buffers to ensure that verifying reads actually access the disk instead of returning data that is resident in internal data buffers. The CP/M 3 PIP utility is an example of such a program.

Upon return, register A is set to zero if the flush operation is successful. If a physical error is encountered, the Flush Buffers function performs differ-

ent actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode,, a message identifying the error is displayed at the console and the calling program is terminated. Otherwise, the Flush Buffers function returns to the calling program with register A set to $FF and register H set to the following physical error code:

- 0 1 Disk I/O error
- 02 Read/only disk
- 04 Invalid drive error

BDOS function 49: **GET/SET SYSTEM CONTROL BLOCK**

Entry Parameters:

C: $31
DE: SCB PB Address

Returned Value:

A: Returned Byte
HL: Returned Word

Function 49 allows access to parameters located in the CP/M 3 System Control Block (SCB). The SCB is a 100-byte data structure residing within the BDOS that contains flags and data used by the BDOS, CCP and other system components. Note that Function 49 is a CP/M 3 specific function. Programs intended for both MP/M 11 and CP/M 3 should either avoid the use of this function or isolate calls to this function in CP/M 3 version-dependent sections.

To use Function 49, the calling program passes the address of a data structure called the SCB parameter block in register pair DE. This data structure identifies the byte or word of the SCB to be updated or returned. The SCB parameter block is defined as:

```
SCBPB: DB OFFSET ; Offset within SCB
       DB SET ; 0FFH if setting a byte
       ; 0FEH if setting a word
       ; 001H - 0FDH are reserved
       ; 000H if a get operation
       DW VALUE ; Byte or word value to be set
```

The OFFSET parameter identifies the offset of the field within the SCB to be updated or accessed. The SET parameter determines whether Function

49 is to set a byte or word value in the SCB or if it is to return a byte from the SCB. The VALUE parameter is used only in set calls. In addition, only the first byte of VALUE is referenced in set byte calls.

Use caution when you set SCB fields. Some of these parameters reflect the current state of the operating system. If they are set to invalid values, software errors can result. In general, do not use Function 49 to set a system parameter if another BDOS function can achieve the same result. For example, Function 49 can be called to update the Current DMA Address field within the SCB. This is not equivalent to making a Function 26, Set DMA Address call, and updating the SCB Current DMA field in this way would result in system errors. However, you can use Function 49 to return the Current DMA address. The System Control Block is summarized in 9.1.

If Function 49 is called with the OFFSET parameter of the SCB parameter block greater than $63, the function performs no action but returns with registers A and HL set to zero.

BDOS function 50: **DIRECT BIOS CALLS**

Entry Parameters:

C: $32
DE: BIOS PB Address

Returned Value:

BIOS RETURN

Function 50 provides a direct BIOS call through the BDOS to the BIOS. The calling program passes the address of a data structure called the BIOS Parameter Block (BIOSPB) in register pair DE. The BIOSPB contains the BIOS function number and register contents as shown below:

```
BIOSPB: db FUNC ; BIOS function no.
        db AREG ; A register contents
        dw BCREG ; BC register contents
        dw DEREG ; DE register contents
        dw HLREG ; HL register contents
```

System Reset (Function 0) is equivalent to Function 50 with a BIOS function number of 1.

Note that the register pair BIOSPB fields (BCREG, DEREG, HLREG) arc defined in low byte, high byte order. For example, in the BCREG field,

the first byte contains the C register value, the second byte contains the B register value. Under CP/M 3, direct BIOS calls via the BIOS jump vector are only supported for the BIOS Console I/O and List functions. You must use Function 50 to call any other

BIOS functions. In addition, Function 50 intercepts BIOS Function 27 (Select Memory) calls and returns with register A set to zero. Refer to the CPIM Plus (CP/M Version 3) Operating System System Guide for the definition of the BIOS functions and their register passing and return conventions.

BDOS function 59: **LOAD OVERLAY**

Entry Parameters:

C: $3B
DE: FCB Address

Returned Value:

A: Error Code
H: Physical Error

Only transient programs with an RSX header can use the Load Overlay function because BDOS Function 59 is supported by the LOADER module. The calling program must have a header to force the LOADER to remain resident after the program is loaded (see Section 1.3).

Function 59 loads either an absolute or relocatable module. Relocatable modules are identified by a filetype of PRL. Function 59 does not call the loaded module.

The referenced FCB must be successfully opened before Function 59 is called. The load address is specified in the first two random record bytes of the FCB, rO and rl. The LOADER returns an error if the load address is less than $100, or if performing the requested load operation would overlay the LOADER, or any other Resident System Extensions that have been previously loaded.

When loading relocatable files, the LOADER requires enough room at the load address for the complete PRL file including the header and bit map (see Appendix B). Otherwise an error is returned. Function 59 also returns an error on PRL file load requests if the specified load address is not on a page boundary.

Upon return, Function 59 sets register A to zero if the load operation is successful. If the LOADER RSX is not resident in memory because the calling

program did not have a RSX header, the BDOS returns with register A set to $FF and register H set to zero. If the LOADER detects an invalid load address, or if insufficient memory is available to load the overlay, Function 59 returns with register A set to $FE. All other error returns are consistent with the error codes returned by BDOS Function 20, Read Sequential.

BDOS function 60: **CALL RESIDENT SYSTEM EXTENSION**

Entry Parameters:

C: $3C
DE: RSX PB Address

Returned Value:

A: Error Code
H: Physical Error

Function 60 is a special BDOS function that you use when you call Resident System Extensions. The RSX subfunction is specified in a structure called the RSX Parameter Block, defined as follows:

```
RSXPB: db FUNC ; RSX Function number
       db NUMPARMS ; Number of word Parameters
       dw PARMETER1 ; Parameter I
       dw PARMETER2 ; Parameter 2
       . . .
       dw PARMETERN ; Parameter n
```

RSX modules filter all BDOS calls and capture RSX function calls that they can handle. If there is no RSX module present in memory that can handle a specific RSX function call, the call is not trapped, and the BDOS returns $FF in registers A and L. RSX function numbers from 0 to 127 are available for CP/M 3 compatible software use. RSX function numbers 128 to 255 are reserved for system use.

BDOS function 98: **FREE BLOCKS**

Entry Parameters:

C: $62

Returned Value:

A: Error Flag
H: Physical Error

The Free Blocks function scans all the currently logged-in drives, and for each drive returns to free space all temporarily-allocated data blocks. A temporarily-allocated data block is a block that has been allocated to a file by a BDOS write operation but has not been permanently recorded in the directory by a BDOS close operation. The CCP calls Function 98 when it receives control following a system warm start. Be sure to close your file, particularly any file you have written to, prior to calling Function 98.

In the nonbanked version of CP/M 3, Function 98 frees only temporarily allocated blocks for systems that request double allocation vectors in GENCPM.

Upon return, register A is set to zero if Function 98 is successful. If a physical error is encountered, the Free Blocks function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the calling program is terminated. Otherwise, the Free Blocks function returns to the calling program with register A set to $FF and register H set to the following physical error code:

- 04 : Invalid drive error

BDOS function 99: **TRUNCATE FILE**

Entry Parameters:

C: $63
DE: FCB Address

Returned Value:

A: Directory Code
H: Extended or Physical Error

The Truncate File function sets the last record of a file to the random record number contained in the referenced FCB. The calling program passes the address of the FCB in register pair DE, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and filetype, and bytes 33 through 35, rO, rl, and r2, specifying the last record number of the file. The last record number is a 24 bit value, stored with the least significant byte first, rO, the middle byte next, rl, and the high byte last, r2. This value can range from 0 to 262,143, which corresponds to a maximum value of 3 in byte r2.

If the file specified by the referenced FCB is password protected, the correct

password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see Function 106).

Function 99 requires that the file specified by the FCB not be open, particularly if the file has been written to. In addition, any activated FCBs naming the file are not valid after Function 99 is called. Close your file before calling Function 99, and then reopen it after the call to continue processing on the file.

Function 99 also requires that the random record number field of the referenced FCB specify a value less than the current file size. In addition, if the file is sparse, the random record field must specify a record in a region of the file where data exists.

Upon return, the Truncate function returns a Directory Code in register A with the value 0 if the Truncate function is successful, or $FF, 255 decimal, if the file is not found or the record number is invalid. Register H is set to zero in both of these cases. If a physical or extended error is encountered, the Truncate function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the program is terminated. Otherwise, the Truncate function returns to the calling program with register A set to $FF and register H set to one of the following physical or extended error codes:

- 01 Disk I/O error
- 02 Read-Only disk
- 03 Read-Only file
- 04 Invalid drive error
- 07 File password error
- 09 ? in filename or filetype field

BDOS function 100: **SET DIRECTORY LABEL**

Entry Parameters:

C: $64
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical or Extended Error

The Set Directory Label function creates a directory label, or updates the existing directory label for the specified drive. The calling program passes in register pair DE the address of an FCB containing the name, type, and extent fields to be assigned to the directory label. The name and type fields of the referenced FCB are not used to locate the directory label in the directory; they are simply copied into the updated or created directory label. The extent field of the FCB, byte 12, contains the user's specificat-on of the directory label data byte. The definition of the directory label data byte is:

bit 7 - Require passwords for password-protected files (Not supported in nonbanked CP/M 3 systems)
bit 6 - Perform access date and time stamping
bit 5 - Perform update date and time stamping
bit 4 - Perform create date and time stamping
bit 0 - Assign a new password to the directory label

If the current directory label is password protected, the correct password must be placed in the first eight bytes of the current DMA, or have been previously established as the default password (see Function 106). If bit 0, the low-order bit, of byte 12 of the FCB is set to 1, it indicates that a new password for the directory label has been placed in the second eight bytes of the current DMA.

Note that Function 100 is implemented as an RSX, DIRLBL.RSX, in nonbanked CP/M 3 systems. If Function 100 is called in nonbanked systems when the DIRLBL.RSX is not resident an error code of $0FF is returned.

Function 100 also requires that the referenced directory contain SFCBs to activate date and time stamping on the drive. If an attempt is made to activate date and time stamping when no SFCBs exist, Function 100 returns an error code of $FF in register A and performs no action. The CP/M 3 INITDIR utility initializes a directory for date and time stamping by placing an SFCB record in every fourth entry of the directory.

Function 100 returns a Directory Code in register A with the value 0 if the directory label create or update is successful, or $FF, 255 decimal, if no space exists in the referenced directory to create a directory label, or if date and time stamping was requested and the referenced directory did not contain SFCBS. Register H is set to zero in both of these cases. If a physical error or extended error is encountered, Function 100 performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error

mode is the default mode, a message identifying the error is displayed at the console and the calling program is terminated. Otherwise, Function 100 returns to the calling program with register A set to $FF and register H set to one of the following physical or extended error codes:

- 01 Disk I/O error
- 02 Read-Only disk
- 04 Invalid drive error
- 07 File password error

BDOS function 101: **RETURN DIRECTORY LABEL DATA**

Entry Parameters:

C: $65
E: Drive

Returned Value:

A: Directory Label Data Byte
H: Physical Error

The Return Directory Label Data function returns the data byte of the directory label for the specified drive. The calling program passes the drive number in register E with 0 for drive A, 1 for drive B, and so on through 15 for drive P in a full sixteen drive system. The format of the directory label data byte is shown below:

bit 7 - Require passwords for password protected files
bit 6 - Perform access date and time stamping
bit 5 - Perform update date and time stamping
bit 4 - Perform create date and time stamping
bit 0 - Directory label exists on drive

Function 101 returns the directory label data byte to the calling program in register A. Register A equal to zero indicates that no directory label exists on the specified drive. If a physical error is encountered by Function 101 when the BDOS Error mode is in one of the return modes (see Function 45), this function returns with register A set to $FF, 25S decimal, and register H set to one of the following:

- 01 Disk I/O error
- 04 Invalid drive error

BDOS function 102: **READ FILE DATE STAMPS AND PASSWORD MODE**

Entry Parameters:

C: $66
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical Error

Function 102 returns the date and time stamp information and password mode for the specified file in byte 12 and bytes 24 through 32 of the specified FCB. The calling program passes in register pair DE, the address of an FCB in which the drive, filename, and filetype fields have been defined.

If Function 102 is successful, it sets the following fields in the referenced FCB:

byte 12 : Password mode field
bit 7 - Read mode
bit 6 - Write mode
bit 4 - Delete mode

Byte 12 equal to zero indicates the file has not been assigned a password. In nonbanked systems, byte 12 is always set to zero.

byte 24 - 27 Create or Access time stamp field
byte 28 - 31 Update time stamp field

The date stamp fields are set to binary zeros if a stamp has not been made. The format of the time stamp fields is the same as the format of the date and time structure described in Function 104.

Upon return, Function 102 returns a Directory Code in register A with the value zero if the function is successful, or $FF, 255 decimal, if the specified file is not found. Register H is set to zero in both of these cases. If a physical or extended error is encountered, Function 102 performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the calling program is terminated. Otherwise, Function 102 returns to the calling program with register A set to $FF and register H set to one of the following physical or extended error codes:

- 01 Disk I/O error
- 04 Invalid drive error
- 09 ? in filename or filetype field

BDOS function 103: **WRITE FILE XFCB**

Entry Parameters:

C: $67
DE: FCB Address

Returned Value:

A: Directory Code
H: Physical Error

The Write File XFCB function creates a new XFCB or updates the existing XFCB for the specified file. The calling program passes in register pair DE the address of an FCB in which the drive, name, type, and extent fields have been defined. The extent field specifies the password mode and whether a new password is to be assigned to the file. The format of the extent byte is shown below:

FCB byte 12 (ex) : XFCB password mode

bit 7 - Read mode
bit 6 - Write mode
bit 5 - Delete mode
bit 0 - Assign new password to the file

If the specified file is currently password protected, the correct password must reside in the first eight bytes of the current DMA, or have been previously established as the default password (see Function 106). If bit 0 is set to 1, the new password must reside in the second eight bytes of the current DMA.

Upon return, Function 103 returns a Directory Code in register A with the value zero if the XFCB create or update is successful, or $FF, 255 decimal, if no directory label exists on the specified drive, or the file named in the FCB is not found, or no space exists in the directory to create an XFCB. Function 103 also returns with $FF in register A if passwords are not enabled by the referenced directory's label. On nonbanked systems, this function always returns with register A = $FF because passwords are not supported. Register H is set to zero in all of these cases. If a physical or extended error is encountered, Function 103 performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the calling program is terminated. Otherwise, Function 103 returns to the calling program with register A set to $FF and register H set to one of the

following physical or extended error codes:

- 01 Disk I/O error
- 02 Read-Only disk
- 04 Invalid drive error
- 07 File password error
- 09 ? in filename or filetype field

BDOS function 104: **SET DATE AND TIME**

Entry Parameters:

C: $68
DE: DAT Address

Returned Value:

none

The Set Date and Time function sets the system internal date and time. The calling program passes the address of a 4-byte structure containing the date and time specification in the register pair DE. The format of the date and time (DAT) data structure is:

byte 0 - 1 Date field
byte 2 Hour field
byte 3 Minute field

The date is represented as a 16-bit integer with day 1 corresponding to January 1, 1978. The time is represented as two bytes: hours and minutes are stored as two BCD digits.

This function also sets the seconds field of the system date and time to zero.

BDOS function 105: **GET DATE AND TIME**

Entry Parameters:

C: $69
DE: DAT Address

Returned Value:

A: seconds
DAT set

The Get Date and Time function obtains the system internal date and time. The calling program passes in register pair DE, the address of a 4-byte

data structure which receives the date and time values. The format of the date and time, DAT, data structure is the same as the format described in Function 104. Function 105 also returns the seconds field of the system date and time in register A as a two digit BCD value.

BDOS function 106: **SET DEFAULT PASSWORD**

Entry Parameters:

C: $6A
DE: Password Address

Returned Value:

none

The Set Default Password function allows a program to specify a password value before a file protected by the password is accessed. When the file system accesses a password-protected file, it checks the current DMA, and the default password for the correct value. If either value matches the file's password, full access to the file is allowed. Note that this function performs no action in nonbanked CP/M 3 systems because file passwords are not supported.

To make a Function 106 call, the calling program sets register pair DE to the address of an 8-byte field containing the password.

BDOS function 107: **RETURN SERIAL NUMBER**

Entry Parameters:

C: $6B
DE: Serial Number Field

Returned Value:

Serial number field set

Function 107 returns the CP/M 3 serial number to the 6-byte field addressed by register pair DE.

BDOS function 108: **GET/SET PROGRAM RETURN CODE**

Entry Parameters:

C: $6C
DE: $0FFFF (Get) or Program Return Code (Set)

Returned Value:

HL: Program Return Code or (no value)

CP/M 3 allows programs to set a return code before terminating. This provides a mechanism for programs to pass an error code or value to a following job step in batch environments. For example, Program Return Codes are used by the CCP in CP/M 3's conditional command line batch facility. Conditional command lines are command lines that begin with a colon, :. The execution of a conditional command depends on the successful execution of the preceding command. The CCP tests the return code of a terminating program to determine whether it successfully completed or terminated in error. Program return codes can also be used by programs to pass an error code or value to a chained program (see Function 47, Chain To Program).

A program can set or interrogate the Program Return Code by calling Function 108. If re 'ster pair DE = $FFFF, then the current Program Return Code is returned in register pair HL. Otherwise, Function 108 sets the Program Return Code to the 'value contained in register pair DE. Program Return Codes are defined in 9.2

BDOS function 109: **GET/SET CONSOLE MODE**

Entry Parameters:

C: $6D
DE: $FFFF (Get) or Console Mode (Set)

Returned Value:

HL: Console Mode or (no value)

A program can set or interrogate the Console Mode by calling Function 109. If register pair DE = $FFFF, then the current Console Mode is returned in register HL. Otherwise, Function 109 sets the Console Mode to the value contained in register pair DE.

The Console Mode is a 16-bit system parameter that determines the action of certain BDOS Console I/O functions. The definition of the Console Mode is:

bit 0 = 1 - CTRL-C only status for Function 1 1.
= 0 - Normal status for Function 1 1.
bit 1 = 1- Disable stop scroll, CTRL-S, start scroll, CTRL-Q, support.
= 0-Enable stop scroll, start scroll support.
bit 2 = 1- Raw console output mode. Disables tab expansion for Functions

2, 9 and 111. Also disables printer echo, CTIRL-P, support.
= 0 - Normal console output mode.

bit 3 = 1 - Disable CTRL-C program termination
= 0 - Enable CTRL-C program termination

bits 8,9 -Console status mode for RSXs that perform console input redirection from a file. These bits determine how the RSX responds to console status requests.

bit 8 = 0, bit 9 = 0 - conditional status
bit 8 = 0, bit 9 = 1 - false status
bit 8 = 1, bit 9 = 0 - true status
bit 8 = 1, bit 9 = 1 - bypass redirection

Note that the Console Mode bits are numbered from right to left.

The CCP initializes the Console Mode to zero when it loads a program unless the program has an RSX that overrides the default value. Refer to Section 2.2.1 for detailed information on Console Mode.

BDOS function 110: **GET/SET OUTPUT DELIMITER**

Entry Parameters:

C: $6E
DE: $FFFF (Get) or
E: Output Delimiter (Set)

Returned Value:

A: Output Delimiter or (no value)

A program can set or interrogate the current Output Delimiter by calling Function 110. If register pair DE = $FFFF, then the current Output Delimiter is returned in register A. Otherwise, Function 110 sets the Output Delimiter to the value contained in register E.

Function 110 sets the string delimiter for Function 9, Print String. The default delimiter value is a dollar sign, $. The CCP restores the Output Delimiter to the default value when a transient program is loaded.

BDOS function 111: **PRINT BLOCK**

Entry Parameters:

C: $6F
DE: CCB Address

Returned Value:

none

The Print Block function sends the character string located by the Character Control Block, CCB, addressed in register pair DE, to the logical console, CONOUT:. If the Console Mode is in the default state (see Section 2.2.1), Function 111 expands tab characters, CTRL-I, in columns of eight characters. It also checks for stop scroll, CTRL-S, start scroll, CTRL-Q, and echoes to the logical list device, LST:, if printer echo, CTRL-P, has been invoked.

The CCB format is:

byte 0 - 1 Address of character string (word value)
byte 2 - 3 Length of character string (word value)

BDOS function 112: **LIST BLOCK**

Entry Parameters:

C: $70
DE: CCB Address

Returned Value:

none

The List Block function sends the character string located by the Character Control Block, CCB, addressed in register pair DE, to the logical list device, LST:.

The CCB format is:

byte 0 - 1 Address of character string (word value)
byte 2 - 3 Length of character string (word value)

BDOS function 152: **PARSE FILENAME**

Entry Parameters:

C: $98
DE: PFCB Address

Returned Value:

HL: Return code
Parsed file control block

The Parse Filename function parses an ASCII file specification and prepares a File Control Block, FCB. The calling program passes the address of a data

structure called the Parse Filename Control Block, PFCB, in register pair DE. The PFCB contains the address of the input ASCII filename string followed by the address of the target FCB as shown below:

```
PFCB: DW INPUT ; Address of input ASCII string
      DW FCB ; Address of target FCB
```

The maximum length of the input ASCII string to be parsed is 128 bytes. The target FCB must be 36 bytes in length.

Function 152 assumes the input string contains file specifications in the following form:

{d:}filename{.typ}{;password}

where items enclosed in curly brackets are optional. Function 152 also accepts isolated drive specifications d: in the input string. When it encounters one, it sets the filename, filetype, and password fields in the FCB to blank.

The Parse Filename function parses the first file specification it finds in the input string. The function first eliminates leading blanks and tabs. The function then assumes that the file specification ends on the first delimiter it encounters that is out of context with the specific field it is parsing. For instance, if it finds a colon, and it is not the second character of the file specification, the colon delimits the entire file specification.

Function 152 recognizes the following characters as detimiters:

space
tab
return
nut]
; (semicolon) - except before password field
= (equal)
< (less than)
> (greater than)
. (period) - except after filename and before filetype
: (colon) - except before filename and after drive
, (comma)
| (vertical bar)
[(left square bracket)
] (right square bracket)

If Function 152 encounters a non-graphic character in the range 1 through 31 not listed above, it treats the character as an error. The Parse Filename function initializes the specified FCB shown in 9.3.

If an error occurs, Function 152 returns an $FFFF in register pair HL.

On a successful parse, the Parse Filename function checks the next item in the input string. It skips over trailing blanks and tabs and looks at the next character. If the character is a null or carriage return, it returns a 0 indicating the end of the input string. If the character is a delimiter, it returns the address of the delimiter. If the character is not a delimiter, it returns the address of the first trailing blank or tab.

If the first non-blank or non-tab character in the input string is a null, 0, or carriage return, the Parse Filename function returns a zero indicating the end of string.

If the Parse Filename function is to be used to parse a subsequent file specification in the input string, the returned address must be advanced over the delimiter before placing it in the PFCB.

9.1.3 BIOS

System Initialization Functions

This section defines the BIOS system initialization routines BOOT, WBOOT, DEVTBL, DEVINI, and DRVTBL.

BIOS Function 0: BOOT

Get Control from Cold Start Loader and Initialize System
Entry Parameters: None
Returned Values: None

The BOOT entry point gets control from the Cold Start Loader in Bank 0 and is responsible for basic system initialization. Any remaining hardware initialization that is not done by the boot ROMS, the Cold Boot Loader, or the LDRBIOS should be performed by the BOOT routine.

BIOS Function 1: WBOOT

Get Control When a Warm Start Occurs
Entry Parameters: None
Returned Values: None

The WBOOT entry point is entered when a warm start occurs. A warm start is performed whenever a user program branches to location 0000H or attempts to return to the CCP.

BIOS Function 20: DEVTBL

Return Address of Character I/O Table
Entry Parameters: None
Returned Values: HL=address of Chrtbl

The DEVTBL and DEVINI entry points allow you to support device assignment with a flexible, yet completely optional system. It replaces the IOBYTE facility of CP/M 2.2.

BIOS Function 21: DEVINI

Initialize Character I/O Device
Entry Parameters: C=device number, 0-15
Returned Values: None

The DEVINI routine initializes the physical character device specified in register C to the baud rate contained in the appropriate entry of the CHRTBL.

BIOS Function 22: DRVTBL

Return Address of Disk Drive Table
Entry Parameters: None
Returned Values:

> HL=Address of Drive Table of Disk Parameter Headers (DPH); Hashing can utilized if specified by the DPHs Referenced by this DRVTBL.
> HL=$ffff if no Drive Table; GENCPM does not set up buffers. Hashing is supported.
> HL=$fffe if no Drive Table; GENCPM does not set up buffers. Hashing is not supported.

The first instruction of this subroutine must be an LXI H,¡address¿ where ¡address¿ is one of the above returned values. The GENCPM utility accesses the address in this instruction to locate the drive table and the disk parameter data structures to determine which system configuration to use.

Character I/O Functions

This section defines the CP/M 3 character I/O routines CONST, CONIN, CONOUT, LIST, AUXOUT, AUXIN, LISTST, CONOST, AUXIST, and

AUXOST. CP/M 3 assumes all simple character I/O operations are performed in eight-bit ASCII, upper and lowercase, with no parity. An ASCII CTRL-Z ($1a) denotes an end-of-file condition for an input device.

In CP/M 3, you can direct each of the five logical character devices to any combination of up to twelve physical devices. Each of the five logical devices has a 16-bit vector in the System Control Block (SCB) . Each bit of the vector represents a physical device where bit 15 corresponds to device zero, and bit 4 is device eleven. Bits 0 through 3 are reserved for future system use.

BIOS Function 2: CONST

Sample the Status of the Console Input Device
Entry Parameters: None
Returned value:

A=$ff if a console character is ready to read
A=$00 if no console character is ready to read

Read the status of the currently assigned console device and return $ff in register A if a character is ready to read, and $ff in register A if no console characters are ready.

BIOS Function 3: CONIN

Read a Character from the Console
Entry Parameters: None
Returned Values: A=Console Character

Read the next console character into register A with no parity. If no console character is ready, wait until a character is available before returning.

BIOS Function 4: CONOUT

Output Character to Console
Entry Parameters: C=Console Character
Returned Values: None

Send the character in register C to the console output device. The character is in ASCII with no parity.

BIOS Function 5: LIST

Output Character to List Device
Entry Parameters: C=Character
Returned Values: None

Send the character from register C to the listing device. The character is in ASCII with no parity.

BIOS Function 6: AUXOUT

Output a Character to the Auxiliary Output Device
Entry Parameters: C=Character
Returned Values: None

Send the character from register C to the currently assigned AUXOUT device. The character is in ASCII with no parity.

BIOS Function 7: AUXIN

Read a Character from the Auxiliary Input Device
Entry Parameters: None
Returned Values: A=Character

Read the next character from the currently assigned AUXIN device into register A with no parity. A returned ASCII CTRL-Z ($1a) reports an end-of-file.

BIOS Function 15: LISTST

Return the Ready Status of the List Device
Entry Parameters: None
Returned Values:

A=$00 if list device is not ready to accept a character
A=$ff if list device is ready to accept a character

BIOS Function 17: CONOST

Return Output Status of Console
Entry Parameters: None
Returned Values:

A=$ff if ready
A=$00 if not ready

The CONOST routine checks the status of the console. CONOST returns an $ff if the console is ready to display another character. This entry point allows for full polled handshaking communications support.

BIOS Function 18: AUXIST

Return Input Status of Auxiliary Port
Entry Parameters: None

Returned Values:

A=$ff if ready
A=$00 if not ready

The AUXIST routine checks the input status of the auxiliary port. This entry point allows full polled handshaking for communications support using an auxiliary port.

BIOS Function 19: AUXOST

Return Output Status of Auxiliary Port
Entry Parameters: None
Returned Values:

A=$ff if ready
A=$00 if not ready

The AUXOST routine checks the output status of the auxiliary port. This routine allows full polled handshaking for communications support using an auxiliary port.

Disk I/O Functions

This section defines the CP/M 3 BIOS disk I/O routines HOME, SELDSK, SETTRK, SETSEC, SETDMA, READ, WRITE, SECTRN, MULTIO, and FLUSH.

BIOS Function 8: HOME

Select Track 00 of the Specified Drive
Entry Parameters: None
Returned Values: None

Return the disk head of the currently selected disk to the track 00 position. Usually, you can translate the HOME call into a call on SETTRK with a parameter of 0.

BIOS Function 9: SELDSK

Select the Specified Disk Drive
Entry Parameters:

C=Disk Drive (0-15)
E=Initial Select Flag

Returned Values:

HL=Address of Disk Parameter Header (DPH) if drive exists
HL=0000H if drive does not exist

Select the disk drive specified in register C for further operations, where register C contains 0 for drive A, 1 for drive B, and so on to 15 for drive P. On each disk select, SELDSK must return in HL the base address of a 25-byte area called the Disk Parameter Header. If there is an attempt to select a nonexistent drive, SELDSK returns HL=$0000 as an error indicator. On entry to SELDSK, you can determine if it is the first time the specified disk is selected. Bit 0, the least significant bit in register E, is set to 0 if the drive has not been previously selected. This information is of interest in systems that read configuration information from the disk to set up a dynamic disk definition table.

BIOS Function 10: SETTRK

Set Specified Track Number
Entry Parameters: BC=Track Number
Returned Values: None

Register BC contains the track number for a subsequent disk access on the currently selected drive. Normally, the track number is saved until the next READ or WRITE occurs.

BIOS Function 11: SETSEC

Set Specified Sector Number
Entry Parameters: BC=Sector Number
Returned Values: None

Register BC contains the sector number for the subsequent disk access on the currently selected drive. This number is the value returned by SECTRN. Usually, you delay actual sector selection until a READ or WRITE operation occurs.

BIOS Function 12: SETDMA

Set Address for Subsequent Disk I/O
Entry Parameters: BC=Direct Memory Access Address
Returned Values: None

Register BC contains the DMA (Direct Memory Access) address for the subsequent READ or WRITE operation. For example, if B = $00 and C = $80 when the BDOS calls SETDMA, then the subsequent read operation

reads its data starting at $80, or the subsequent write operation gets its data from 80H, until the next call to SETDMA occurs.

BIOS Function 13: READ

Read a Sector from the Specified Drive
Entry Parameters: None
Returned Values:

A=$00 if no errors occurred
A=$01 if nonrecoverable error condition occurred
A=$ff if media has changed

Assume the BDOS has selected the drive, set the track, set the sector, and specified the DMA address. The READ subroutine attempts to read one sector based upon these parameters, then returns one of the error codes in register A as described above.

If the value in register A is $00, then CP/M 3 assumes that the disk operation completed properly. If an error occurs, the BIOS should attempt several retries to see if the error is recoverable before returning the error code.

If an error occurs in a system that supports automatic density selection, the system should verify the density of the drive. If the density has changed, return a $ff in the accumulator. This causes the BDOS to terminate the current operation and relog in the disk.

BIOS Function 14: WRITE

Write a Sector to the Specified Disk
Entry Parameters: C=Deblocking Codes
Returned Values:

A=$00 if no error occurred
A=$01 if physical error occurred
A=$02 if disk is Read-Only
A=$ff if media has changed

Write the data from the currently selected DMA address to the currently selected drive, track, and sector. Upon each call to WRITE, the BDOS provides the following information in register C:

0 = deferred write
1 = nondeferred write
2 = deferred write to the first sector of a new data block

This information is provided for those BIOS implementations that do blocking/deblocking in the BIOS instead of the BDOS.

BIOS Function 16: SECTRN

Translate Sector Number Given Translate Table Entry Parameters:

BC=Logical Sector Number
DE=Translate Table Address

Returned Values: HL=Physical Sector Number

SECTRN performs logical sequential sector address to physical sector translation to improve the overall response of CP/M 3.

BIOS Function 23: MULTIO

Set Count of Consecutive Sectors for READ or WRITE
Entry Parameters: C=Multisector Count
Returned Values: None

To transfer logically consecutive disk sectors to or from contiguous memory locations, the BDOS issues a MULTIO call, followed by a series of READ or WRITE calls. This allows the BIOS to transfer multiple sectors in a single disk operation. The maximum value of the sector count is dependent on the physical sector size, ranging from 128 with 128-byte sectors, to 4 with 4096-byte sectors. Thus, the BIOS can transfer up to 16K directly to or from the TPA with a single operation.

BIOS Function 24: FLUSH

Force Physical Buffer Flushing for User-supported Deblocking
Entry Parameters: None
Returned Values:

A=$00 if no error occurred
A=$001 if physical error occurred
A=$002 if disk is Read-Only

The flush buffers entry point allows the system to force physical sector buffer flushing when your BIOS is performing its own record blocking and deblocking. The BDOS calls the FLUSH routine to ensure that no dirty buffers remain in memory.

9.1.4 Memory Select and Move Functions

This section defines the memory management functions MOVE, XMOVE, SELMEM, and SETBNK.

BIOS Function 25: MOVE

Memory-to-Memory Block Move
Entry Parameters:

HL=Destination address
DE=Source address
BC=Count

Returned Values: HL and DE must point to next bytes following move operation

The BDOS calls the MOVE routine to perform memory to memory block moves to allow use of the Z80 LDIR instruction or special DMA hardware, if available. Note that the arguments in HL and DE are reversed from the Z8O machine instruction, necessitating the use of XCHG instructions on either side of the LDIR. The BDOS uses this routine for all large memory copy operations. On return, the HL and DE registers are expected to point to the next bytes following the move.

Usually, the BDOS expects MOVE to transfer data within the currently selected bank or common memory. However, if the BDOS calls the XMOVE entry point before calling MOVE, the MOVE routine must perform an interbank transfer.

BIOS Function 27: SELMEM

Select Memory Bank
Entry Parameters: A=Memory Bank
Returned Values; None

The SELMEM entry point is only present in banked systems. The banked version of the CP/M 3 BDOS calls SELMEM to select the current memory bank for further instruction execution or buffer references. You must preserve or restore all registers other than the accumulator, A, upon exit.

BIOS Function 28: SETBNK

Specify Bank for DMA Operation
Entry Parameters: A=Memory Bank
Returned Values: None

SETBNK only occurs in the banked version of CP/M 3. SETBNK specifies the bank that the subsequent disk READ or WRITE routine must use for memory transfers. The BDOS always makes a call to SETBNK to identify the DMA bank before performing a READ or WRITE call. Note that the BDOS does not reference banks other than 0 or 1 unless another bank is specified by the BANK field of a Data Buffer Control Block (BCB).

BIOS Function 29: XMOVE

Set Banks for Following MOVE
Entry Parameters:

B=destination bank
C=source bank

Returned Values: None

XMOVE is provided for banked systems that support memory-to- memory DMA transfers over the entire extended address range. Systems with this feature can have their data buffers located in an alternate bank instead of in common memory, as is usually required. An XMOVE call affects only the following MOVE call. All subsequent MOVE calls apply to the memory selected by the latest call to SELMEM. After a call to the XMOVE function, the following call to the MOVE function is not more than 128 bytes of data.

Clock Support Function

This section defines the clock support function TIME.

BIOS Function 26: TIME

Get and Set Time
Entry Parameters: C=Time Get/Set Flag
Returned values: None

The BDOS calls the TIME function to indicate to the BIOS whether it has just set the Time and Date fields in the SCB, or whether the BDOS is about to get the Time and Date from the SCB. On entry to the TIME function, a zero in register C indicates that the BIOS should update the Time and Date fields in the SCB. A $ff in register C indicates that the BDOS has just set the Time and Date in the SCB and the BIOS should update its clock. Upon exit, you must restore register pairs HL and DE to their entry values.

Table 9.1: System Control Block

Offset-7	Description
00 - 04	Reserved For System Use
05	BDOS version number
06 - 09	User Flags
0A - 0F	Reserved For System Use
10 - 11	Program Error return code
12 - 19	Reserved For System Use
1A	Console Width (columns)
1B	Console Column Position
1C	Console Page Length
1D - 21	Reserved For System Use
22 - 23	CONIN Redirection flag
24 - 25	CONOUT Redirection flag
26 - 27	AUXIN Redirection flag
2A - 2B	LSTOUT Redirection flag
2C	Page Mode
2D	Reserved For System Use
2E	CTRL-H Active
2F	Rubout Active
30 - 32	Reserved For System Use
33 - 34	Console Mode
35 - 36	Reserved For System Use
37	Output Delimiter
39 - 3B	Reserved For System Use
3C - 3D	Current DMA Address
3E	Current Disk
3F - 43	Reserved For System Use
44	Current User Number
45 - 49	Reserved For System Use
4A	BDOS Multi-Sector Count
4B	BDOS Error Mode
4C - 4F	Drive Search Chain (DISKS A: E: F:)
50	Temporary File Drive
51	Error Disk
52 - 56	Reserved For System Use
57	BDOS flags
58 - 5C	Date Stamp
5D - 5E	Common Memory Base Address
5F - 63	Reserved For System Use

Table 9.2: Program Return Codes

Code	Meaning
0000 - FEFF	Successful return
FF00 - FFFE	Unsuccessful return
0000	The CCP initializes the Program Return Code to zero unless the program is loaded as the result of program chain.
FF00 - FFFC	Reserved
FFFD	The program is terminated because of a fatal BDOS error.
FFFE	The program is terminated by the BDOS because the user typed a CTRL-C.

Table 9.3: FCB Format

Location	Contents
byte 0	The drive field is set to the specified drive. If the drive is not specified the default drive code is used. 0 = default 1 = A 2 = B.
byte 1-8	The name is set to the specified filename. All letters are converted to upper-case. If the name is not eight characters long the remaining bytes in the filename field are padded with blanks. If the filename has an asterisk * all remaining bytes in the filename field are filled in with question marks ?. An error occurs if the filename is more than eight bytes long.
byte 9-11	The type is set to the specified filetype. If no filetype is specified the type field is initialized to blanks. All letters are converted to upper-case. If the type is not three characters long the remaining bytes in the filetype field are padded with blanks. If an asterisk * occurs all remaining bytes are filled in with question marks ?. An error occurs if the type field is more than three bytes long.
byte 12-15	Filled in with zeros.
byte 16-23	The password field is set to the specified password. If no password is specified it is initialized to blanks. If the password is less than eight characters long remaining bytes are padded with blanks. All letters are converted to upper-case. If the password field is more than eight bytes long an error occurs. Note that a blank in the first position of the password field implies no password was specified.
byte 24-31	Reserved for system use.

9.2 NextZXOS

9.3 NextZXOS

A ZX Spectrum I/O system supported by the ZX Spectrum Next. This Documentation is largely from Garry Lancaster's DOCs at https://gitlab.com/thesmog358/tbblue/blo
Before making any calls disable writes to Layer 2 in the $0000-$3fff area with port $123b.

9.3.1 +3DOS compatible API

Generally to make these calls, you need to set up: place ROM 2 at $0000-$3fff, RAM bank 7 at $c000-$ffff, stack below $bfe0, and set up the parameters for the call in the indicated registers. Call the function at its address. Then, restore your system to its previous configuration. In general the carry bit of F is cleared on error with the error code in A. Calls generally affect the contnts of AF, BC, DE, HL, and IX leaving AF', BC', DE', HL', IY, and SP intact. To simplify, descriptions will assume this is true and only indicate exceptions to the rule.

$0056 **IDE_STREAM_OPEN**

Open stream to a channel

$0059 **IDE_STREAM_CLOSE**

Close stream and attached channel

$005c **IDE_STREAM_IN**

Get byte from current stream

$005f **IDE_STREAM_OUT**

Write byte to current stream

$0062 **IDE_STREAM_PTR**

Get or set pointer information for current stream

$00A0 **IDE_VERSION**

Get IDEDOS version number

$00A3 **IDE_INTERFACE**

Initialise card interfaces $00A6 **IDE_INIT**

Initialise IDEDOS

$00A9 **IDE_DRIVE**

Get unit handle

$00AC **IDE_SECTOR_READ**

Low-level sector read

$00AF **IDE_SECTOR_WRITE**

Low-level sector write

$00B2 **IDE_FORMAT**

Format a partition

$00B5 **IDE_PARTITION_FIND**

Find named partition

$00B8 **IDE_PARTITION_NEW**

Create partition

$00BB **IDE_PARTITION_INIT**

Initialise partition

$00BE **IDE_PARTITION_ERASE**

Delete a partition

$00C1 **IDE_PARTITION_RENAME**

Rename a partition

$00C4 **IDE_PARTITON_READ**

Read a partition entry

$00C7 **IDE_PARTITION_WRITE**

Write a partition entry

$00CA **IDE_PARTITION_WINFO**

Write type-specific partition information

$00CD **IDE_PARTITION_OPEN**

Open a partition

$00D0 **IDE_PARTITION_CLOSE**

Close a partition

$00D3 **IDE_PARTITION_GETINFO**

Get byte from type-specific partition information

$00D6 **IDE_PARTITION_SETINFO**

Set byte in type-specific partition information

$00D9 (217) **IDE_SWAP_OPEN**

Open a swap partition (file)

A swap file will be opened on success. The file must be unfragmented or the call will return with error $4a (rc_fragmented). Further swap related calls will use units related to the selected block size.

Entry:

A bits 0-6 = block size, 1-32 512-byte sectors
A bit 7 = 0, open available system swap file (c:/nextzxos/swp-N.p3d) that is large enough.
BC=max block size required
A bit 7 = 1, open indicated swap file (c:/nextzxos/swp-N.p3d) that is large enough.
BC=$ff terminated pointer to swap file name

Exit: IX=swap handle

$00DC **IDE_SWAP_CLOSE**

Close a swap partition

$00DF **IDE_SWAP_OUT**

Write block to swap partition

$00E2 **IDE_SWAP_IN**

Read block from swap partition

$00E5 (231) **IDE_SWAP_EX**

Exchange block with swap partition

Deprecated, use IDE_SWAP_IN and IDE_SWAP_OUT

$00E8 **IDE_SWAP_POS**

Get current block number in swap partition

$00EB **IDE_SWAP_MOVE**

Set current block number in swap partition

$00EE **IDE_SWAP_RESIZE**

Change block size of swap partition

$00F1 (241) **IDE_DOS_MAP**

Map drive to partition or physical device

Entry:

A = Unit 0-15 (4=RAMdisk)
BC=partition number (A != 4)
A = $ff for filesystem image
BC=$ff terminated image filename
L=drive letter A-P

Exit: IX is preserved

$00F4 (244) **IDE_DOS_UNMAP**

Unmap drive

Remove mapping from the specified drive

Entry: L=drive letter A-P
Exit: IX is preserved

$00F7 (247) **IDE_DOS_MAPPING**

Get drive mapping

Entry:

L=drive letter A-P
BC=pointer to 18-byte buffer

Exit:

Zero set: drive not mapped
Zero clear:
A=Unit 0-15, 4=RAMdisk, $ff=filesystem image
BC=partition number
buffer contains description or nothing
IX is preserved

$00FA **IDE_DOS_UNPERMANENT**

Remove permanent drive mapping

$00FD (253) **IDE_SNAPLOAD**

Load a snapshot

Loads a supported snapshot (currently .Z80, .SNA, .O, or .P). Call must be made in layer 0 mode. For .O and .P SP bust be ¡$8000.

Entry: HL=$ff terminated filespec Exit: Does not return on success.

$0100 **DOS_INITIALISE**

Initialise +3DOS

$0103 **DOS_VERSION**

Get +3DOS issue and version numbers

$0106 (262) **DOS_OPEN**

Create and/or open a file

Opeens the file corresponding to the filename pointed to by HL.

Entry:

B = File number 0-15
C = Access mode required
1 = exclusive-read
2 = exclusive-write
3 = exclusive-read/write
5 = shared-read
D = Create action
0 - Error if file does not exist
1 - Create and open new file with header
2 - Create and open new file without header
E = Open action
0 - Error if file already exist s
1 - Open file and read header
2 - Open file and ignore header
3 - Erase filename.BAK, rename file to filename.BAK, follow create action
4 - Erase existing file, follow create action
HL = Address of filename (no wildcards, unless D=0 and E=1 or 2)

Exit:

New file: Zero set
Existing file: Zero clear

$0109 **DOS_CLOSE**

Close a file

$010C **DOS_ABANDON**

Abandon a file

$010F **DOS_REF_HEAD**

Point at the header data for this file

$0112 **DOS_READ**

Read bytes into memory

$0115 **DOS_WRITE**

Write bytes from memory

$0118 **DOS_BYTE_READ**

Read a byte

$011B **DOS_BYTE_WRITE**

Write a byte

$011E (286) **DOS_CATALOG**

Catalog disk directory

Fills a buffer with part of the directory indicated by the path pointed to by HL. If the buffer is filled with zeros, the call places the first n entries from the directory into the buffer. If it begins with a directory entry, it fills the buffer starting with that directory entry. Directory entries are 13 bytes long and consist of an 8-byte filename, a 3-byte type (extension), and a 2-byte size. Shorter filenames and types are padded with spaces. The most significant bit of the filename and type may be set to indicicate extra information about the file.

Entry:

B = size of buffer in entries
C = Filter
bit 0: Include system files

bit 1: Set bit 7 of f7 if entry has a valid long file name
bit 2: Include directories, set bit 7 of f8 for directories
bits 3-7: reserved (0)
DE = Address of buffer
HL = Address of directory name (wildcards permitted)

Exit:

B = number of completed entries, if the size that is passed, there may be more entries
HL = Directory handle for long filenames, used by IDE_GET_LFN

$0121 (289) **DOS_FREE_SPACE**

Free space on disk

How much free space is on this disk?

Entry: A = Drive, ASCII A-P
Exit:

HL = Free space in kB up to 65535k
BCDE = Free space in kB

$0124 **DOS_DELETE**

Delete a file

$0127 **DOS_RENAME**

Rename a file

$012A **DOS_BOOT**

Boot an operating system or other program

$012D **DOS_SET_DRIVE**

Set/get default drive

$0130 **DOS_SET_USER**

Set/get default user number

$0133 (307) **DOS_GET_POSITION**

Get file pointer for random access

Entry: B = File number
Exit: DEHL = File Pointer

$0136 **DOS_SET_POSITION**

Set file pointer for random access

$0139 (313) **DOS_GET_EOF**

Get end of file position for random access

Gets the first position after the last byte written without considering soft-EOF. This does not accect the file pointer.

Entry: B = File number
Exit: DEHL = EOF position

$013C **DOS_GET_1346**

Get memory usage in pages 1 3 4 6

$013F **DOS_SET_1346**

Re-allocate memory usage in pages 1 3 4 6

$0142 **DOS_FLUSH**

Bring disk up to date

$0145 **DOS_SET_ACCESS**

Change open file's access mode

$0148 **DOS_SET_ATTRIBUTES**

Change a file's attributes

$014B **DOS_OPEN_DRIVE**

Open a drive as a single file

$014E **DOS_SET_MESSAGE**

Enable/disable error messages

$0151 **DOS_REF_XDPB**

Point at XDPB for low level disk access

$0154 **DOS_MAP_B**

Map B: onto unit 0 or 1

$0157 **DD_INTERFACE**

Is the floppy disk driver interface present?

$015A **DD_INIT**

Initialise disk driver

$015D **DD_SETUP**

Specify drive parameters

$0160 **DD_SET_RETRY**

Set try/retry count

$0163 **DD_READ_SECTOR**

Read a sector

$0166 **DD_WRITE_SECTOR**

Write a sector

$0169 **DD_CHECK_SECTOR**

Check a sector

$016C **DD_FORMAT**

Format a track

$016F **DD_READ_ID**

Read a sector identifier

$0172 **DD_TEST_UNSUITABLE**

Test media suitability

$0175 **DD_LOGIN**

Log in disk, initialise XDPB

$0178 **DD_SEL_FORMAT**

Pre-initialise XDPB for DD FORMAT

$017B **DD_ASK_1**

Is unit 1 (external drive) present?

$017E **DD_DRIVE_STATUS**

Fetch drive status

$0181 **DD_EQUIPMENT**

What type of drive?

$0184 **DD_ENCODE**

Set intercept routine for copy protection

$0187 **DD_L_XDPB**

Initialise an XDPB from a disk specification

$018A **DD_L_DPB**

Initialise a DPB from a disk specification

$018D **DD_L_SEEK**

uPD765A seek driver

$0190 **DD_L_READ**

uPD765A read driver

$0193 **DD_L_WRITE**

uPD765A write driver

$0196 **DD_L_ON_MOTOR**

Motor on, wait for motor-on time

$0199 **DD_L_T_OFF_MOTOR**

Start the motor-off ticker

$019C **DD_L_OFF_MOTOR**

Turn the motor off

$01a2 **IDE_IDENTIFY**

Return IDE drive identity information

$01a5 **IDE_PARTITIONS**

Get number of open partitions

$01b1 (433) **IDE_PATH**

Create, delete, change or get directory

Read or manipulate the IDE path (directory), does not affect what the current default drive is.
rc_path_change: change directory
rc_path_get: get current directory
rc_path_make: create a new directory

rc_path_delete: remove a directory

Entry:

A=reason code
0=rc_path_change
1=rc_path_get
2=rc_path_make
3=rc_path_delete
HL=pointer to $ff terminated filespec or buffer for returned filespec data (256 bytes)

Exit: IY is affected

$01b4 (436) **IDE_CAPACITY**

Get card capacity

Entry: C=unit (0 or 1)
Exit: DEHL=total card capacity in 512-byte sectors

$01b7 (439) **IDE_GET_LFN**

Get long filename

Obtain a long file name and other file information.

Entry:

HL=address of filespec provided by last DOS_CATALOG call
IX=directory handle provided by last DOS_CATALOG call
DE=address of a file entry filled by the last DOS_CATALOG call
BC=address of a 261-byte buffer to receive the long filename

Exit:

buffer contains file name
BC=date (MS-DOS format)
DE=time (MS-DOS format)
HLIX=filesize, bytes (for directories 0)

$01ba (442) **IDE_BROWSER**

File browser

Run the file browser. The filetypes buffer pointed to by HL is a $ff terminated buffer of filetypes. Each filetype consists of a length followed by a type description. Types are the file extension, wild cards ? and * are

permitted, a colon (:) and an optional command to execute on the filetype with vertical bar representing the filemane in the command. The help text may contain window control codes, but if the character size is changed, it must be returned to size 5 at the end of the string.

Entry:

IY=$5c3a (ERR_NR)
HL=address of supported filetypes buffer
DE=address of $ff terminated help text
A=browser capability mask
bit 0: BROSWERCAPS_COPY - files may be copied
bit 1: BROSWERCAPS_RENAME - files/dirs may be renamed
bit 2: BROSWERCAPS_MKDIR - directories may be created
bit 3: BROSWERCAPS_ERASE - files/dirs may be deleted
bit 4: BROSWERCAPS_REMOUNT - SD card may be remounted
bit 5: BROSWERCAPS_UNMOUNT - drives may be unmounted
bit 6: reserved (0)
bit 7: BROSWERCAPS_SYSCFG - system use - use browser.cfg

Exit:

Zero flag set: file selected (ENTER pressed)
HL=address of short $ff terminated filename in RAM 7
DE=address of long $ff teminated filename in RAM 7
Zero flag clear: no file selected (SPACE/BREAK pressed)

$01bd (445) **IDE_BANK**

Allocate or free 8K pages in ZX or DivMMC memory

$01c0 **IDE_BASIC**

Execute a BASIC command line

$01c3 **IDE_WINDOW_LINEIN**

Input line from current window stream

$01c6 **IDE_WINDOW_STRING**

Output string to current window stream

$01c9 **IDE_INTEGER_VAR**

Get or set NextBASIC integer variable

$01cc **IDE_RTC**

Query the real-time-clock module

$01cf **IDE_DRIVER**

Access the driver API

$01d2 **IDE_MOUNT**

Unmount/remount SD cards

$01d2 **IDE_MOUNT**

Unmount/remount SD cards

$01d5 **IDE_MODE**

Query NextBASIC display mode info, or change mode

$01d8 **IDE_TOKENISER**

Convert BASIC between plain text & tokenised forms

9.3.2 esxDOS compatible API

Appendix A

Ports

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 $6B (107) DMA Control (Z80 Mode, 3.01.02)

Port $0F (15) DAC A

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $1F (31) Kempston/Mega Drive Joystick 1/DAC B
Read

bit 7 = ”start” button

bit 6 = A/X button
bit 5 = C/Z button
bit 4 = Fire/C/Y button
bit 3 = Up
bit 2 = Down
bit 1 = Left
bit 0 = Right

Disable with Nextreg $05
Write

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08 Port $37 (55) Kempston/Mega Drive Joystick 2
Read

bit 7 = "start" button
bit 6 = A/X button
bit 5 = C/Z button
bit 4 = Fire/C/Y button
bit 3 = Up
bit 2 = Down
bit 1 = Left
bit 0 = Right

Disable with Nextreg $05
Write ($00 on reset, 3.01.04)

bits 7-6 = Select I/O Mode
00 = Bit Bang
01 = Clock
10 = UART
11 = Reserved (don't use)
bit 5 = Reserved, must be 0
bit 4 = Select Joystick Port for Read
0 = Left
1 = Right
bits 3-1 = Reserved, must be $00
bit 0 = Pin 7 state (both ports)
- Bit Bang - bit 0 on pin 7
- Clock - clock on pin 7
0 = Slow clock (Fsys/2048 = 12.672 kHz)

1 = Fast clock (Fsys/8 = 3.5 MHz)
– UART - Pin 7 = TX, Pin 9 = RX 0 = ESP
1 = Pi

** A Runt clock may appear in the first cycle
The I/O mode should be set by writing this port first followed by enabling io mo de on the joysticks with a write to nextreg 0x05.

Port $3F (63) DAC A

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $4F (79) DAC C

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $57 (87) Sprite Attributes
Byte 1

bits 7-0 = LSB of X coordinate (bit 8 is in byte 3)

Byte 2

bits 7-0 = LSB of Y coordinate (bit 8 is in byte 5)

Byte 3

bits 7-4 = Palette Offset
bit 3 = Enable X Mirror
bit 2 = Enable Y Mirror
bit 1 = Enable Roration
bit 0 = By Sprite Type
Anchor = MSB of X coordinate
Relative = Enable relative palette offset

Byte 4

bit 7 = Enable visibility
bit 6 = Enable Byte 5
bit 5-0 = Pattern Index (“name”)

Byte 5 (optional)
Anchor

bit 7-6 = type and pattern

00 = 8-bit color
01 = relative
10 = 4-bit color, lower half of pattern (bytes 0-127)
11 = 4-bit color, upper half of pattern (byets 128-255)
bit 5 = Attached relative sprite type
0 = composite
1 = big sprite
bit 4-3 = X-axis scale factor
00 = 1×
01 = 2×
10 = 4×
11 = 8×
bit 2-1 = Y-axis scale factor
bit 0 = MSB of Y coordinate

Composite Relative

bits 7-6 = 01
bit 5 = N6
8-bit
Reserved, must be 0
4-bit
0 = lower half of pattern (bytes 0-127)
1 = upper half of pattern (bytes 128-255)
bit 4-3 = X-axis scale factor
bit 2-1 = Y-axis scale factor
bit 0 = Enable relative pattern offset

Big-sprite Relative

bits 7-6 = 01
bit 5 = N6
8-bit
Reserved, must be 0
4-bit
0 = lower half of pattern (bytes 0-127)
1 = upper half of pattern (bytes 128-255)
bit 4-1 = Reserved, must be 0
bit 0 = Enable relative pattern offset

Port $5B (91) Sprite Pattern
Load data into sprite pattern memory auto-incrementing. Port $303B can

be used to set the starting sprite pattern number.

Port $5F (95) DAC D

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $6B (107) DMA Control (Next Mode, 3.01.02)

Port $DF (223) DAC A/C, SpecDrum

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $E3 (227) divMMC Control
Disable with bit 2 of Nextreg $09

bit 7 = conmem, enable divMMC memory
bit 6 = mapram, enable divMMC allRAM mode
bits 3-0 = bank, selected divMMC ram bank for $2000-$3FFF region

- conmem can be used to manually control divMMC mapping. When enabled
  $0000-$1FFF contains esxDOS ROM or esxDOS page 3
  $2000-$3FFF contains esxDOS RAM page selected by bits 3-0
- divMMC automatically maps itself in when instruction fetches hit specific addresses in the ROM. When this happens, the esxDOS ROM (or divMMC bank 3 if mapram is set) appears in $0000-$1FFF and the selected divMMC bank appears as RAM in $2000-$3FFF
- bit 6 can only be set, once set only a power cycle can reset it. nextreg $09 bit 3 can be set to reset this bit.

divMMC automapping is normally disabled by NextZXOS see nextreg $06 bit 4.
Port $E7 (231) SPI $\overline{CS}$ (SD card, flash, rpi)
Disable with bit 2 of Nextreg $09

Port $EB (235) SPI $\overline{DATA}$ (SD card, flash, rpi)
Disable with bit 2 of Nextreg $09

Port $F1 (241) DAC A (precedence over $xxFD)

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $F3 (243) DAC B

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $F9 (249) DAC C (precedence over $xxFD)

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $FB (251) DAC D

bits 7-0 = DAC Value

Disable with bit 3 of Nextreg $08

Port $FE (254) ULA

bits 7-5 = Unused
bit 4 = enable ear output
bit 3 = enable mic output
bit 2-0 = border colour

Port $FF (255) Timex Sinclair/Floating Bus

bit 7 = memory paging (not on ZX Next)
bit 6 = Disable generation of interrupts
bit 5-3 = Hi-res mode color combination
- 000 = Black on white (indexes 0, 135)
- 001 = Blue on Yellow (indexes 1, 134)
- 010 = Green on Magenta (indexes 2, 133)
- 011 = Cyan on Red (indexes 3, 132)
- 100 = Red on Cyan (indexes 4, 131)
- 101 = Magenta on Green (indexes 5, 130)
- 110 = Yellow on Blue (indexes 6, 129)
- 111 = White on Black (indexes 7, 128)

bit 2-0 = ULA Mode
- 000 = Normal ULA address
- 001 = Alternate ULA address
- 010 = Hi-color mode
- 110 = Hi-res mode

Disable with bit 2 of Nextreg $08

A.2 16-bit

Port $103B (4155) I^2C SCL (rtc, rpi)

Port $113B (4411) I^2C SDA (rtc, rpi)

Port $123B (4667) Layer 2
Bit 4 = 0

bits 7-6 = Video RAM bank select
00 = first 16k
01 = second 16k
10 = third 16k
11 = first 48k
bit 5 = Reserved, must be 0
bit 4 = 0
bit 3 = Shadow layer 2 select
bit 2 = Enable layer 2 read paging
bit 1 = Layer 2 visible (mirrored in register $69)
bit 0 = Enable layer 2 write paging

Bit 4 = 1

bits 7-5 = Reserved, must be 0
bit 4 = 1
bit 3 = Reserved, must be 0
bit 2-0 = 16k bank relative offset

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 = select prescalar part
0 = Bits 6-0 of prescalar

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

Port $153B (5435) UART control

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 $1FFD (8189) Plus 3 Memory Paging Control

bits 7-3 = Unused, nust be 0
bit 2 = High bit of ROM selection (low bit is in Port $7FFD)
00 = ROM0 = 128k editor and menu system
01 = ROM1 = 128k syntax checker
10 = ROM2 = +3DOS
11 = ROM3 = 48k BASIC
bit 1 = Special mode: Low bit of memory configuration number
bit 0 = Paging mode
0 = Normal
1 = Special

You should echo writes to $5B67

Port $243B (9275) Next Register Select

Port $253B (9531) Next Register Data

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

Port $7FFD (32765) Memory Paging Control

bits 6-7 = reserved

bit 5 = Lock memory paging
bit 4 = low bit of ROM Select (high bit is in Port $1FFD)
00 = ROM0 = 128k editor and menu system
01 = ROM1 = 128k syntax checker
10 = ROM2 = +3DOS
11 = ROM3 = 48k BASIC
bit 3 = Shadow screen toggle
bits 2-0 = LSB of Bank number for slot 4 (MSB is in Port $DFFD)

Disable with bit 5 port $7FFD

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

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

Port $BF3D (48957) ULAplus register

Port $BFFD (49149) Turbosound Data

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) Next Memory Bank Select

bits 7-4 = Reserved, must be 0
bits 3-0 = MSB of bank number for slot 4 (LSB is in 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 Buttons

bits 7-4 = Wheel delta since last read
bit 3 = fourth button
bit 2 = middle button
bit 1 = left button
bit 0 = right button

Port $FBDF (64479) Kempston Mouse X

bits 7-0 = X coordinate of mouse

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 $FFDF (65503) Kempston Mouse Y

bits 7-0 = Y coordinate of mouse (0-192)

Port $FFFD (65533) Turbo Sound Next Control Select Chip

bit 7 = 1
bit 6 = Enable left
bit 5 = Enable Right
bits 4-2 = Reserved, must be 1
bits 1-0 = AY chip select
00 = Unused
01 = AY 3
10 = AY 2
11 = AY 1

Select Register

bit 7 = 0
bits 6-4 = Reserved, must be 000
bits 3-0 = Register Number

Appendix B

Registers

B.1 ZX Spectrum Next Registers

The ZX Next 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.

Register (R) $00 (0) ⇒ Machine ID

- 00000001 = DE1A
- 00000010 = DE2A
- 00000101 = FBLABS
- 00000110 = VTRUCCO
- 00000111 = WXEDA
- 00001000 = EMULATORS*
- 00001010 = ZX Spectrum Next*
- 00001011 = Multicore
- 11101010 = ZX Spectrum Next on ZX-DOS fpga platform *
- 11111010 = ZX Spectrum Next Anti-brick*

* Relevant for ZX Next machines & software

Register (R) $01 (1) ⇒ Core Version

- bits 7-4 = Major version number

- bits 3-0 = Minor version number
  See register $0E for sub minor version number

Register (R/W) $02 (2) ⇒ Reset
Read

- bit 7 = Expansion bus $\overline{\text{RESET}}$ Asserted
- bits 6-2 = Reserved
- bit 1 = Last reset was Hard reset
- bit 0 = Last reset was Soft reset

Write

- bit 7 = Generate/Release Expansion bus $\overline{\text{RESET}}$
- bits 6-2 = Reserved, must be 0
- bit 1 = generate Hard reset
- bit 0 = generate Soft reset

Register (R/W) $03 (3) ⇒ Machine Type
A write to this register disables the boot rom in config mode
bits 2-0 select machine type when in config mode

- bit 7 = (W) Display Timing change enable (allow changes to bits 6-4) (0 on hard reset)
- bits 6-4 = Display Timing
- bit 3 = Display Timing user lock control
  Read
  - 0 = No user lock on display timing
  - 1 = User lock on display timing

  Write
  - 1 = Apply user lock on display timing (0 on hard reset)
- bits 2-1 = Machine Type
  Machine Types/Display Timings
  - 000 or 001 = ZX 48K
  - 010 = ZX 128K/+2 (Grey)
  - 011 = ZX +2A-B/+3e/Next Native
  - 100 = Pentagon 128K

Register (W) $04 (4) ⇒ Configuration Mapping

- bits 7-5 = Reserved, must be 0
- bits 4-0 = 16k SRAM bank mapping* ($00 on hard reset)
  *Maps a 16k SRAM bank over the bottom 16k. Applies only in config mode when the bootrom is disabled

Register (R/W) $05 (5) ⇒ Peripheral 1 Settings

- bits 7-6 = joystick 1 mode (MSB)
- bits 5-4 = joystick 2 mode (MSB)
- bit 3 = joystick 1 mode (LSB)
- bit 2 = 50/60 Hz mode (0 = 50Hz, 1 = 60Hz)
- bit 1 = joystick 2 mode (LSB)
- bit 0 = Enable Scandoubler

Joystick modes

- 000 = Sinclair 2 (67890)
- 001 = Kempston 2 (port $37)
- 010 = Kempston 1 (port $1F)
- 011 = Megadrive 1 (port $1F)
- 100 = Cursor
- 101 = Megadrive 2 (port $37)
- 110 = Sinclair 1 (12345)
- 111 = I/O Mode Both joysticks are places in I/O Mode if either is set to I/O Mode. The underlying joystick type is not changed and reads of this register will continue to return the last joystick type. Ehether the joystick is in io mode or not is invisible but this state can be cleared either through reset or by re-writing the gegister with joystick type not equal to 111. Recovery time for a normal joystick read after leaving I/O Mode is at most 64 scan lines.

Register (R/W) $06 (6) ⇒ Peripheral 2 Settings

- bit 7 = F8 CPU Speed Hotkey Enable (1 on reset)
- bit 6 = Enable classic audio mode (beep and tape to internal speaker, other audio to ear and HDMI, 3.01.02)
- bit 5 = F3 50Hz/60Hz Hotkey Enable (1 on reset)
- bit 4 = divMMC Automap/NMI Enable (0 on hard reset)
- bit 3 = NMI Button Enable (0 on hard reset)
- bit 2 = PS/2 Mode (0 = keyboard, 1 = mouse)
- bits 1-0 = PSG Mode (00 = YM, 01 = AY, 11 = hold all PSGs in Reset)

Register (R/W) $07 (7) ⇒ Turbo mode
Read

- bits 7-6 = Reserved
- bits 5-4 = Current Actual CPU Speed

- bits 3-2 = Reserved
- bits 1-0 = Current Selected CPU Speed (00 on reset)

Write

- bits 7-2 = Reserved, must be 0
- bits 1-0 = Select CPU Speed

CPU Speeds

- 00 = 3.5MHz
- 01 = 7MHz
- 10 = 14MHz
- 11 = 28MHz

Register (R/W) $08 (8) ⇒ Peripheral 3 Settings

- bit 7 = 128K Banking Unlock (inverse of port $7FFD, bit 5) (0 on reset)
- bit 6 = Disable RAM and Port Contention (0 on reset)
- bit 5 = PSG Stereo Mode Control (0 = ABC, 1 = ACB) (0 on hard reset)
- bit 4 = Enable internal speaker (1 on hard reset)
- bit 3 = Enable DACs (0 on hard reset)
- bit 2 = Enable read of port $FF (Timex) (0 on hard reset)
- bit 1 = Enable Multiple PSGs (0 on hard reset)
- bit 0 = Enable Issue 2 Keyboard

Register (R/W) $09 (9) ⇒ Peripheral 4 setting:

- bit 7 = PSG 2 Mono Enable (0 on hard reset)
- bit 6 = PSG 1 Mono Enable (0 on hard reset)
- bit 5 = PSG 0 Mono Enable (0 on hard reset)
- bit 4 = Sprite ID lockstep enable (1 = Nextreg $34 and IO Port $303B are in lockstep, 0 on reset)
- bit 3 = divMMC mapRAM bit Control (reset bit 7 of port $E3)
- bit 2 = HDMI audio mute (0 on hard reset)
- bits 1-0 = scanlines
  - 00 = scanlines off
  - 01 = scanlines 12.5%
  - 10 = scanlines 25%
  - 11 = scanlines 50%

  In Sprite lockstep, NextREG $34 and Port $303B are in lockstep

Register (R) $0E (14) ⇒ Core Version (sub minor number)

- bits 7-0 = Core sub minor version number
  (see register $01 for the major and minor version number)

Register (R/W) $10 (16) ⇒ Core Boot
Read

- bits 7-2 = Reserved
- bit 1 = Drive button pressed
- bit 0 = NMI button pressed

Write

- bit 7 = Reboot FPGA using selected core (0 on reset)
- bits 6-5 = Reserved, must be 0
- bits 4-0 = Core ID
  Core ID with bits 4-0 can only be set in configuration mode

Register (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
  Only writable in config mode

Register (R/W) $12 (18) ⇒ Layer 2 Active 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)

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

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

Register (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)

Register (R/W) $15 (21) ⇒ Sprite and Layer System Setup

- bit 7 = LoRes mode (0 on reset)
- bit 6 = Sprite priority (1 = sprite 0 on top, 0 = sprite 127 on top) (0 on reset)
- bit 5 = Enable sprite clipping in over border mode (0 on reset)
- bits 4-2 = set layers priorities (000 on reset)
  - 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 = Enable Sprites Over border (0 on reset)
- bit 0 = Enable Sprites (0 on reset)

Register (R/W) $16 (22) ⇒ Layer 2 Horizontal Scroll Control

- bits 7-0 = X Offset (0-255)(0 on reset)

Register (R/W) $17 (23) ⇒ Layer 2 Vertical Scroll Control

- bits 7-0 = Y Offset (0-191)(0 on reset)

Register (R/W) $18 (24) ⇒ Layer 2 Clip Window Definition

- 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

Register (R/W) $19 (25) ⇒ Sprite Clip Window Definition

- 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.

Register (R/W) $1A (26) ⇒ Layer 0 (ULA/LoRes) Clip Window Definition

- 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

Register (R/W) $1B (27) ⇒ Layer 3 (Tilemap) Clip Window Definition

- 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.

Register (R/W) $1C (28) ⇒ Clip Window Control
Read

- bits 7-6 = Layer 3 Clip Index
- bits 5-4 = Layer 0/1 Clip Index
- bits 3-2 = Sprite clip index
- bits 1-0 = Layer 2 Clip Index

Write

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

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

- bits 7-1 = Reserved
- bit 0 = Active line MSB

Register (R) $1F (31) ⇒ Active video line (LSB)

- bits 7-0 = Active line LSB (0-255)

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

- bit 7 = (R) ULA asserting interrupt
- bit 7 = (W) Reserved, must be 0
- bits 6-3 = Reserved, must be 0
- bit 2 = Disable ULA Interrupt (0 on reset)
- bit 1 = Enable Line Interrupt (0 on reset)
- bit 0 = MSB of Line Interrupt line value (0 on reset)

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

- bits 7-0 = Line Interrupt line value LSB (0-255)(0 on reset)

Register (R/W) $26 (38) ⇒ ULA Horizontal Scroll Control

- bits 7-0 = ULA X Offset (0-255) (0 on reset)

Register (R/W) $27 (39) ⇒ ULA Vertical Scroll Control

- bits 7-0 = ULA Y Offset (0-191) (0 on reset)

Register (R/W) $28 (40) ⇒ Stored Palette Value and PS/2 Keymap Address MSB
Read

- bits 7-0 = Stored palette value (see NextREG $44)

Write

- bits 7-1 = Reserved, must be 0
- bit 0 = PS/2 Keymap Address MSB

Register (W) $29 (41) ⇒ PS/2 Keymap Address LSB

- bits 7-0 = PS/2 Keymap Address LSB

Register (W) $2A (42) ⇒ PS/2 Keymap Data MSB

- bits 7-1 = Reserved, must be 0
- bit 0 = PS/2 Keymap Data MSB

Register (W) $2B (43) ⇒ PS/2 Keymap Data LSB

- bits 7-0 = PS/2 Keymap Data LSB

(writing this register auto-increments the address)

Register (R/W) $2C (44) ⇒ DAC B Mirror (Left)/ I^2S Left Sample MSB
Read

- bits 7-0 = I^2S Left Sample MSB

Write

- bits 7-0 = 8-bit sample left DAC ($80 on reset)

Register (R/W) $2D (45) ⇒ DAC A+D Mirror (mono/ I^2S Sample LSB
Read

- bits 7-0 = I^2S Last Sample LSB

Write

- bits 7-0 = 8-bit sample DACs A + D ($80 on reset)

Register (R/W) $2E (46) ⇒ DAC C Mirror (Right/ I^2S Risht Sample MSB
Read

- bits 7-0 = I^2S Right Sameple MSB

Write

- bits 7-0 = 8-bit sample Right DACs C ($80 on reset)

Register (R/W) $2F (47) ⇒ Layer 3 (Tilemap) Horizontal Scroll Control
MSB

- bits 7-2 = Reserved, must be 0
- bits 1-0 = X Offset MSB ($00 on reset)

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

Register (R/W) $30 (48) ⇒ Layer 3 (Tilemap) Horizontal Scroll Control
LSB

- bits 7-0 = X Offset LSB ($00 on reset)

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

Register (R/W) $31 (49) ⇒ Layer 3 (Tilemap) Vertical Scroll Control

- bits 7-0 = Y Offset (0-255))$00 on reset)

Register (R/W) $32 (50) ⇒ Layer 1,0 (LoRes) Horizontal Scroll Control)

- bits 7-0 = X Offset (0-255) ($00 on reset)

Layer 1,0 (LoRes) scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

Register (R/W) $33 (51) ⇒ Layer 1,0 (LoRes) Vertical Scroll Control)

- bits 7-0 = Y Offset (0-191) ($00 on reset)

Layer 1,0 (LoRes) scrolls in "half-pixels" at the same resolution and smoothness as Layer 2.

Register (R/W) $34 (52) ⇒ Sprite Number
Lockstep (NextReg $09 bit 4 set)

- bit 7 = Pattern address offset (Add 128 to pattern address)
- bits 6-0 = Sprite number 0-127, Pattern number 0-63
  effectively performs an out to port $303B

No Lockstep (NextReg $09 bit 4 clear)

- bit 7 = Reserved, must be 0
- bits 6-0 = Sprite number 0-127

This register selects which sprite has its attributes connected to the sprite attribute registers

Register (W) $35 (53) ⇒ Sprite Attribute 0

- bits 7-0 = Sprite X coordinate LSB (MSB in NextReg $37)

Register (W) $36 (54) ⇒ Sprite Attribute 1

- bits 7-0 = Sprite Y coordinate LSB (MSB in NextReg $39)

Register (W) $37 (55) ⇒ Sprite Attribute 2

- bits 7-4 = 4-bit Palette offset
- bit 3 = Enable horizontal mirror (reverse)
- bit 2 = Enable vertical mirror (reverse)
- bit 1 = Enable 90^O Clockwise Rotation

Normal Sprites

- bit 0 = X coordinate MSB

Relative Sprites

- bit 0 = Palette offset is relative to anchor sprite

Rotation is applied before mirroring

Register (W) $38 (56) ⇒ Sprite Attribute 3

- bit 7 = Enable Visiblity
- bit 6 = Enable Attribute 4 (0 = Attribute 4 effectively $00)
- bits 5-0 = Sprite Pattern Number

Register (W) $39 (57) ⇒ Sprite Attribute 4
Normal Sprites

- bit 7 = 4-bit pattern switch (0 = 8-bit sprite, 1 = 4-bit sprite)
- bit 6 = Pattern number bit-7 for 4-bit, 0 for 8-bit
- bit 5 = Type of attached relative sprites (0 = Composite, 1 = Unified)
- 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

Relative, Composite Sprites

- bit 7-6 = 01
- bit 5 = Pattern number bit-7 for 4-bit, 0 for 8-bit
- 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 = Pattern number is relative to anchor

Relative, Unified Sprites

- bit 7-6 = 01
- bit 5 = Pattern number bit-7 for 4-bit, 0 for 8-bit
- bits 4-1 = 0000
- bit 0 = Pattern number is relative to anchor

Register (R/W) $40 (64) ⇒ Palette Index Select

- bits 7-0 = Palette Index Number

Selects the palette index to change the associated colour

For ULA only, INKs are mapped to indices 0 through 7, BRIGHT INKs to indices 8 through 15, PAPERs to indices 16 through 23 and BRIGHT PAPERs to indices 24 through 31. In EnhancedULA mode, INKs come from a subset of indices from 0 through 127 and PAPERs from a subset of indices

from 128 through 255.

The number of active indices depends on the number of attribute bits assigned to INK and PAPER out of the attribute byte.

In ULAplus mode, the last 64 entries (indices 192 to 255) hold the ULAplus palette. The ULA always takes border colour from PAPER for standard ULA and Enhanced ULA

Register (R/W) $41 (65) ⇒ 8-bit Palette Data

- bits 7-0 = Colour Entry in RRRGGGBB format

The lower blue bit of the 9-bit internal colour will be the logical or of bits 0 and 1 of the 8-bit entry. After each write, the palette index auto-increments if aut-increment has been enabled (NextReg $43 bit 7), Reads do not auto-increment.

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

- bits 7-0 = Attribute byte's INK representation mask (7 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.

Register (R/W) $43 (67) ⇒ Palette Control

- bit 7 = 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 = Sprite first palette
  - 110 = Sprite second palette
  - 011 = Layer 3 first palette
  - 111 = Layer 3 second palette
- bit 3 = Select Sprite 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 EnhancedULA mode if 1. (0 after a reset)

Register (R/W) $44 (68) ⇒ 9-bit Palette Data
Non Level 2

1st write
- bits 7-0 = MSB (RRRGGGBB)

2nd write
- bits 7-1 = Reserved, must be 0
- bit 0 = LSB (B)

Level 2

1st write
- bits 7-0 = MSB (RRRGGGBB)

2nd write
- bit 7 = Priority
- bits 6-1 = Reserved, must be 0
- bit 0 = LSB (B)

9-bit Palette Data is entered in two consecutive writes; the second write autoincrements the palette index if auto-increment is enabled in NextREG $43 bit 7

If writing an L2 palette, the second write's D7 holds the L2 priority bit which if set (1) brings the colour defined at that index on top of all other layers. If you also need the same colour in regular priority (for example: for enviromental masking) you will have to set it up again, this time with no priority.

Reads return the second byte and do not autoincrement.

Register (R/W) $4A (74) ⇒ Fallback Colour Value

- bits 7-0 = 8-bit colour if all layers are transparent ($E3 on reset)

(black on reset = 0)

Register (R/W) $4B (75) ⇒ Sprite Transparency Index

- bits 7-0 = Index value ($E3 if reset)

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

Register (R/W) $4C (76) ⇒ Level 3 Transparency Index

- bits 7-4 = Reserved, must be 0

- bits 3-0 = Index value ($0F on reset)

Register (R/W) $50 (80) ⇒ MMU Slot 0 Control

- bits 7-0 = 8k RAM page at position $0000 to $1FFF ($ff on reset)

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

Register (R/W) $51 (81) ⇒ MMU Slot 1 Control

- bits 7-0 = 8k RAM page at position $2000 to $3FFF ($ff on reset)

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

Register (R/W) $52 (82) ⇒ MMU Slot 2 Control

- bits 7-0 = 8k RAM page at position $4000 to $5FFF ($0A on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $53 (83) ⇒ MMU Slot 3 Control

- bits 7-0 = 8k RAM page at position $6000 to $7FFF ($0B on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $54 (84) ⇒ MMU Slot 4 Control

- bits 7-0 = 8k RAM page at position $8000 to $9FFF ($04 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $55 (85) ⇒ MMU Slot 5 Control

- bits 7-0 = 8k RAM page at position $A000 to $BFFF ($05 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $56 (86) ⇒ MMU Slot 6 Control

- bits 7-0 = 8k RAM page at position $C000 to $DFFF ($00 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Register (R/W) $57 (87) ⇒ MMU Slot 7 Control

- bits 7-0 = 8k RAM page at position $E000 to $FFFF ($01 on reset)

Pages can be from 0 to 223 on a fully expanded Next.

Writing to ports $1FFD, $7FFD and $DFFD writes $FF 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.

Register (W) $60 (96) ⇒ Copper Data 8-bit Write

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

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

After a write, the index is auto-incremented to the next memory position.

Register (W) $61 (97) ⇒ Copper Address LSB

- bits 7-0 = Copper instruction memory address LSB (0 on reset)

Register (W) $62 (98) ⇒ Copper Control

- 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) (0 on reset)

Register (W) $63 (99) ⇒ Copper Data 16-bit Write

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

The 16-bit value is written in pairs. The first 8-bits are the MSB and are destined for an even copper instruction address. The sesond 8-bits are the LSB and are destined for an odd copper instruction address.

After each write, the copper address is auto-incremented to the next memory position.

After a write to an odd address, the all 16-bits are written to copper memory at once.

Register (R/W) $64 (100) ⇒ Vertical Line Count Offset (3.01.05)

- bits 7-0 = Offset added to the vertical line counter
  affects copper, line interrupt and active line count.

Normally the ula's pixel row 0 aligns with vertical line count 0. With a non-zero offset, the ula's pixel row 0 will align with the vertical line offset.
Eg, if the offset is 32 then vertical line 32 will correspond to the first pixel row in the ula and vertical line 0 will align with the first pixel row of the tilemap and sprites.

* Since a change in offset takes effect when the ula reaches row 0, the change can take up to one frame to occur. Register (R/W) $68 (104) ⇒ ULA Control

- bit 7 = Disable ULA output (0 on reset)
- bit 6-5 = Color blending control for layering modes 6 & 7
  - 00 = ULA as blend colour
  - 01 = No blending
  - 10 = ULA/Tilemap mix result as blend colour
  - 11 = Tilemap as blend colour
- bit 4 = Cancel entries in 8x5 matrix for extended keys (3.01.04)
- bit 3 = Enable ULAplus (0 on reset)
- bit 2 = Enable ULA half pixel scroll (0 on reset)
  may change
- bit 1 = Reserved (must be 0)
- bit 0 = Enable stencil mode (0 on reset)
  When ULA and Layer 3 are enabled, if either are transparent, the result is transparent, otherwise the result is the logical AND of both colours.

Register (R/W) $69 (105) ⇒ Display Control 1

- bit 7 = Layer 2 Enable (Port $123B bit 1 alias)
- bit 6 = ULA Shadow display enable (Port $7FFD bit 3 alias)
- bits 5-0 = Timex alias (Port $FF alias)

Register (R/W) $6A (106) ⇒ Layer 1,0 (LoRes) Control

- bits 7-6 = reserved, must be 0
- bit 5 = Enable Radistan (16-colour) (0 on reset)
- bit 4 = Radistan DFILE switch (xor with bit 0 of port $ff) (0 on reset)
- bits 3-0 = Radistsan palette offset (0 on reset)
- bits 1-0 = ULAplus palette offset (0 on reset)

Register (R/W) $6B (107) ⇒ Layer 3 (Tilemap) Control

- bit 7 = Layer 3 Enable (0 on reset)

- bit 6 = Layer 3 Size control (0 on reset)
  - 0 = 40x32
  - 1 = 80x32
- bit 5 = Disable Arrtibute Entry (0 on reset)
- bit 4 = palette select (0 on reset)
- bit 3 = Enable Text mode (1-bit tilemap) (0 on reset)
- bit 2 = Reserved, must be 0
- bit 1 = Activate 512 tile mode (0 on reset)
- bit 0 = Enable Layer 3 on top of ULA (0 on reset)

Register (R/W) $6C (108) ⇒ Default Layer 3 Attribute*

- bits 7-4 = Palette Offset ($00 on reset)
- bit 3 = X mirror (0 on reset)
- bit 2 = Y mirror (0 on reset)
- bit 1 = Rotate (0 on reset)
- bit 0 = Bit 8 of the tile number (512 tile mode) (0 on reset)
- bit 0 = ULA over tilemap (256 tile mode) (0 on reset)

*Active tile attribute if bit 5 of nextreg $6B is set.

Register (R/W) $6E (110) ⇒ Layer 3 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 ($2C on reset)

Soft Reset default $2C - This is because the address is $6C00 so the MSB is $6C. But the stored value is only the lower 6 bits so it's an offset into the 16k Bank 5. To calculate therefore subtract $40 leaving you with $2C.

The value written is an offset into the 16k 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.

Register (R/W) $6F (111) ⇒ Layer 3 Tile Definitions Base Address

- bits 7-6 = Read back as zero, write values ignored
- bits 5-0 = MSB of address of the tilemap in Bank 5 ($0C on reset)

Soft Reset default $0C - This is because the address is $4C00 so the MSB is $4C. But the stored value is only the lower 6 bits so it's an offset into the 16k Bank 5. To calculate therefore subtract $40 leaving you with $0C.

The value written is an offset into the 16k 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.

Register (R/W) $70 (112) ⇒ Layer 2 Control

- bits 7-6 = Reserved, must be 0
- bits 5-4 = Resolution (00 on soft reset)
  - 00 = 256 × 192 × 256
  - 01 = 320 × 256 × 256
  - 10 = 640 × 256 × 16
  - 11 = Do not use
- bits 3-0 = Palette offset ($0 on soft reset)

Register (W) $75 (117) ⇒ Sprite Attribute 0 (Auto-incrementing)
See nextreg $35

Register (W) $76 (118) ⇒ Sprite Attribute 1 (Auto-incrementing)
See nextreg $36

Register (W) $77 (119) ⇒ Sprite Attribute 2 (Auto-incrementing)
See nextreg $37

Register (W) $78 (120) ⇒ Sprite Attribute 3 (Auto-incrementing)
See nextreg $38

Register (W) $79 (121) ⇒ Sprite Attribute 4 (Auto-incrementing)
See nextreg $39

Register (R/W) $7F (127) ⇒ User Register 0

- bits 7-0 = User Register ($FF on reset)

Caution NextReg numbers above $7F are inaccessible to the Copper

Register (R/W) $80 (128) ⇒ Expansion Bus Enable
Immediate

- bit 7 = Expansion Bus Enable (0 on hard reset)
- bit 6 = Enable ROMCS ROM replacement from divmmc banks 14/15 (experimental, 3.01.03)
- bit 5 = I/O cycle Disable/Ignore $\overline{\text{IORQULA}}$ (0 on hard reset)
- bit 4 = Memory cycle Disable/Ignore $\overline{\text{ROMCS}}$ (0 on hard reset)

After Soft Reset (Copied into bits 7-4)

- bit 3 = Expansion Bus Enable (0 on hard reset)
- bit 2 = Enable ROMCS ROM replacement from divmmc banks 14/15 (experimental, 3.01.03)
- bit 1 = I/O cycle Disable/Ignore $\overline{\text{IORQULA}}$ (0 on hard reset)
- bit 0 = Memory cycle Disable/Ignore $\overline{\text{ROMCS}}$ (0 on hard reset)

Register (R/W) $81 (129) ⇒ Expansion Bus Control

- bit 7 = (R) Expansion bus $\overline{\text{ROMCS}}$ asserted
- bits 6-5 = Reserved, must be 0
- bit 4 = (W) Propagate max CPU clock at all times (0 on hard reset)
- bits 3-2 = Reserved, must be 0
- bits 1-0 = Max CPU Speed when Expansion Bus is enabled ($00 on hard reset, currently fixed at $00)
  - 00 = 3.5 MHz
  - 01 = 7 MHz
  - 10 = 14 MHz
  - 11 = 28 MHz

Register (R/W) $82 (130) ⇒ Internal Port decoding control 1/4

- bit 7 = Enable Kempston Port 2 (Port $37) (1 on reset)
- bit 6 = Enable Kempston Port 1 (Port $1F) (1 on reset)
- bit 5 = Enable DMA (Port $6B) (1 on reset)
- bit 4 = Enable +3 Floating Bus (1 on reset)
- bit 3 = Enable +3 Paging (Port $1FFD) (1 on reset)
- bit 2 = Enable Next Memory Paging (Port $DFFD) (1 on reset)
- bit 1 = Enable Paging (Port $7FFD) (1 on reset)
- bit 0 = Enable Timex (Port $FF) (1 on reset)

Register (R/W) $83 (131) ⇒ Internal Port decoding control 2/4

- bit 7 = Enable Layer 2 (Port $123B) (1 on reset)
- bit 6 = Enable Sprites (Ports $57, $5B, $303B) (1 on reset)
- bit 5 = Enable Kempston Mouse (Ports $FADF, $FBDF, $FFDF) (1 on reset)
- bit 4 = Enable UART (Ports $133B, $143B, $153B) (1 on reset)
- bit 3 = Enable SPI (Ports $E7, $EB) (1 on reset)
- bit 2 = Enable I^2C (Ports $103B, $113B) (1 on reset)
- bit 1 = Enable Multiface (two variable ports) (1 on reset)
- bit 0 = Enable divMMC (Port $E3) (1 on reset)

Register (R/W) $84 (132) ⇒ Internal Port decoding control 3/4

- bit 7 = Enable SPECdrum Mono DAC (Port $DF) (1 on reset)
- bit 6 = Enable Covox/GS Mono DAC (Port $B3) (1 on reset)
- bit 5 = Enable Pentagon/ATM DAC (Port $FB) (1 on reset)
- bit 4 = Enable Covox Stereo DAC (Ports $0F, $4F) (1 on reset)
- bit 3 = Enable Profi/Covox Stereo DAC (Ports $3F, $5F) (1 on reset)
- bit 2 = Enable Soundrive DAC Mode 2 (Ports $F1, $F3, $F9, $FB) (1 on reset)
- bit 1 = Enable Soundrive DAC Mode 1 (Ports $0F, $1F, $4F, $5F) (1 on reset)
- bit 0 = Enable AY (Ports $FFFD, $BFFD) (1 on reset)

Register (R/W) $85 (133) ⇒ Internal Port decoding control 4/4

- bit 7 = Enable configuration of port decoding on soft reset (3.01.01)
- bits 6-2 = Reserved
- bit 1 = Enable DMA port $0B (3.01.02)
- bit 0 = Enable ULAplus (Ports $BF3B, $FF3B) (1 on reset)

Register (R/W) $86 (134) ⇒ Expansion Port decoding control 1/4

- bit 7 = Enable Kempston Port 2 (Port $37) (1 on reset)
- bit 6 = Enable Kempston Port 1 (Port $1F) (1 on reset)
- bit 5 = Enable DMA (Port $6B) (1 on reset)
- bit 4 = Enable +3 Floating Bus (1 on reset)
- bit 3 = Enable +3 Paging (Port $1FFD) (1 on reset)
- bit 2 = Enable Next Memory Paging (Port $DFFD) (1 on reset)
- bit 1 = Enable Paging (Port $7FFD) (1 on reset)
- bit 0 = Enable Timex (Port $FF) (1 on reset)

Register (R/W) $87 (135) ⇒ Expansion Port decoding control 2/4

- bit 7 = Enable Layer 2 (Port $123B) (1 on reset)
- bit 6 = Enable Sprites (Ports $57, $5B, $303B) (1 on reset)
- bit 5 = Enable Kempston Mouse (Ports $FADF, $FBDF, $FFDF) (1 on reset)
- bit 4 = Enable UART (Ports $133B, $143B, $153B) (1 on reset)
- bit 3 = Enable SPI (Ports $E7, $EB) (1 on reset)
- bit 2 = Enable I^2C (Ports $103B, $113B) (1 on reset)
- bit 1 = Enable Multiface (two variable ports) (1 on reset)
- bit 0 = Enable divMMC (Port $E3) (1 on reset)

Register (R/W) $88 (136) ⇒ Expansion Port decoding control 3/4

- bit 7 = Enable SPECdrum Mono DAC (Port $DF) (1 on reset)
- bit 6 = Enable Covox/GS Mono DAC (Port $B3) (1 on reset)
- bit 5 = Enable Pentagon/ATM DAC (Port $FB) (1 on reset)
- bit 4 = Enable Covox Stereo DAC (Ports $0F, $4F) (1 on reset)
- bit 3 = Enable Profi/Covox Stereo DAC (Ports $3F, $5F) (1 on reset)
- bit 2 = Enable Soundrive DAC Mode 2 (Ports $F1, $F3, $F9, $FB) (1 on reset)
- bit 1 = Enable Soundrive DAC Mode 1 (Ports $0F, $1F, $4F, $5F) (1 on reset)
- bit 0 = Enable AY (Ports $FFFD, $BFFD) (1 on reset)

Register (R/W) $89 (137) ⇒ Expansion Port decoding control 4/4

- bit 7 = Enable configuration of port decoding on soft reset (3.01.01)
- bits 6-2 = Reserved
- bit 1 = Enable DMA port $0B (3.01.02)
- bit 0 = Enable ULAplus (Ports $BF3B, $FF3B) (1 on reset)

The Internal Port Decoding Enables always apply.

When the Expansion Bus is enabled, the Expansion Bus Port Decoding Enables are logically ANDed with the Internal Enables. A result of 0 for the corresponding bit indicates the internal device is *disabled*. If the Expansion Bus is enabled, this allows I/O cycles for disabled ports to propagate to the Expansion Bus, otherwise corresponding I/O cycles to the Expansion Bus are filtered.

Register (R/W) $8A (138) ⇒ Expansion Bus I/O Propagate Control

- bits 7-3 = Reserved, must be 0
- bit 4 = Propagate port $FF I/O Cycles (0 on hard reset, 3.01.02)
- bit 3 = Propagate port $1FFD I/O Cycles (0 on hard reset)
- bit 2 = Propagate port $DFFD I/O Cycles (0 on hard reset)
- bit 1 = Propagate port $7FFD I/O Cycles (0 on hard reset)
- bit 0 = Propagate port $FE I/O Cycles (1 on hard reset, 3.01.03: 0 on hard reset)

Register (R/W) $8C (140) ⇒ Alternate ROM
Immediate

- bit 7 = Alt ROM Enable (0 on hard reset)
- bit 6 = Alt ROM visible ONLY during writes (0 on hard reset)
- bit 5 = Reserved, must be 0
- bit 4 = 48k ROM Lock (0 on hard reset)

After Soft Reset (copied into bits 7-4)

- bit 3 = Alt ROM Enable (0 on hard reset)
- bit 2 = Alt ROM visible ONLY during writes (0 on hard reset)
- bit 1 = Reserved, must be 0
- bit 0 = 48k ROM Lock (0 on hard reset)

Register (R/W) $8E (142) ⇒ Legacy Memory Paging Control (3.01.01)

- bit 7 = Bank number bit 3
- bit 6-4 = Bank number bits 2-0
- bit 3 = Enable change ram page (read as 1)
- bit 2 = Paging mode
  - 0 = Normal paging mode
  - 1 = Special paging mode (lot bit of memory configuration)

  Normal Paging Mode
- bits 1-0 = ROM selection

  Special (all RAM) Paging Mode
- bits 1-0 = RAM configuration selection

Register (R/W) $90 (144) ⇒ Pi GPIO output enable 1/4

- bit 7 = Enable Pin 7 (0 on reset)
- bit 6 = Enable Pin 6 (0 on reset)
- bit 5 = Enable Pin 5 (0 on reset)
- bit 4 = Enable Pin 4 (0 on reset)
- bit 3 = Enable Pin 3 (0 on reset)
- bit 2 = Enable Pin 2 (0 on reset)
- bit 1 = Enable Pin 1 (cannot be enabled) (0 on reset)
- bit 0 = Enable Pin 0 (cannot be enabled) (0 on reset)

Register (R/W) $91 (145) ⇒ Pi GPIO output enable 2/4

- bit 7 = Enable Pin 15 (0 on reset)
- bit 6 = Enable Pin 14 (0 on reset)
- bit 5 = Enable Pin 13 (0 on reset)
- bit 4 = Enable Pin 12 (0 on reset)
- bit 3 = Enable Pin 11 (0 on reset)
- bit 2 = Enable Pin 10 (0 on reset)
- bit 1 = Enable Pin 9 (0 on reset)
- bit 0 = Enable Pin 8 (0 on reset)

Register (R/W) $92 (146) ⇒ Pi GPIO output enable 3/4

- bit 7 = Enable Pin 23 (0 on reset)

- bit 6 = Enable Pin 22 (0 on reset)
- bit 5 = Enable Pin 21 (0 on reset)
- bit 4 = Enable Pin 20 (0 on reset)
- bit 3 = Enable Pin 19 (0 on reset)
- bit 2 = Enable Pin 18 (0 on reset)
- bit 1 = Enable Pin 17 (0 on reset)
- bit 0 = Enable Pin 16 (0 on reset)

Register (R/W) $93 (147) ⇒ Pi GPIO output enable 4/4

- bits 7-4 = Reserved
- bit 3 = Enable Pin 27 (0 on reset)
- bit 2 = Enable Pin 26 (0 on reset)
- bit 1 = Enable Pin 25 (0 on reset)
- bit 0 = Enable Pin 24 (0 on reset)

Register (R/W) $98 (152) ⇒ Pi GPIO Pin State 1/4

- bit 7 = Pin 7 Data (1 on reset)
- bit 6 = Pin 6 Data (1 on reset)
- bit 5 = Pin 5 Data (1 on reset)
- bit 4 = Pin 4 Data (1 on reset)
- bit 3 = Pin 3 Data (1 on reset)
- bit 2 = Pin 2 Data (1 on reset)
- bit 1 = Pin 1 Data (1 on reset)
- bit 0 = Pin 0 Data (1 on reset)

Register (R/W) $99 (153) ⇒ Pi GPIO Pin State 2/4

- bit 7 = Pin 15 Data (1 on reset)
- bit 6 = Pin 14 Data (1 on reset)
- bit 5 = Pin 13 Data (1 on reset)
- bit 4 = Pin 12 Data (1 on reset)
- bit 3 = Pin 11 Data (1 on reset)
- bit 2 = Pin 10 Data (1 on reset)
- bit 1 = Pin 9 Data (1 on reset)
- bit 0 = Pin 8 Data (1 on reset)

Register (R/W) $9A (154) ⇒ Pi GPIO Pin State 3/4

- bit 7 = Pin 23 Data (1 on reset)
- bit 6 = Pin 22 Data (1 on reset)
- bit 5 = Pin 21 Data (1 on reset)
- bit 4 = Pin 20 Data (1 on reset)

- bit 3 = Pin 19 Data (1 on reset)
- bit 2 = Pin 18 Data (1 on reset)
- bit 1 = Pin 17 Data (1 on reset)
- bit 0 = Pin 16 Data (1 on reset)

Register (R/W) $9B (155) ⇒ Pi GPIO Pin State 4/4

- bits 7-4 = Reserved
- bit 3 = Pin 27 Data (1 on reset)
- bit 2 = Pin 26 Data (1 on reset)
- bit 1 = Pin 25 Data (1 on reset)
- bit 0 = Pin 24 Data (1 on reset)

Register (R/W) $A0 (160) ⇒ Pi Peripheral Enable

- bits 7-6 = Reserved, must be 0
- bit 5 = Enable UART on GPIO 14, 15 (0 on reset)*
- bit 4 = Communication Type (0 on reset)
  - 0 = Rx to GPIO 15, Tx to GPIO 14 (Pi)
  - 1 = Rx to GPIO 14, Tx to GPIO 15 (Pi Hats)
- bit 3 = Enable I^2C on GPIO 2, 3 (0 on reset)*
- bits 2-1 = Reserved, must be 0
- bit 0 = Enable SPI on GPIO 7, 8, 9, 10, 11 (0 on reset)*

*Overrides GPIO Enables

Register (R/W) $A2 (162) ⇒ Pi I^2S Audio Control

- bits 7-6 = I^2S State ($00 on reset)
  - 00 = I^2S Disabled
  - 01 = I^2S is mono, source R
  - 10 = I^2S is mono, source L
  - 11 = I^2S is stereo
- bit 5 = Reserved, must be 0
- bit 4 = Audio Flow Direction (0 on reset)
  - 0 = PCM_DOUT to Pi, PCM_DIN from Pi (Hats)
  - 1 = PCM_DOUT from Pi, PCM_DIN to Pi (Pi)
- bit 3 = Mute left (0 on reset)
- bit 2 = Mute right (0 on reset)
- bit 1 = Reserved must be 1 (3.01.05)
- bit 0 = Direct I^2S audio to EAR on port $FE (0 on reset)

Register (R/W) $A3 (163) ⇒ Pi I^2S Clock Divide (Master Mode) (removed in 3.01.05)

- bits 7-0 = Clock divide value ($0B on reset)

Divider = $\frac{538461}{\text{Rate}} - 1$ or Rate = $\frac{538461}{\text{Divider}+1}$

Register (R) $B0 (176) ⇒ Extended Keys 0 (3.01.05)

- bit 7 = 1 if ; pressed
- bit 6 = 1 if p̈ressed
- bit 5 = 1 if , pressed
- bit 4 = 1 if . pressed
- bit 3 = 1 if UP pressed
- bit 2 = 1 if DOWN pressed
- bit 1 = 1 if LEFT pressed
- bit 0 = 1 if RIGHT pressed

Register (R) $B1 (177) ⇒ Extended Keys 1 (3.01.05)

- bit 7 = 1 if DELETE pressed
- bit 6 = 1 if EDIT pressed
- bit 5 = 1 if BREAK pressed
- bit 4 = 1 if INV VIDEO pressed
- bit 3 = 1 if TRUE VIDEO pressed
- bit 2 = 1 if GRAPH pressed
- bit 1 = 1 if CAPS LOCK pressed
- bit 0 = 1 if EXTEND pressed

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

B.2 AY-3-8912

(R/W) $00 (0) ⇒ Channel A fine tune

- bits 7-0 = Channel A frequency bits 7-0

(R/W) $01 (1) ⇒ Channel A coarse tune

- bits 7-4 = Reserved
- bits 4-0 = Channel A frequency bits 11-8

(R/W) $02 (0) ⇒ Channel B fine tune

- bits 7-0 = Channel A frequency bits 7-0

(R/W) $03 (1) ⇒ Channel B coarse tune

- bits 7-4 = Reserved
- bits 4-0 = Channel A frequency bits 11-8

(R/W) $04 (0) ⇒ Channel C fine tune

- bits 7-0 = Channel A frequency bits 7-0

(R/W) $05 (1) ⇒ Channel C coarse tune

- bits 7-4 = Reserved
- bits 4-0 = Channel A frequency bits 11-8

(R/W) $06 (6) ⇒ Noise period

- bits 7-5 = Reserved
- bits 4-0 = Noise period to noise generator

(R/W) $07 (7) ⇒ Mixer control I/O Enable

Active low (0=enable, 1= disable)

- bit 7-6: Reserved
- bit 5: Channel C noise enable
- bit 4: Channel B noise enable
- bit 3: Channel A noise enable
- bit 2: Channel C tone enable
- bit 1: Channel B tone enable
- bit 0: Channel A tone enable

(R/W) $0A (10) ⇒ Channel A amplitude

- bits 7-5 = Reserved
- bit 4 = Amplitude mode
  - 0=fixed amplitude
  - 1=use envelope generator (bits 0-3 ignored)
- bits 0-3 = value of fixed amplitude

(R/W) $0B (11) ⇒ Channel B amplitude

like channel A amplitude

(R/W) $0C (12) ⇒ Channel C amplitude

like channel A amplitude

(R/W) $0D (13) ⇒ Envelope period fine

- bits 7-0 = Envelop period LSB

(R/W) $0E (14) ⇒ Envelope period coarse

- bits 7-0 = Envelop period MSB

(R/W) $0F (15) ⇒ Envelope shape

- bits 7-4 = Reserved
- 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 1-0 = Alternate & Hold
  - 00=generator resets after each cycle
  - 01=hold final value
  - 10=generator reverses direction each cycle
  - 11=hold initial value

B.3 zxDMA

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	

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 IX Bit Operations

Table C.26: $DDCB00-$DDCBFF

Opcode	Mnemonic	Sz	T	Opcode	Mnemonic	Sz	T
$DDCB06	rlc (ix+x)	4	23	$DDCB0E	rrc (ix+x)	4	23
$DDCB16	rl (ix+x)	4	23	$DDCB1E	rr (ix+x)	4	23
$DDCB26	sla (ix+x)	4	23	$DDCB2E	sra (ix+x)	4	23
$DDCB36	sll (ix+x)	4	23	$DDCB3E	srl (ix+x)	4	23
$DDCB46	bit 0,(ix+x)	4	20	$DDCB4E	bit 1,(ix+x)	4	20
$DDCB56	bit 2,(ix+x)	4	20	$DDCB5E	bit 3,(ix+x)	4	20
$DDCB66	bit 4,(ix+x)	4	20	$DDCB6E	bit 5,(ix+x)	4	20
$DDCB76	bit 6,(ix+x)	4	20	$DDCB7E	bit 7,(ix+x)	4	20
$DDCB86	res 0,(ix+x)	4	23	$DDCB8E	res 1,(ix+x)	4	23
$DDCB96	res 2,(ix+x)	4	23	$DDCB9E	res 3,(ix+x)	4	23
$DDCBA6	res 4,(ix+x)	4	23	$DDCBAE	res 5,(ix+x)	4	23
$DDCBB6	res 6,(ix+x)	4	23	$DDCBBE	res 7,(ix+x)	4	23
$DDCBC6	set 0,(ix+x)	4	23	$DDCBCE	set 1,(ix+x)	4	23
$DDCBD6	set 2,(ix+x)	4	23	$DDCBDE	set 3,(ix+x)	4	23
$DDCBE6	set 4,(ix+x)	4	23	$DDCBEE	set 5,(ix+x)	4	23
$DDCBF6	set 6,(ix+x)	4	23	$DDCBFE	set 7,(ix+x)	4	23

C.7 $FDCBxx IY Bit Operations

Table C.27: $FDCB00-$FDCBFF

Opcode	Mnemonic	Sz	T	Opcode	Mnemonic	Sz	T
$FDCB06	rlc (iy+x)	4	23	$FDCB0E	rrc (iy+x)	4	23
$FDCB16	rl (iy+x)	4	23	$FDCB1E	rr (iy+x)	4	23
$FDCB26	sla (iy+x)	4	23	$FDCB2E	sra (iy+x)	4	23
$FDCB36	sll (iy+x)	4	23	$FDCB3E	srl (iy+x)	4	23
$FDCB46	bit 0,(iy+x)	4	20	$FDCB4E	bit 1,(iy+x)	4	20
$FDCB56	bit 2,(iy+x)	4	20	$FDCB5E	bit 3,(iy+x)	4	20
$FDCB66	bit 4,(iy+x)	4	20	$FDCB6E	bit 5,(iy+x)	4	20
$FDCB76	bit 6,(iy+x)	4	20	$FDCB7E	bit 7,(iy+x)	4	20
$FDCB86	res 0,(iy+x)	4	23	$FDCB8E	res 1,(iy+x)	4	23
$FDCB96	res 2,(iy+x)	4	23	$FDCB9E	res 3,(iy+x)	4	23
$FDCBA6	res 4,(iy+x)	4	23	$FDCBAE	res 5,(iy+x)	4	23
$FDCBB6	res 6,(iy+x)	4	23	$FDCBBE	res 7,(iy+x)	4	23
$FDCBC6	set 0,(iy+x)	4	23	$FDCBCE	set 1,(iy+x)	4	23
$FDCBD6	set 2,(iy+x)	4	23	$FDCBDE	set 3,(iy+x)	4	23
$FDCBE6	set 4,(iy+x)	4	23	$FDCBEE	set 5,(iy+x)	4	23
$FDCBF6	set 6,(iy+x)	4	23	$FDCBFE	set 7,(iy+x)	4	23

Appendix D

Mnemonics to Extended Opcodes

Table D.1: aci-adc

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
aci x	$CE	2	7	adc a,l	$8D	1	4
adc a,(hl)	$8E	1	7	adc a,x	$CE	2	7
adc a,(ix+x)	$DD8E	3	19	adc a	$8F	1	4
adc a,(iy+x)	$FD8E	3	19	adc b	$88	1	4
adc a,a	$8F	1	4	adc c	$89	1	4
adc a,b	$88	1	4	adc d	$8A	1	4
adc a,c	$89	1	4	adc e	$8B	1	4
adc a,d	$8A	1	4	adc hl,bc	$ED4A	2	15
adc a,e	$8B	1	4	adc hl,de	$ED5A	2	15
adc a,h	$8C	1	4	adc hl,hl	$ED6A	2	15
adc a,ixh	$DD8C	2	8	adc hl,sp	$ED7A	2	15
adc a,ixl	$DD8D	2	8	adc h	$8C	1	4
adc a,iyh	$FD8C	2	8	adc l	$8D	1	4
adc a,iyl	$FD8D	2	8	adc m	$8E	1	7

Table D.2: add

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
add a,(hl)	$86	1	7	add de,xx *	$ED35	4	16
add a,(ix+x)	$DD86	3	19	add d	$82	1	4
add a,(iy+x)	$FD86	3	19	add e	$83	1	4
add a,b	$80	1	4	add hl,bc	$09	1	11
add a,d	$82	1	4	add hl,hl	$29	1	11
add a,e	$83	1	4	add hl,sp	$39	1	11
add a,ixh	$DD84	2	8	add h	$84	1	4
add a,ixl	$DD85	2	8	add ix,bc	$DD09	2	15
add a,iyh	$FD84	2	8	add ix,de	$DD19	2	15
add a,iyl	$FD85	2	8	add ix,ix	$DD29	2	15
add a,l	$85	1	4	add ix,sp	$DD39	2	15
add a,x	$C6	2	7	add iy,bc	$FD09	2	15
add a	$87	1	4	add iy,de	$FD19	2	15
add bc,a *	$ED33	2	8	add iy,iy	$FD29	2	15
add bc,xx *	$ED36	4	16	add iy,sp	$FD39	2	15
add b	$80	1	4	add l	$85	1	4
add c	$81	1	4	add m	$86	1	7
add de,a *	$ED32	2	8				

Table D.3: adi-ani

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
adi x	$C6	2	7	and b	$A0	1	4
ana a	$A7	1	4	and c	$A1	1	4
ana b	$A0	1	4	and d	$A2	1	4
ana c	$A1	1	4	and e	$A3	1	4
ana d	$A2	1	4	and h	$A4	1	4
ana e	$A3	1	4	and ixh	$DDA4	2	8
ana h	$A4	1	4	and ixl	$DDA5	2	8
ana l	$A5	1	4	and iyh	$FDA4	2	8
ana m	$A6	1	7	and iyl	$FDA5	2	8
and (hl)	$A6	1	7	and l	$A5	1	4
and (ix+x)	$DDA6	3	19	and x	$E6	2	7
and (iy+x)	$FDA6	3	19	ani x	$E6	2	7
and a	$A7	1	4				

Table D.4: bit 0-bit 3

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
bit 0,(hl)	$CB46	2	12	bit 2,(hl)	$CB56	2	12
bit 0,(ix+x)	$DDCB46	4	20	bit 2,(ix+x)	$DDCB56	4	20
bit 0,(iy+x)	$FDCB46	4	20	bit 2,(iy+x)	$FDCB56	4	20
bit 0,a	$CB47	2	8	bit 2,a	$CB57	2	8
bit 0,b	$CB40	2	8	bit 2,b	$CB50	2	8
bit 0,c	$CB41	2	8	bit 2,c	$CB51	2	8
bit 0,d	$CB42	2	8	bit 2,d	$CB52	2	8
bit 0,e	$CB43	2	8	bit 2,e	$CB53	2	8
bit 0,h	$CB44	2	8	bit 2,h	$CB54	2	8
bit 0,l	$CB45	2	8	bit 2,l	$CB55	2	8
bit 1,(hl)	$CB4E	2	12	bit 3,(hl)	$CB5E	2	12
bit 1,(ix+x)	$DDCB4E	4	20	bit 3,(ix+x)	$DDCB5E	4	20
bit 1,(iy+x)	$FDCB4E	4	20	bit 3,(iy+x)	$FDCB5E	4	20
bit 1,a	$CB4F	2	8	bit 3,a	$CB5F	2	8
bit 1,b	$CB48	2	8	bit 3,b	$CB58	2	8
bit 1,c	$CB49	2	8	bit 3,c	$CB59	2	8
bit 1,d	$CB4A	2	8	bit 3,d	$CB5A	2	8
bit 1,e	$CB4B	2	8	bit 3,e	$CB5B	2	8
bit 1,h	$CB4C	2	8	bit 3,h	$CB5C	2	8
bit 1,l	$CB4D	2	8	bit 3,l	$CB5D	2	8

Table D.5: bit 4-bit 7

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
bit 4,(hl)	$CB66	2	12	bit 6,(hl)	$CB76	2	12
bit 4,(ix+x)	$DDCB66	4	20	bit 6,(ix+x)	$DDCB76	4	20
bit 4,(iy+x)	$FDCB66	4	20	bit 6,(iy+x)	$FDCB76	4	20
bit 4,a	$CB67	2	8	bit 6,a	$CB77	2	8
bit 4,b	$CB60	2	8	bit 6,b	$CB70	2	8
bit 4,c	$CB61	2	8	bit 6,c	$CB71	2	8
bit 4,d	$CB62	2	8	bit 6,d	$CB72	2	8
bit 4,e	$CB63	2	8	bit 6,e	$CB73	2	8
bit 4,h	$CB64	2	8	bit 6,h	$CB74	2	8
bit 4,l	$CB65	2	8	bit 6,l	$CB75	2	8
bit 5,(hl)	$CB6E	2	12	bit 7,(hl)	$CB7E	2	12
bit 5,(ix+x)	$DDCB6E	4	20	bit 7,(ix+x)	$DDCB7E	4	20
bit 5,(iy+x)	$FDCB6E	4	20	bit 7,(iy+x)	$FDCB7E	4	20
bit 5,a	$CB6F	2	8	bit 7,a	$CB7F	2	8
bit 5,b	$CB68	2	8	bit 7,b	$CB78	2	8
bit 5,c	$CB69	2	8	bit 7,c	$CB79	2	8
bit 5,d	$CB6A	2	8	bit 7,d	$CB7A	2	8
bit 5,e	$CB6B	2	8	bit 7,e	$CB7B	2	8
bit 5,h	$CB6C	2	8	bit 7,h	$CB7C	2	8
bit 5,l	$CB6D	2	8	bit 7,l	$CB7D	2	8

Table D.6: brlc-cnz

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
brlc de,b *	$ED2C	2	8	ccf	$3F	1	4
bsla de,b *	$ED28	2	8	cm xx	$FC	3	17/10
bsra de,b *	$ED29	2	8	cma	$2F	1	4
bsrf de,b *	$ED2B	2	8	cmc	$3F	1	4
bsrl de,b *	$ED2A	2	8	cmp a	$BF	1	4
call c,xx	$DC	3	17/11	cmp b	$B8	1	4
call m,xx	$FC	3	17/10	cmp c	$B9	1	4
call nc,xx	$D4	3	17/10	cmp d	$BA	1	4
call nz,xx	$C4	3	17/10	cmp e	$BB	1	4
call p,xx	$F4	3	17/10	cmp h	$BC	1	4
call pe,xx	$EC	3	17/10	cmp l	$BD	1	4
call po,xx	$E4	3	17/10	cmp m	$BE	1	7
call xx	$CD	3	17	cnc xx	$D4	3	17/10
call z,xx	$CC	3	17/10	cnz xx	$C4	3	17/10
cc xx	$DC	3	17/11				

Table D.7: cp-dcr

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
cp (hl)	$BE	1	7	cpir	$EDB1	2	21/16
cp (ix+x)	$DDBE	2	19	cpi	$EDA1	2	16
cp (iy+x)	$FDBE	2	19	cpl	$2F	1	4
cp a	$BF	1	4	cpo xx	$E4	3	17/10
cp b	$B8	1	4	cz xx	$CC	3	17/10
cp c	$B9	1	4	daa	$27	1	4
cp d	$BA	1	4	dad b	$09	1	11
cp e	$BB	1	4	dad d	$19	1	11
cp h	$BC	1	4	dad h	$29	1	11
cp ixh	$DDBC	2	8	dad sp	$39	1	11
cp ixl	$DDBD	2	8	dcr a	$3D	1	4
cp iyh	$FDBC	2	8	dcr b	$05	1	4
cp iyl	$FDBD	2	8	dcr c	$0D	1	4
cp l	$BD	1	4	dcr d	$15	1	4
cp xx	$F4	3	17/10	dcr e	$1D	1	4
cp x	$FE	2	7	dcr h	$25	1	4
cpdr	$EDB9	2	21/16	dcr l	$2D	1	4
cpd	$EDA9	2	16	dcr m	$35	1	11
cpe	$EC	3	17/10				

Table D.8: dcx-im

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
dcx b	$0B	1	6	dec iyl	$FD2D	2	8
dcx d	$1B	1	6	dec iy	$FD2B	2	10
dcx h	$2B	1	6	dec l	$2D	1	4
dcx sp	$3B	1	6	dec sp	$3B	1	6
dec (hl)	$35	1	11	di	$F3	1	4
dec (ix+x)	$DD35	3	23	djnz x	$10	2	13/8
dec (iy+x)	$FD35	3	23	ei	$FB	1	4
dec a	$3D	1	4	ex (sp),hl	$E3	1	19
dec bc	$0B	1	6	ex (sp),ix	$DDE3	2	23
dec b	$05	1	4	ex (sp),iy	$FDE3	2	23
dec c	$0D	1	4	ex af,af'	$08	1	4
dec de	$1B	1	6	ex de,hl	$EB	1	4
dec d	$15	1	4	exx	$D9	1	4
dec e	$1D	1	4	halt	$76	1	4+
dec hl	$2B	1	6	icr c	$0C	1	4
dec h	$25	1	4	icr e	$1C	1	4
dec ixh	$DD25	2	8	im 0	$ED46	2	8
dec ixl	$DD2D	2	8	im 1	$ED56	2	8
dec ix	$DD2B	2	10	im 2	$ED5E	2	8
dec iyh	$FD25	2	8				

Table D.9: in-inx

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
in a,(c)	$ED78	2	12	inc ixl	$DD2C	2	8
in a,(x)	$DB	2	11	inc ix	$DD23	2	10
in b,(c)	$ED40	2	12	inc iyh	$FD24	2	8
in c,(c)	$ED48	2	12	inc iyl	$FD2C	2	8
in d,(c)	$ED50	2	12	inc iy	$FD23	2	10
in e,(c)	$ED58	2	12	inc l	$2C	1	4
in f,(c)	$ED70	2	12	inc sp	$33	1	6
in h,(c)	$ED60	2	12	indr	$EDBA	2	21/16
in l,(c)	$ED68	2	12	ind	$EDAA	2	16
in x	$DB	2	11	inir	$EDB2	2	21/16
inc (hl)	$34	1	11	ini	$EDA2	2	16
inc (ix+x)	$DD34	3	23	inr a	$3C	1	4
inc (iy+x)	$FD34	3	23	inr b	$04	1	4
inc a	$3C	1	4	inr d	$14	1	4
inc bc	$03	1	6	inr h	$24	1	4
inc b	$04	1	4	inr l	$2C	1	4
inc c	$0C	1	4	inr m	$34	1	11
inc de	$13	1	6	inx b	$03	1	6
inc d	$14	1	4	inx d	$13	1	6
inc e	$1C	1	4	inx h	$23	1	6
inc hl	$23	1	6	inx sp	$33	1	6
inc h	$24	1	4				

Table D.10: jc-jz

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
jc xx	$DA	3	10	jp pe,xx	$EA	3	10
jm xx	$FA	3	10	jp po,xx	$E2	3	10
jmp xx	$C3	3	10	jp xx	$C3	3	10
jnc xx	$D2	3	10	jp xx	$F2	3	10
jnz xx	$C2	3	10	jp z,xx	$CA	3	10
jp (c) *	$ED98	2	13	jpe xx	$EA	3	10
jp (hl)	$E9	1	4	jpo xx	$E2	3	10
jp (ix)	$DDE9	3	8	jr c,x	$38	2	12/7
jp (iy)	$FDE9	3	8	jr nc,x	$30	2	12/7
jp c,xx	$DA	3	10	jr nz,x	$20	2	12/7
jp m,xx	$FA	3	10	jr x	$18	2	12
jp nc,xx	$D2	3	10	jr z,x	$28	2	12/7
jp nz,xx	$C2	3	10	jz xx	$CA	3	10
jp p,xx	$F2	3	10				

Table D.11: ld (bc),a-ld (iy+x),x

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
ld (bc),a	$02	1	7	ld (ix+x),e	$DD73	3	19
ld (de),a	$12	1	7	ld (ix+x),h	$DD74	3	19
ld (hl),a	$77	1	7	ld (ix+x),l	$DD75	3	19
ld (hl),b	$70	1	4	ld (ix+x),x	$DD36	5	19
ld (hl),c	$71	1	4	ld (iy+x),a	$FD77	3	19
ld (hl),d	$72	1	4	ld (iy+x),b	$FD70	3	19
ld (hl),e	$73	1	4	ld (iy+x),c	$FD71	3	19
ld (hl),h	$74	1	4	ld (iy+x),d	$FD72	3	19
ld (hl),l	$75	1	4	ld (iy+x),e	$FD73	3	19
ld (hl),x	$36	2	10	ld (iy+x),h	$FD74	3	19
ld (ix+x),a	$DD77	3	19	ld (iy+x),l	$FD75	3	19
ld (ix+x),b	$DD70	3	19	ld (iy+x),x	$FD36	5	19
ld (ix+x),c	$DD71	3	19				

Table D.12: ld (xx),a-ld a,x

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
ld (xx),a	$32	3	13	ld a,b	$78	1	4
ld (xx),bc	$ED43	4	20	ld a,c	$79	1	4
ld (xx),de	$ED53	4	20	ld a,d	$7A	1	4
ld (xx),hl	$22	3	16	ld a,e	$7B	1	4
ld (xx),ix	$DD22	4	20	ld a,ixh	$DD7C	2	8
ld (xx),iy	$FD22	4	20	ld a,ixl	$DD7D	2	8
ld (xx),sp	$ED73	4	20	ld a,iyh	$FD7C	2	8
ld a,(bc)	$0A	1	7	ld a,iyl	$FD7D	2	8
ld a,(de)	$1A	1	7	ld a,i	$ED57	2	9
ld a,(hl)	$7E	1	7	ld a,l	$7D	1	4
ld a,(ix+x)	$DD7E	3	19	ld a,r	$ED5F	2	9
ld a,(iy+x)	$FD7E	3	19	ld a,x	$3E	2	7
ld a,(xx)	$3A	3	13				

Table D.13: ld b,(hl)-ld c,x

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
ld b,(hl)	$46	1	7	ld c,(hl)	$4E	1	7
ld b,(ix+x)	$DD46	2	19	ld c,(ix+x)	$DD4E	3	19
ld b,(iy+x)	$FD46	2	19	ld c,(iy+x)	$FD4E	3	19
ld b,a	$47	1	4	ld c,a	$4F	1	4
ld b,b	$40	1	4	ld c,b	$48	1	4
ld b,c	$41	1	4	ld c,c	$49	1	4
ld b,d	$42	1	4	ld c,d	$4A	1	4
ld b,e	$43	1	4	ld c,e	$4B	1	4
ld b,h	$44	1	4	ld c,h	$4C	1	4
ld b,ixh	$DD44	2	8	ld c,ixh	$DD4C	2	8
ld b,ixl	$DD45	2	8	ld c,ixl	$DD4D	2	8
ld b,iyh	$FD44	2	8	ld c,iyh	$FD4C	2	8
ld b,iyl	$FD45	2	8	ld c,iyl	$FD4D	2	8
ld b,l	$45	1	4	ld c,l	$4D	1	4
ld b,x	$06	2	7	ld c,x	$0E	2	7
ld bc,(xx)	$ED4B	4	20				

Table D.14: ld d,(hl)-ld e,x

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
ld d,(hl)	$56	1	7	ld e,(hl)	$5E	1	7
ld d,(ix+x)	$DD56	3	19	ld e,(ix+x)	$DD5E	3	19
ld d,(iy+x)	$FD56	3	19	ld e,(iy+x)	$FD5E	3	19
ld d,a	$57	1	4	ld e,a	$5F	1	4
ld d,b	$50	1	4	ld e,b	$58	1	4
ld d,c	$51	1	4	ld e,c	$59	1	4
ld d,d	$52	1	4	ld e,d	$5A	1	4
ld d,e	$53	1	4	ld e,e	$5B	1	4
ld d,h	$54	1	4	ld e,h	$5C	1	4
ld d,ixh	$DD54	2	8	ld e,ixh	$DD5C	2	8
ld d,ixl	$DD55	2	8	ld e,ixl	$DD5D	2	8
ld d,iyh	$FD54	2	8	ld e,iyh	$FD5C	2	8
ld d,iyl	$FD55	2	8	ld e,iyl	$FD5D	2	8
ld d,l	$55	1	4	ld e,l	$5D	1	4
ld d,x	$16	2	7	ld e,x	$1E	2	7
ld de,(xx)	$ED5B	4	20				

Table D.15: ld h,(hl)-ld ixl,x

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
ld h,(hl)	$66	1	7	ld ixh,a	$DD67	2	8
ld h,(ix+x)	$DD65	3	19	ld ixh,b	$DD60	2	8
ld h,(iy+x)	$FD65	3	19	ld ixh,c	$DD61	2	8
ld h,a	$67	1	4	ld ixh,d	$DD62	2	8
ld h,b	$60	1	4	ld ixh,e	$DD63	2	8
ld h,c	$61	1	4	ld ixh,ixh	$DD64	2	8
ld h,d	$62	1	4	ld ixh,ixl	$DD65	2	8
ld h,e	$63	1	4	ld ixh,x	$DD26	3	11
ld h,h	$64	1	4	ld ixl,a	$DD6F	2	8
ld h,l	$65	1	4	ld ixl,b	$DD68	2	8
ld h,x	$26	2	7	ld ixl,c	$DD69	2	8
ld hl,(xx)	$2A	3	16	ld ixl,d	$DD6A	2	8
ld hl,(xx)	$ED6B	4	20	ld ixl,e	$DD6B	2	8
ld hl,xx	$21	3	10	ld ixl,ixh	$DD6C	2	2
ld i,a	$ED47	2	9	ld ixl,ixl	$DD6D	2	2
ld ix,(xx)	$DD2A	4	20	ld ixl,x	$DD2E	4	11
ld ix,xx	$DD21	4	14				

Table D.16: ld iyh,a-ld sp,xx

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
ld iyh,a	$FD67	2	8	ld l,(ix+x)	$DD6E	3	19
ld iyh,b	$FD60	2	8	ld l,(iy+x)	$FD6E	3	19
ld iyh,c	$FD61	2	8	ld l,a	$6F	1	4
ld iyh,d	$FD62	2	8	ld l,b	$68	1	4
ld iyh,e	$FD63	2	8	ld l,c	$69	1	4
ld iyh,iyh	$FD64	2	8	ld l,d	$6A	1	4
ld iyh,iyl	$FD65	2	8	ld l,e	$6B	1	4
ld iyh,x	$FD26	3	11	ld l,h	$6C	1	4
ld iyl,a	$FD6F	2	8	ld l,l	$6D	1	4
ld iyl,b	$FD68	2	8	ld l,x	$2E	2	7
ld iyl,c	$FD69	2	8	ld r,a	$ED4F	2	9
ld iyl,d	$FD6A	2	8	ld sp,(xx)	$ED7B	4	20
ld iyl,e	$FD6B	2	8	ld sp,hl	$F9	1	6
ld iyl,iyh	$FD6C	2	2	ld sp,ix	$DDF9	2	10
ld iyl,iyl	$FD6D	2	2	ld sp,iy	$FDF9	2	10
ld iyl,x	$FD2E	4	11	ld sp,xx	$31	3	10
ld l,(hl)	$6E	1	7				

Table D.17: lda-mirror

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
lda xx	$3A	3	13	ldi	$EDA0	2	16
ldax b	$0A	1	7	ldpirx *	$EDB7	2	21/16
ldax d	$1A	1	7	ldws *	$EDA5	2	14
lddrx *	$EDBC	2	21/16	lhld xx	$2A	3	16
lddr	$EDB8	2	21/16	lxi b,xx	$01	3	10
lddx *	$EDAC	2	16	lxi d,xx	$11	3	10
ldd	$EDA8	2	16	lxi h,xx	$21	3	10
ldirx *	$EDB4	2	21/16	lxi sp,xx	$31	3	10
ldir	$EDB0	2	21/16	mirror a *	$ED24	2	8
ldix *	$EDA4	2	16				

Table D.18: mov a,a-mov d,m

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
mov a,a	$7F	1	4	mov c,b	$48	1	4
mov a,b	$78	1	4	mov c,c	$49	1	4
mov a,c	$79	1	4	mov c,d	$4A	1	4
mov a,d	$7A	1	4	mov c,e	$4B	1	4
mov a,e	$7B	1	4	mov c,h	$4C	1	4
mov a,h	$7C	1	4	mov c,l	$4D	1	4
mov a,l	$7D	1	4	mov c,m	$4E	1	7
mov a,m	$7E	1	7	mov d,a	$57	1	4
mov b,a	$47	1	4	mov d,b	$50	1	4
mov b,b	$40	1	4	mov d,c	$51	1	4
mov b,c	$41	1	4	mov d,d	$52	1	4
mov b,d	$42	1	4	mov d,e	$53	1	4
mov b,e	$43	1	4	mov d,h	$54	1	4
mov b,h	$44	1	4	mov d,l	$55	1	4
mov b,l	$45	1	4	mov d,m	$56	1	7
mov b,m	$46	1	7				

Table D.19: mov e,a-mov m,l

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
mov e,a	$5F	1	4	mov l,a	$6F	1	4
mov e,b	$58	1	4	mov l,b	$68	1	4
mov e,c	$59	1	4	mov l,c	$69	1	4
mov e,d	$5A	1	4	mov l,d	$6A	1	4
mov e,e	$5B	1	4	mov l,e	$6B	1	4
mov e,h	$5C	1	4	mov l,h	$6C	1	4
mov e,l	$5D	1	4	mov l,l	$6D	1	4
mov e,m	$5E	1	7	mov l,m	$6E	1	7
mov h,a	$67	1	4	mov m,a	$77	1	7
mov h,b	$60	1	4	mov m,b	$70	1	4
mov h,c	$61	1	4	mov m,c	$71	1	4
mov h,d	$62	1	4	mov m,d	$72	1	4
mov h,e	$63	1	4	mov m,e	$73	1	4
mov h,h	$64	1	4	mov m,h	$74	1	4
mov h,l	$65	1	4	mov m,l	$75	1	4
mov h,m	$66	1	7				

Table D.20: mul-otir

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
mul d,e *	$ED30	2	8	or e	$B3	1	4
mvi a,x	$3E	2	7	or h	$B4	1	4
mvi b,x	$06	2	7	or ixh	$DDB4	2	8
mvi c,x	$0E	2	7	or ixl	$DDB5	2	8
mvi d,x	$16	2	7	or iyh	$FDB4	2	8
mvi e,x	$1E	2	7	or iyl	$FDB5	2	8
mvi h,x	$26	2	7	or l	$B5	1	4
mvi l,x	$2E	2	7	or x	$F6	2	7
mvi m,x	$36	2	10	ora a	$B7	1	4
neg	$ED44	2	8	ora b	$B0	1	4
nextreg r,a *	$ED92	3	17	ora c	$B1	1	4
nextreg r,v *	$ED91	4	20	ora d	$B2	1	4
nop	$00	1	4	ora e	$B3	1	4
or (hl)	$B6	1	7	ora h	$B4	1	4
or (ix+x)	$DDB6	3	19	ora l	$B5	1	4
or (iy+x)	$FDB6	3	19	ora m	$B6	1	7
or a	$B7	1	4	ori x	$F6	2	7
or b	$B0	1	4	otdr	$EDBB	2	12/16
or c	$B1	1	4	otir	$EDB3	2	21/16
or d	$B2	1	4				

Table D.21: out-rc

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
out (c),a	$ED79	2	12	pop hl	$E1	1	10
out (c),b	$ED41	2	12	pop h	$E1	1	10
out (c),c	$ED49	2	12	pop ix	$DDE1	2	14
out (c),d	$ED51	2	12	pop iy	$FDE1	2	14
out (c),e	$ED59	2	12	pop psw	$F1	1	10
out (c),f	$ED71	2	12	push af	$F5	1	11
out (c),h	$ED61	2	12	push bc	$C5	1	11
out (c),l	$ED69	2	12	push b	$C5	1	11
out (x),a	$D3	2	11	push de	$D5	1	11
out x	$D3	2	11	push d	$D5	1	11
outd	$EDAB	2	16	push hl	$E5	1	11
outinb *	$ED90	2	16	push h	$E5	1	11
outi	$EDA3	2	16	push ix	$DDE5	2	15
pchl	$E9	1	4	push iy	$FDE5	2	15
pixelad *	$ED94	2	8	push psw	$F5	1	11
pixeldn *	$ED93	2	8	push xx	$ED8A	4	*
pop af	$F1	1	10	ral	$17	1	4
pop bc	$C1	1	10	rar	$1F	1	4
pop b	$C1	1	10	rc	$D8	1	11/5
pop de	$D1	1	10				

Table D.22: res 0-res 3

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
res 0,(hl)	$CB86	2	15	res 2,(ix+x)	$DDCB96	4	23
res 0,(ix+x)	$DDCB86	4	23	res 2,(iy+x)	$FDCB96	4	23
res 0,(iy+x)	$FDCB86	4	23	res 2,a	$CB97	2	8
res 0,a	$CB87	2	8	res 2,b	$CB90	2	8
res 0,b	$CB80	2	8	res 2,c	$CB91	2	8
res 0,c	$CB81	2	8	res 2,d	$CB92	2	8
res 0,d	$CB82	2	8	res 2,e	$CB93	2	8
res 0,e	$CB83	2	8	res 2,h	$CB94	2	8
res 0,h	$CB84	2	8	res 2,l	$CB95	2	8
res 0,l	$CB85	2	8	res 3,(hl)	$CB9E	2	15
res 1,(hl)	$CB8E	2	15	res 3,(ix+x)	$DDCB9E	4	23
res 1,(ix+x)	$DDCB8E	4	23	res 3,(iy+x)	$FDCB9E	4	23
res 1,(iy+x)	$FDCB8E	4	23	res 3,a	$CB9F	2	8
res 1,a	$CB8F	2	8	res 3,b	$CB98	2	8
res 1,b	$CB88	2	8	res 3,c	$CB99	2	8
res 1,c	$CB89	2	8	res 3,d	$CB9A	2	8
res 1,d	$CB8A	2	8	res 3,e	$CB9B	2	8
res 1,e	$CB8B	2	8	res 3,h	$CB9C	2	8
res 1,h	$CB8C	2	8	res 3,l	$CB9D	2	8
res 1,l	$CB8D	2	8				

Table D.23: res 4-res 7

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
res 4,(hl)	$CBA6	2	15	res 6,(ix+x)	$DDCBB6	4	23
res 4,(ix+x)	$DDCBA6	4	23	res 6,(iy+x)	$FDCBB6	4	23
res 4,(iy+x)	$FDCBA6	4	23	res 6,a	$CBB7	2	8
res 4,a	$CBA7	2	8	res 6,b	$CBB0	2	8
res 4,b	$CBA0	2	8	res 6,c	$CBB1	2	8
res 4,c	$CBA1	2	8	res 6,d	$CBB2	2	8
res 4,d	$CBA2	2	8	res 6,e	$CBB3	2	8
res 4,e	$CBA3	2	8	res 6,h	$CBB4	2	8
res 4,h	$CBA4	2	8	res 6,l	$CBB5	2	8
res 4,l	$CBA5	2	8	res 7,(hl)	$CBBE	2	15
res 5,(hl)	$CBAE	2	15	res 7,(ix+x)	$DDCBBE	4	23
res 5,(ix+x)	$DDCBAE	4	23	res 7,(iy+x)	$FDCBBE	4	23
res 5,(iy+x)	$FDCBAE	4	23	res 7,a	$CBBF	2	8
res 5,a	$CBAF	2	8	res 7,b	$CBB8	2	8
res 5,b	$CBA8	2	8	res 7,c	$CBB9	2	8
res 5,c	$CBA9	2	8	res 7,d	$CBBA	2	8
res 5,d	$CBAA	2	8	res 7,e	$CBBB	2	8
res 5,e	$CBAB	2	8	res 7,h	$CBBC	2	8
res 5,h	$CBAC	2	8	res 7,l	$CBBD	2	8
res 5,l	$CBAD	2	8				

Table D.24: ret-rp

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
ret c	$D8	1	11/5	rla	$17	1	4
ret m	$F8	1	11/5	rlc (hl)	$CB06	2	15
ret nc	$D0	1	11/5	rlc (ix+x)	$DDCB06	4	23
ret nz	$C0	1	11/5	rlc (iy+x)	$FDCB06	4	23
ret pe	$E8	1	11/5	rlc a	$CB07	2	8
ret po	$E0	1	11/5	rlc b	$CB00	2	8
ret p	$F0	1	11/5	rlc c	$CB01	2	8
ret z	$C8	1	11/5	rlc d	$CB02	2	8
reti	$ED4D	2	14	rlc e	$CB03	2	8
retn	$ED45	2	14	rlc h	$CB04	2	8
ret	$C9	1	10	rlc l	$CB05	2	8
rl (hl)	$CB16	2	15	rlca	$07	1	4
rl (ix+x)	$DDCB16	4	23	rlc	$07	1	4
rl (iy+x)	$FDCB16	4	23	rld	$ED6F	2	18
rl a	$CB17	2	8	rm	$F8	1	11/5
rl b	$CB10	2	8	rnc	$D0	1	11/5
rl c	$CB11	2	8	rnz	$C0	1	11/5
rl d	$CB12	2	8	rpe	$E8	1	11/5
rl e	$CB13	2	8	rpo	$E0	1	11/5
rl h	$CB14	2	8	rp	$F0	1	11/5
rl l	$CB15	2	8				

Table D.25: rr-rz

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
rr (hl)	$CB1E	2	15	rrca	$0F	1	4
rr (ix+x)	$DDCB1E	4	23	rrc	$0F	1	4
rr (iy+x)	$FDCB1E	4	23	rrd	$ED67	2	18
rr a	$CB1F	2	8	rst 00h	$C7	1	11
rr b	$CB18	2	8	rst 08h	$CF	1	11
rr c	$CB19	2	8	rst 0	$C7	1	11
rr d	$CB1A	2	8	rst 10h	$D7	1	11
rr e	$CB1B	2	8	rst 18h	$DF	1	11
rr h	$CB1C	2	8	rst 1	$CF	1	11
rr l	$CB1D	2	8	rst 20h	$E7	1	11
rra	$1F	1	4	rst 28h	$EF	1	11
rrc (hl)	$CB0E	2	15	rst 2	$D7	1	11
rrc (ix+x)	$DDCB0E	4	23	rst 30h	$F7	1	11
rrc (iy+x)	$FDCB0E	4	23	rst 38h	$FF	1	11
rrc a	$CB0F	2	8	rst 3	$DF	1	11
rrc b	$CB08	2	8	rst 4	$E7	1	11
rrc c	$CB09	2	8	rst 5	$EF	1	11
rrc d	$CB0A	2	8	rst 6	$F7	1	11
rrc e	$CB0B	2	8	rst 7	$FF	1	11
rrc h	$CB0C	2	8	rz	$C8	1	11/5
rrc l	$CB0D	2	8				

Table D.26: sbb-scf

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
sbb a	$9F	1	4	sbc a,e	$9B	1	4
sbb b	$98	1	4	sbc a,h	$9C	1	4
sbb c	$99	1	4	sbc a,ixh	$DD9C	2	8
sbb d	$9A	1	4	sbc a,ixl	$DD9D	2	8
sbb e	$9B	1	4	sbc a,iyh	$FD9C	2	8
sbb h	$9C	1	4	sbc a,iyl	$FD9D	2	8
sbb l	$9D	1	4	sbc a,l	$9D	1	4
sbb m	$9E	1	7	sbc a,x	$DE	2	7
sbc a,(hl)	$9E	1	7	sbc hl,bc	$ED42	2	15
sbc a,(ix+x)	$DD9E	3	1	sbc hl,de	$ED52	2	15
sbc a,(iy+x)	$FD9E	3	1	sbc hl,hl	$ED62	2	15
sbc a,a	$9F	1	4	sbc hl,sp	$ED72	2	15
sbc a,b	$98	1	4	sbi x	$DE	2	7
sbc a,c	$99	1	4	scf	$37	1	4
sbc a,d	$9A	1	4				

Table D.27: set 0-set 3

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
set 0,(hl)	$CBC6	2	15	set 2,(ix+x)	$DDCBD6	4	23
set 0,(ix+x)	$DDCBC6	4	23	set 2,(iy+x)	$FDCBD6	4	23
set 0,(iy+x)	$FDCBC6	4	23	set 2,a	$CBD7	2	8
set 0,a	$CBC7	2	8	set 2,b	$CBD0	2	8
set 0,b	$CBC0	2	8	set 2,c	$CBD1	2	8
set 0,c	$CBC1	2	8	set 2,d	$CBD2	2	8
set 0,d	$CBC2	2	8	set 2,e	$CBD3	2	8
set 0,e	$CBC3	2	8	set 2,h	$CBD4	2	8
set 0,h	$CBC4	2	8	set 2,l	$CBD5	2	8
set 0,l	$CBC5	2	8	set 3,(hl)	$CBDE	2	15
set 1,(hl)	$CBCE	2	15	set 3,(ix+x)	$DDCBDE	4	23
set 1,(ix+x)	$DDCBCE	4	23	set 3,(iy+x)	$FDCBDE	4	23
set 1,(iy+x)	$FDCBCE	4	23	set 3,a	$CBDF	2	8
set 1,a	$CBCF	2	8	set 3,b	$CBD8	2	8
set 1,b	$CBC8	2	8	set 3,c	$CBD9	2	8
set 1,c	$CBC9	2	8	set 3,d	$CBDA	2	8
set 1,d	$CBCA	2	8	set 3,e	$CBDB	2	8
set 1,e	$CBCB	2	8	set 3,h	$CBDC	2	8
set 1,h	$CBCC	2	8	set 3,l	$CBDD	2	8
set 1,l	$CBCD	2	8				

Table D.28: set 4-set 7

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
set 4,(hl)	$CBE6	2	15	set 6,(ix+x)	$DDCBF6	4	23
set 4,(ix+x)	$DDCBE6	4	23	set 6,(iy+x)	$FDCBF6	4	23
set 4,(iy+x)	$FDCBE6	4	23	set 6,a	$CBF7	2	8
set 4,a	$CBE7	2	8	set 6,b	$CBF0	2	8
set 4,b	$CBE0	2	8	set 6,c	$CBF1	2	8
set 4,c	$CBE1	2	8	set 6,d	$CBF2	2	8
set 4,d	$CBE2	2	8	set 6,e	$CBF3	2	8
set 4,e	$CBE3	2	8	set 6,h	$CBF4	2	8
set 4,h	$CBE4	2	8	set 6,l	$CBF5	2	8
set 4,l	$CBE5	2	8	set 7,(hl)	$CBFE	2	15
set 5,(hl)	$CBEE	2	15	set 7,(ix+x)	$DDCBFE	4	23
set 5,(ix+x)	$DDCBEE	4	23	set 7,(iy+x)	$FDCBFE	4	23
set 5,(iy+x)	$FDCBEE	4	23	set 7,a	$CBFF	2	8
set 5,a	$CBEF	2	8	set 7,b	$CBF8	2	8
set 5,b	$CBE8	2	8	set 7,c	$CBF9	2	8
set 5,c	$CBE9	2	8	set 7,d	$CBFA	2	8
set 5,d	$CBEA	2	8	set 7,e	$CBFB	2	8
set 5,e	$CBEB	2	8	set 7,h	$CBFC	2	8
set 5,h	$CBEC	2	8	set 7,l	$CBFD	2	8
set 5,l	$CBED	2	8				

Table D.29: setae-stc

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
setae *	$ED95	2	8	sra (ix+x)	$DDCB2E	4	23
shld xx	$22	3	16	sra (iy+x)	$FDCB2E	4	23
sla (hl)	$CB26	2	15	sra a	$CB2F	2	8
sla (ix+x)	$DDCB26	4	23	sra b	$CB28	2	8
sla (iy+x)	$FDCB26	4	23	sra c	$CB29	2	8
sla a	$CB27	2	8	sra d	$CB2A	2	8
sla b	$CB20	2	8	sra e	$CB2B	2	8
sla c	$CB21	2	8	sra h	$CB2C	2	8
sla d	$CB22	2	8	sra l	$CB2D	2	8
sla e	$CB23	2	8	srl (hl)	$CB3E	2	15
sla h	$CB24	2	8	srl (ix+x)	$DDCB3E	4	23
sla l	$CB25	2	8	srl (iy+x)	$FDCB3E	4	23
sll (hl)	$CB36	2	15	srl a	$CB3F	2	8
sll (ix+x)	$DDCB36	4	23	srl b	$CB38	2	8
sll (iy+x)	$FDCB36	4	23	srl c	$CB39	2	8
sll a	$CB37	2	8	srl d	$CB3A	2	8
sll b	$CB30	2	8	srl e	$CB3B	2	8
sll c	$CB31	2	8	srl h	$CB3C	2	8
sll d	$CB32	2	8	srl l	$CB3D	2	8
sll e	$CB33	2	8	sta xx	$32	3	13
sll h	$CB34	2	8	stax b	$02	1	7
sll l	$CB35	2	8	stax d	$12	1	7
sphl	$F9	1	6	stc	$37	1	4
sra (hl)	$CB2E	2	15				

Table D.30: sub-xthl

Mnemonic	Opcode	Sz	T	Mnemonic	Opcode	Sz	T
sub (hl)	$96	1	7	xor a	$AF	1	4
sub (ix+x)	$DD96	3	19	xor b	$A8	1	4
sub (iy+x)	$FD96	3	19	xor c	$A9	1	4
sub a	$97	1	4	xor d	$AA	1	4
sub b	$90	1	4	xor e	$AB	1	4
sub c	$91	1	4	xor h	$AC	1	4
sub d	$92	1	4	xor ixh	$DDAC	2	8
sub e	$93	1	4	xor ixl	$DDAD	2	8
sub h	$94	1	4	xor iyh	$FDAC	2	8
sub ixh	$DD94	2	8	xor iyl	$FDAD	2	8
sub ixl	$DD95	2	8	xor l	$AD	1	4
sub iyh	$FD94	2	8	xor x	$EE	2	7
sub iyl	$FD95	2	8	xra a	$AF	1	4
sub l	$95	1	4	xra b	$A8	1	4
sub m	$96	1	7	xra c	$A9	1	4
sub x	$D6	2	7	xra d	$AA	1	4
sui x	$D6	2	7	xra e	$AB	1	4
swapinb *	$ED23	2	8	xra h	$AC	1	4
test x *	$ED27	3	11	xra l	$AD	1	4
xchg	$EB	1	4	xra m	$AE	1	7
xor (hl)	$AE	1	7	xri x	$EE	2	7
xor (ix+x)	$DDAE	3	19	xthl	$E3	1	19
xor (iy+x)	$FDAE	3	19				

Appendix E

File Formats

Appendix F

Call Tables

F.1 BDOS Call Table

F.2 BIOS Call Table

F.3 ESXDOS Call Table

List of Figures

List of Tables

Index