AMOEBA Hardware Manual

Last updated: 05-10-2011 for AMOEBA v10B

AMOEBA is the default FPGA architecture for the EZ80P computer by Phil Ruston.
It offers the following features (amongst others): 256 colour VGA output in up to 640x480 resolution, bitmap, tile and text mode displays, hardware sprites, 8 channel 8-bit sound.

AMOEBA Memory Map:

000000h - 07FFFFh	System RAM	"sysram_addr"
800000h - 87FFFFh	Video RAM A (Background)	"vram_a_addr"
c00000h - c7FFFFh	Video RAM B (Sprites/Audio)	"vram_b_addr"
FF0000h - FF1FFFh	Video Registers, detail below:	
FF0000h - FF07FFh	Colour Palette	"hw_palette"
FF0800h - FF0FFFh	Sprite registers	"hw_sprite_registers"
FF1000h - FF1030h	Display Settings	"hw_video_parameters"
FF1400h - FF14FFh	Sound control registers	"hw_audio_registers"
FF1800h - FF1FFFh	General video control	"hw_video_settings"

AMOEBA Port Map:

0000h-000Ah: Control of peripherals within AMOEBA
0080h-00FFh: Control of peripherals within the eZ80 CPU

System / Peripheral Control

Controlling systems such as the keyboard and mouse ports, SD card and EEPROM communication is achieved via eZ80 port addresses. Some EZ80P systems such as the serial and joystick ports use the eZ80's built in peripheral controllers, whilst others are implemented in the AMOEBA architecture.

AMOEBA-based ports:

The following ports control the AMOEBA-based systems. Note the same port number can have different purposes depending on whether it is read from or written to.

Port: 0000h - Read/Write - Label: "port_pic_data"

A data buffer which passes bytes to and from the EZ80P's onboard EEPROM, via a PIC microcontroller (see the section on PIC communication). Control of data flow is achieved by reading bit 0 ("byte received") and bit 4 ("busy - sending") from Port 1.

Bits: 0:7 - Byte to write or read data from the PIC/EEPROM sub-system

Port 0001h - Read - Label: "port_hw_flags"

Provides information about the status of various hardware systems. bits 0:3 are concerned with PIC/EEPROM communication (see the relevant section). Bits 4, 5 and 6 are self explanatory, bit 7 allows the 16 bit AMOEBA version string to be read, the bit that appears here is selected by port 8.

Bits: 0 - An EEPROM byte has been received when set
1 - PIC comms clock line status
2 - PIC (output) serializer is busy when set
3 - Background video scanline build is complete when set
4 - SD card output serializer is busy when set
5 - VRT Latch. The last scan line of display sets this flip-flop.
6 - Audio channel loop status
7 - H/W config ID bit

Port 0001h - Write - Label: "port_pic_ctrl"

Involved in the PIC/EEPROM communication system (see relevant section)

Bits: 0 - Force pic comms data line high when set (for databurst clocking)
1:7 - unused at present

Port 0002h - Read- Label: "port_keyboard_data"

Returns a byte from the AMOEBA keyboard buffer queue. When the queue is empty, bit 4 of PORT 7 is cleared.

Bits 0:7 - scan code byte

Port 0002h - Write - Label: "port_sdc_ctrl"

Controls the SD card communication. It is a set/reset type register, bit 7 controls whether bit positions in 6-0 written with ones are set or cleared (Other bits are unchanged).

Bit: 0: SD card power (active high)
1: SD card /CS (active low)
2: SD card speed (full speed when set)
7: Set / Reset control for bits 0:2

Port 0003h - Read/Write - Label "port_sdc_data"

SD card data buffer. Bytes written here are immediately transmitted to the SD card's "D_in" pin (bit 4 of port 1 is set during transmission) and bytes transmitted from the SD card's "D_out" pin appear here (following 8 clocks being sent to the SD card, IE: writing FFh to this port).

Bits 0:7 - byte from SD card

Port 0004h - Write - Label "port_memory_paging"

Controls memory paging.

Bit 0 - ROM paging. The FPGA-based ROM at 0-7FFh is replaced by RAM when set.

Port 0005h- Read - Label "port_irq_flags"

Contains flags to determine the source of an interrupt

Bits 2 - Audio IRQ flag
5 - Raster line match flip-flop status (NMI)
6 - IO1 pin (freezer button) flip-flop status (NMI)

Port 0005h - Write - Label "port_irq_ctrl"

Controls which interrupt sources actually cause interrupts. It is a set/reset type register, IE: bit 7 controls whether bits0:6 written with ones are set or cleared (other bits are unchanged).

Bit 0 - INT - Allow Keyboard interrupt when set
1 - INT - Allow mouse interrupt when set
2 - INT - Allow audio interrupt when set
5 - NMI - Raster line interrupt enabled when set
6 - NMI - Spare IO1 (NMI button) enabled when set
7 - Set / Reset control for bits 0-6

Port 0006h - Read- Label: "port_mouse_data"

Returns a byte from the AMOEBA mouse buffer queue. When the queue is empty, bit 5 in PORT 7 is cleared.

Bits 0:7 - mouse packet byte

Port 0007h - Write - Label: "port_ps2_ctrl"

Allows manual (output) control of the PS/2 control lines for the mouse and keyboard.

Bits: 0 - Keyboard clock output (set to 1 to pull signal low)
1 - Keyboard data output ("")
2 - Mouse clock output ("")
3 - Mouse data output ("")

Port 0007h - Read - Label: "port_ps2_ctrl"

Contains keyboard and mouse buffer status bits and allows manual reading of the PS/2 clock and data lines.

Bits: 0 - Read Keyboard clock state (controlled by device or by writing to this port)
1 - Read Keyboard data current state ("")
2 - Read Mouse clock current state ("")
3 - Read Mouse data current state ("")
4 - Keyboard buffer status
5 - Mouse buffer status

Port 0008h - Write - Label "port_selector"

Generic selector. The value written here is used for selecting different things:

- Bit 0-15 from the hardware version ID string which appears at port 0, bit 7.
- Audio Channel 0-7 buffer status appearing in port 0, bit 6

Bits: 0:3 - selection value

Port 0009h - Write - Label "port_clear_flags"

Bits written with ones clear the relevant latch, other latches are unchanged.

Bits 0 - Clears the VRT latch (bit 5 of port 1)
2 - Clear the Audio IRQ flag
5 - Clears the raster line NMI latch
6 - Clears the Spare IO1 NMI (push button) latch

Port 000Ah - Write - Label "port_hw_enable"

Allows individual AMOEBA systems to be enabled/disabled

Bits 0 - Sound system enabled when set.

Native eZ80 CPU ports:

The eZ80's peripheral port pins are configured by the ROM for their normal EZ80P usage but they can be freely reconfigured by user code if desired.

Port 009Eh - Read / (Write) - Label: "PC_DR"

This eZ80 port reads the joysticks / I.O pins:

Bits
0 - Joystick A pin 1* (up)
1 - Joystick A pin 2* (down)
2 - Joystick A pin 3* (left)
3 - Joystick A pin 4* (right)
4 - Joystick B pin 1* (up)
5 - Joystick B pin 2* (down)
6 - Joystick B pin 3* (left)
7 - Joystick B pin 4* (right)

Port 00A2h - Read / (Write) - Label: "PD_DR"

Bits
0 - RS232 serial TX (to FT232RL chip pin RX)
1 - RS232 serial RX (to FT232RL chip pin TX)
2 - RS232 serial RTS (to FT232RL chip pin CTS)
3 - RS232 serial CTS (to FT232RL chip pin RTS)
4 - Joystick B pin 9* (button 2)
5 - Joystick B pin 6* (button 1)
6 - Joystick A pin 9* (button 2)
7 - Joystick A pin 6* (button 1)

* These pins are pulled up to 3.3v or 5.0v via 10K resistors. The voltage can be selected with the PCB jumper near the joystick ports (this also selects the voltage supplied - via a 15 Ohm resistor - to pin 7 of each joystick port).

Writing to these two ports will not have any effect unless the port is reconfigured by writing to its respective data direction and control ports - see eZ80 manual for details.

Port 009Ah - (Read) / Write - Label "PB_DR"

The EZ80's "Port B" bits 1:7 are connected directly to a pin header on the PCB for external use. They are not pulled high or low but are configured as outputs by default by the AMOEBA ROM.

EZ80 "Port B" bit 0 is used as an interrupt source from the FPGA (it is configured as such by the AMOEBA ROM, see following section for more details)

Bits:
0 - Maskable interrupt from FPGA - (Vector: 0030h)
1 - Header Pin 6
2 - Header Pin 5
3 - Header Pin 8
4 - Header Pin 7
5 - Header Pin 10
6 - Header Pin 9
7 - Header Pin 12

EZ80 Interrupts Under AMOEBA:

“Maskable" interrupt sources:

- PS/2 Keyboard - internal buffer is not empty
- PS/2 Mouse - internal buffer is not empty
- Audio system - channels have looped/restarted

The three IRQ sources each have an enable mask bit. They are combined in AMOEBA and output to the EZ80 CPU’s pin "PB0", normally causing an interrupt to address 0030h.

The keyboard and mouse create interrupts whenever there is data in their respective internal 16-byte buffers and their interrupt mask bit is set. The status of the buffers can be read from port 7:

Port 0007h: "port_ps2_ctrl" (read only)

Bit 4 - Keyboard buffer status
5 - Mouse buffer status

These flags are cleared when all the buffered bytes have been read from port 0002h ("port_keyboard_data") and/or port 0006h ("port_mouse_data") The internal buffers are 16 byte FIFOs and will wrap around if their ports are not read soon enough.

To enable the keyboard interrupt, set bit 0 of port 5 (“port_irq_ctrl”) - This port is a set/reset type register which allows individual bits to be set with a single write operation. When bit 7 is set, bits in 6:0 that are one are set, when bit 7 is 0, bits that are one are reset. In both cases, bits in 6:0 that are written with zero are unchanged.

To enable the keyboard IRQ source:

```
LD A,10000001b
OUT0 (port_irq_ctrl),A
```

..to disable the keyboard interrupt:

```
ld a,00000001b
OUT0 (port_irq_ctrl),A
```

For the mouse interrupt, bit 1 is the IRQ on/off control:

To enable the mouse IRQ source:

```
LD A,10000010b
OUT0 (port_irq_ctrl),A
```

..and to disable the mouse interrupt:

```
LD A,00000010b
OUT0 (port_irq_ctrl),A
```

The audio system creates an interrupt whenever any of the channels included in the IRQ source by their control bits have raised their swap flags (and the Audio IRQ source is enabled) The status of main Audio System IRQ flag can be read from:

Port 0005h (“port_irq_flags”) - Bit 2 - When set, one or more channels have a swap flag raised.

The Audio Interrupt is cleared by writing 04h to port 9 (“port_clear_flags”)

Note that the audio interrupt flag will (re)assert itself until the swap flag of the channel responsible has been cleared (or it is removed from inclusion in the IRQ source). To reset the individual channel swap flags, write (any byte) to register 0Eh of the channel’s control group - EG:

```
LD (0FF140Eh),A      ; Reset channel 0’s swap flag
```

"Non Maskable" Interrupt sources:

- The scanline match flip-flop set, IE: Line is/was equal to the video register "irq_line"
- Spare I.O1 jumper (IE: NMI button at rear of EZ80P pressed)

These two sources set latches in AMOEBA which are individually masked and combined with the resulting signal connected to the eZ80 CPU's NMI pin (which cause an interrupt to address 0066h). The internal latches which can be read from port 5 to determine the source of the NMI:

$05: "port_irq_flags" (read only)

Bit 5 - Raster Line match flip-flop status
6 - Spare IO1 (NMI button) flip-flop status

Despite the naming convention, NMI sources as well as normal interrupts can be masked by AMOEBA to prevent them from interrupting the CPU. This is controlled by writing to port 5:

Port 0005h: "port_irq_ctrl" (write)

5 - NMI(A) - Allow Raster line interrupt NMI when set
6 - NMI(B)- Allow Spare IO1 (NMI button) NMI when set
7 - Set / Reset control

This port is a set/reset type register which allows individual bits to be set with a single write operation. When bit 7 is set, bits in 6:0 that are one are set, when bit 7 is 0, bits that are one are reset. In both cases, bits in 6:0 that are written with zero are unchanged.

To enable the scanline NMI source:

```
LD A,10100000b
OUT0 (port_irq_ctrl),A
```

..to disable the scanline NMI source:

```
LD A,00100000b
OUT0 (port_irq_ctrl),A
```

Physical interrupt connections:

EZ80 pin [PB0] is configured as a maskable interrupt input by the ROM code. It is connected to FPGA pin [p67]. The EZ80'S NMI line is connected directly to FPGA pin 2.

The Video System

AMOEBA supports the following display modes: Bitmap and Tilemap (in 256 colours) and Text mode (in 16 colours). The EZ80P has 4 bits per red, green and blue outputs so offers a total palette of 4096 colours. All modes supports a maximum resolution of 640x480 and scanline and/or pixel doubling can be used to halve the resolution vertically and/or horizontally. In addition, AMOEBA provides hardware sprites and a scan-line synchronized high-priortiy interrupt. Video Output is 60Hz VGA standard (25MHz pixel clock)

Bitmap mode:

The display generator normally operates in a simple "chunky" VGA-style where each byte read from video RAM is an index in a colour table. Four separate colour tables (palettes) are available.

When 16-colour mode is selected, each byte read from VRAM holds the colour indexes for two pixels: Bits [7:4] = Leftmost pixel index, [3:0] = Rightmost pixel index. Only the first 16 colours of a palette can be used but the display takes half as much video RAM space.

The following parameters can be set:

- Start Location of video data
- Width of display (granularity of 8 bytes)
- Modulo (bytes to add to location count at the right side of scan lines)
- Pixel count increment (value to add to the video data fetch address each pixel)

Tile map Mode:

Each word (pair of bytes) read from VRAM selects an 8x8 tile to display. Each tile definition is 64 bytes long (in a linear 1 byte=1 pixel colour index format, left to right, top to bottom). Tile 0 is aligned to the start of video RAM (800000h), tile1 is at 800040h etc.. Theoretically 8192 tiles can be defined although VRAM space is needed for the tile map too.

The following tilemap parameters can be set:

- Start address of tilemap in VRAM
- Horizontal tile increment (Normally 2, IE: two bytes per map entry)
- Map Line Modulo (the number of bytes to skip at the end of each row of tiles)
- Number of visible tiles across screen
- Horizontal pixel fine-scroll position (0-7)
- Vertical line fine-scroll position (0-7)

It is possible to set the position of the right border (to 8 pixel granularity) to hide new map data when scrolling horizontally.

Text Mode

A variation of Tile Map Mode. Each pair of bytes read from video RAM are interpreted as a character index (first byte) and character colours (second byte). As a single byte is used for the character index, only 256 character "tiles" can be selected. The colour byte is interpreted as: bits [7:4] background colour, bits and [3:0] pixel colour - therefore only 2 colours can appear within each 8x8 tile. As this mode is a modification of Tile Map Mode, the character "tiles" still use 64 bytes of memory each, however only the first byte of each line definition block is used. This byte is a planar-style binary representation of the 8-pixel character line and is drawn serially from bit 7 (leftmost pixel) to bit 0 (rightmost pixel) by the video hardware.

Format of text mode character tiles in video RAM:

Character $00:

800000h Character 0,line 0 (single byte)
800001h - 800007h: unused
800008h Character 0, line 1 (single byte)
800009h - 80000fh: unused
800010h Character 0, line 2 (single byte)
800011h - 800017h: unused

.. and so on, for five more lines..

Character $01:

800040h Character 1, line 0 (single byte)
800041h - 800047h: unused
800048h Character 1, line 1 (single byte)
800049h - 80004fh: unused
800050h Character 1, line 2 (single byte) .. etc. for 5 more lines.
800051h - 800057h: unused

.. etc..

The following text mode parameters can be set:

- Start address of character map in VRAM (as 256 character definitions use 16384
- bytes, the lowest value for the character map should ideally be 804000h)
- Horizontal tile increment (Normally 2, IE: two bytes per map entry)
- Map Line Modulo (the number of bytes to skip at the end of each row of tiles)
- Number of visible characters across screen
- Vertical line scroll position (0-7)
- (Horizontal fine pixel scrolling is not supported in character map mode).

It is possible to mix modes in the same video frame to create split screen effects by using the scanline-synchronized NMI. When doing so it is desirable to change the video registers only once the video system has finished building a scanline. This can be achieved by polling bit 3 of port 1 (port_hw_flags), once the flag reads as 1, the registers can be safely updated.

Display Parameter Registers:

Tile Map / Text Mode Set Up Registers - all write-only.

FF1000h Start address of tile map in VRAM

FF1004h Tile step (Normally 0002h, IE: use consecutive 16 bit words for map)

FF1008h Scan Line Modulo.
This is more of a requirement of the display engine than a useful design parameter. To make the hardware read the same addresses for each block of 8 scanlines, this register must be loaded with a value equal to inverse of the data fetch read length IE: Set to 80000h - (visible tile width * tile step)
EG: 07ff60h for 80 columns.

FF100Ch Map Line Modulo.
This allows a number of bytes to be skipped at the end of each row of tiles allowing a smaller window from within a larger map to be easily displayed. The value depends on the overall width of the user's tile map and the visible window width. Use the formula: Tile Step * (Overall map width in tiles - Window width in tiles)
(If the tile map is no bigger than the visible display, set this to zero.)

FF1010h Number of tiles visible on a line, minus one.

FF1011h Horizontal pixel scroll position (0-7)

FF1012h Vertical line scroll position (0-7)

Bit Map Set Up Registers - all write-only:

FF1020h Start address of bitmap in VRAM

FF1024h Horizontal pixel address increment (Normally 0001h)

FF1028h Unused: Set to 0

FF102Ch Line Modulo.
This allows a number of bytes to be skipped at the end of each line of pixels allowing a section of a larger video image to be easily shown in smaller display window. The value here depends on the visible display width value below and the overall size of the image. If the two are the same, write this register with a zero.

FF1030h Visible display width / 8, less one.

The first four registers of each group (FF1000h-FF100Ch and FF1020h-FF102Ch) require19 bit values. None of the above registers are affected by the right border control, which is merely a cosmetic mask.

Sprites:

AMOEBA hardware sprites are 16 pixels wide and have a user-definable height. Image data is in linear left-to-right, top-to-bottom format with each byte of the definition data selecting one pixel colour (0 is transparent)

The following settings affect sprites globally:

- Master sprite enable
- Active register bank (allows double-buffered sprite registers)
- Pixel double
- Line double
- Sprite layer build termination style (either by register count or by time)
- Palette select (4 palettes are available)

Individual sprites have the following controls:

- x coordinate
- y coordinate
- Height
- Definition
- Background priority control
- Mirror
- Matte mode

Each sprite is controlled by a group of four write-only control register words, each of which is 16 bits long (IE: 8 bytes in each group).

Sprite 0 control register group:

FF0800h - x coordinate [bits 9:0 used]
FF0802h - y coordinate [bits 9:0 used]
FF0804h - Height of sprite [bits 8:0 used] & Control bits [15:13]
FF0806h - Definition location [bits 14:0 used]

The Sprite 1 control register group is at FF0808h, Sprite 2 control register group is at FF0810h and so on..

The visible **x coordinate** range for normal sprites is from 100h (left side) to 37Fh (right side) IE: 640 positions. When sprites are pixel-doubled, the visible range goes from 100h to 23Fh, IE: 320 positions. Only 10 bits from the x coord register are used so values above 3FFh will be interpreted as being within 0-3FFh.

The visible **y coordinate range for normal sprites is from** 200h (top) to 3DFh (bottom). IE: 480 lines. When sprite line doubling is enabled, the visible range goes from 200h to 2EFh (IE: 240 lines). Only 10 bits from the y coord register are used so locations above 3FFh are interpreted as being within 0-3FFh.

The **Height** register sets the numbers of lines of definition data that the sprite uses (sprites can be a maximum of 511 lines tall). The upper bits of this register are used as control bits:

Bit 15 - Background priority level 0 / 1 (see details later)
Bit 14 - Horizontally mirror this sprite's image (enabled when set to 1)
Bit 13 - Matte mode (IE: set all non-zero pixels to colour index 255). Enabled when set to 1.

The **Definition location** word specifies the position of the sprite's image data within VRAM_B. Sprite definition data is accessed by the hardware in 16-byte blocks, and the word value written here is actually the memory location divided by 16 (bits 23:16 from the full eZ80 memory address are not required). EG: Writing 0000h to this register will make the sprite use image data from C00000h, writing 0001h uses data from C00010h, writing 0002h uses data from C00020h and so on.

Sprite Priorites:

AMOEBA builds the display using two seperate layers: the background layer and the sprite layer. The two layers are built simultaneously and combined afterwards, this puts some constraints on the flexibility of pixel priority:

- Sprites from the higher number registers always appear in front those from lower registers.
- Sprite to background priority is based purely on background pixel colour.

For the purposes of sprite-to-background display priority, the background colour index table is divided into 16 ranges: 00h-0Fh, 10h-1Fh, 20h-2Fh etc. Each sprite can be obscured by (IE: appear behind) none or one or more of these ranges of colours. Also, each sprite has a priority bit (bit 15 of its Height / CTRL register) which selects between two of these "palette masks".

The palette masks are located at FF1100h-FF110Fh. In each of these registers bit 0 is mask level 0 and bit 1 is mask level 1 (bits 6:2 are unused). When a sprite's control register bit 15 is 0, the sprite uses mask level 0 and when it is 1, the sprite uses mask level 1.

FF1100h : Bit 0: Preserve colours 00-0F mask level 0, Bit 1: Preserve colours 00-0F mask level 1
FF1101h : Bit 0: Preserve colours 10-1F mask level 0, Bit 1: Preserve colours 10-1F mask level 1
FF1102h : Bit 0: Preserve colours 20-2F mask level 0, Bit 1: Preserve colours 20-2F mask level 1
FF1103h : Bit 0: Preserve colours 30-3F mask level 0, Bit 1: Preserve colours 30-3F mask level 1
FF1104h : Bit 0: Preserve colours 40-4F mask level 0, Bit 1: Preserve colours 40-4F mask level 1
FF1105h : Bit 0: Preserve colours 50-5F mask level 0, Bit 1: Preserve colours 50-5F mask level 1
FF1106h : Bit 0: Preserve colours 60-6F mask level 0, Bit 1: Preserve colours 60-6F mask level 1
FF1107h : Bit 0: Preserve colours 70-7F mask level 0, Bit 1: Preserve colours 70-7F mask level 1
FF1108h : Bit 0: Preserve colours 80-8F mask level 0, Bit 1: Preserve colours 80-8F mask level 1
FF1109h : Bit 0: Preserve colours 90-9F mask level 0, Bit 1: Preserve colours 90-9F mask level 1
FF110Ah : Bit 0: Preserve colours A0-AF mask level 0, Bit 1: Preserve colours A0-AF mask level 1
FF110Bh : Bit 0: Preserve colours B0-BF mask level 0, Bit 1: Preserve colours B0-BF mask level 1
FF110Ch : Bit 0: Preserve colours C0-CF mask level 0, Bit 1: Preserve colours C0-CF mask level 1
FF110Dh : Bit 0: Preserve colours D0-DF mask level 0, Bit 1: Preserve colours D0-DF mask level 1
FF110Eh : Bit 0: Preserve colours E0-EF mask level 0, Bit 1: Preserve colours E0-EF mask level 1
FF110Fh : Bit 0: Preserve colours F0-FF mask level 0, Bit 1: Preserve colours F0-FF mask level 1

In effect, the locations above where bits have been set indicate which background colour ranges appear infront of sprites.

Global sprite control registers:

FF1801h "Sprite_control" :

Bit: 0 - Enable sprites. 1 = sprite system enabled.
1 - Register bank select. 0 = Use registers @ FF0800h , 1 = use registers @ FF0C00h
2 - Pixel double (horizontally), 1 = on
3 - Line double sprites, 1 = on
4 - Sprite generation termination: 0 = by register count, 1 = by time (see details below)

FF1803h "sprite_palette_select" :

The sprites can use a different set of 256 colours to the background video if desired. Write a value from 0 to 3 here to select the AMOEBA colour bank to be used by sprites.

Sprite resources:

A maximum of around 40 sprites can appear on any one scanline, however there are actually 128* sprite control registers. Because of the way AMOEBA builds the sprite layer, if a particular sprite is not present on a particular scanline, it does not waste time fetching image data for that sprite. This means that there may be some spare time allowing sprites from registers beyond 40 to be used instead. Whilst it is not guaranteed that these extra sprites will be drawn - it depends how busy the sprite system is - it is unusual for many sprites to appear on the same line as they are normally distributed throughout the video frame. In general, Sprite registers 40-127 are best thought of as providing “hardware sprite multiplexing”, IE: the sprites will appear if there is free time.

* There are actually two sets of 128 sprite control registers - the first set is at FF0800h and the second set is at FF0C00h. AMOEBA builds the sprite layer based on the Active Register Bank setting in the **“Sprite_Control” register at FF1801h,** bit 1. This feature allows the sprite registers to be updated during a frame and switched with single CPU write off-screen thus avoiding glitches etc.

VRAM_B contention:

When AMOEBA is building the sprite layer, the CPU is forced to wait if it tries to access VRAM_B. Usually, this is not important - generally a program writes all the required sprite definition data to VRAM_B and afterwards only needs to access the control registers. However, it may be advantageous for the CPU to have more access time, especially as VRAM_B is shared with audio.

AMOEBA allows the user to set the balance between sprite and CPU access to VRAM_B. By default, the sprite system scans 32 sprite registers and then stops. There are 794 cycles per VGA scanline, and the audio system uses 64 cycles plus overhead on some scanlines, therefore we can assume there is around 720 free cycles on each line. A sprite takes 18 cycles to draw if it is present on a particular scanline and 3 cycles if it is not, therefore the worst case scenario for this default system leaves 154 cycles available for the CPU on each scanline.

If CPU access is absolutely not required, those 154 cycles could be used for additional sprites. Conversely, if more guaranteed CPU time is needed, the system can be adjusted to scan fewer sprite registers. This is set with the following register:

FF1807h - “sprite_gen_stop_pos” [0:2] - Registers to scan:

The value written tells AMOEBA how many sprite registers to scan (**sprite control: FF1801h** bit 4 is zero)

0 - invalid - do not use this value
1 - scan 8 sprite registers
2 - scan 16 sprite registers
3 - scan 24 sprite registers
4 - scan 32 sprite registers
5 - scan 40 sprite registers (if there is enough time)
6 - scan 48 sprite registers (if there is enough time)
7 - scan 56 sprite registers (if there is enough time)

Instead of specifying the CPU / Sprite access balance in terms of sprite registers it is also possible to set it more directly in terms of cycles per line. To use this mode, set **sprite control: FF1801h** bit 4 to one - the value written to FF1807h is then interpreted as follows:

FF1807h - “sprite_gen_stop_pos” [0:2] - Maximum sprite generation time:

0 - unlimited cycles (until end of line)
1 - 128 cycles
2 - 256 cycles
3 - 384 cycles
4 - 512 cycles
5 - 640 cycles
6 - 768 cycles
7 - 768 cycles

In this mode the sprite system will abort operations at the desired time regardless of how many registers have been scanned.

Note: When Sprite Line Doubling is enabled, Sprite/CPU contention only occurs on alternate scanlines.

Colour Palette:

AMOEBA has four colour palettes. Each colour palettes is 512 bytes long and holds 256 words, each defining the colour that is to displayed when a pixel at that indexed location is selected. The data in the palette is formatted as follows:

0 - $0RGB colour 0
2 - $0RGB colour 1
4 - $0RGB colour 2
etc..

R, G and B refer to 4 bit values for the Red, Green and Blue components of the colour. The upper 4 bits of each palette word are unused. Therefore the bit assignments are:

Bits 15:12 - Unused, set a zero
Bits 11:8: - Red
Bits 7:4 - Green
Bits 3:0 - Blue

The palettes are write-only and mapped in at the following locations:

Address:

FF0000h - Palette 0
FF0200h - Palette 1
FF0400h - Palette 2
FF0600h - Palette 3

The palette which is to be used by normal video data is selected by the video register

FF1802h - Write-only - Label: "bgnd_palette_select"

Bits 0:1 = Palette 0-3 used for background

The palette which is to be used for the sprites is selected by the video register:

FF1803h - Write-only - Label: "sprite_palette_select"

Bits 0:1 = Palette 0-3 used for sprites

Right Side Border Adjust

It is possible to mask off the right side of the video window by repositioning the right border. The video register "right_border_position" at FF1804h allows this to be set with a granularity of 8 hi-res pixels. The value written to this register is the number of 25MHz pixel clocks from the start of each scanline divided by 64. Normally this is 794 clocks (IE: a full line) / 8 = 99*. Every value lower brings the border left by 8 more pixels. This register does not affect the number of bytes actually read from VRAM by the hardware.

(* In fact, only 6 bits are stored, so only positions beyond clock 512 will ever be selected)

Video Control Registers

These registers are located at FF1800h and are write-only.

Reg: FF1800h - Label: "video_control"

Bit 0 - 16 colour mode when set, 256 color mode when zero.
1 - Line doubling (240 lines when set, 480 when zero)
2 - Pixel doubling (320 pixels horizontally when set, 640 when zero)
3 - Tilemap Mode (on when set)
4 - Text Mode (on when set, also set bit 3:tilemode when using Text Mode)

Reg FF1801h - Label: "sprite_control"

Bit 0 - All sprites on (1) / off (0)
1 - Sprite system builds scanline using registers @ FF0800h (0) / FF0C00h (1)
2 - Pixel double (horizontally) on (1) / off (0)
3 - Line double mode on (1) / off (0)
4 - Quit sprite generation mode: 0 = by register count, 1 = by time.

Reg FF1802h - Label: "bgnd_palette_select"

Bits [0:1] - Select palette 0-3 used by background

Reg: FF1803h - Label : "sprite_palette_select"

Bits [0:1] = Palette 0-3 used for sprites

Reg: FF1804h - Label: "right_border_position"

Bits [0:5] = Sets position of right border (cosmetic mask, does not affect datafetch)

Reg: FF1805h - Label: "irq_line_lo"

7:0 - eight LSBs of the VGA line at which the NMI is to occur (The count is always in 480 line-mode.)

Reg: FF1806h - Label: "irq_line_hi"

0:1 - two MSBs [bits 9:8] of the VGA line at which the NMI is to occur

Reg: FF1807h - Label: "sprite_gen_stop_pos"

0:2 - Maximum sprite register or pixel count position where sprite generation is halted.

The Audio System:

There are 8 sound channels (0-3 to left stereo side, 4-7 to right stereo side) each plays 8-bit signed sound samples. There are individual frequency, volume, start location and length registers.

The audio control registers are located at FF1400h-FF14FFh. Each channel is assigned a group of 16 bytes for its (write only) control registers:

```
Channel 0: FF1400h - Location of sample (in VRAM_B)
           FF1404h - Length of sample in bytes
           FF1408h - Frequency constant
           FF140Ch - Volume
           FF140Eh - Clear swap flag
           FF140Fh - Control bits

Channel 1: FF1410h - Location of sample (in VRAM_B)
           FF1414h - Length of sample in bytes
           FF1418h - Frequency constant
           FF141Ch - Volume
           FF141Eh - Clear swap flag
           FF141Fh - Control bits

           and so on..
```

Additionally there is a single byte register at FF1480h which allows channels to be “locked” (this allows synchronization when updating the registers - detail later). Each bit in this register is assigned to one of the each channels, bit 0 = channel 0, up to bit 7 = channel 7. Also, there is a control bit to completely disable the audio hardware - this is bit 0 of port 0000Ah (“port_hw_enable” - note this register is a set/reset type.)

- **Location** - A 19 bit address of the sound sample in VRAM_B. (The full 24 bit address in eZ80 address space can be written if desired as only the lowest 19 bits are used here)
- **Length** - A 19 bit value, the length in bytes of the sound sample.
- **Frequency constant** - A 16 bit value (details below)
- **Volume** - A 7 bit value, linear range 0-40h, 40h is full volume.
- **Clear swap flag** - any byte written here clears the channel’s swap flag
- **Control bits:** The bits in each channel’s control register are assigned as follows:

```
Bit 0: When written with a 1, the channel will “immediately” restart
       using the values written to location and length registers.
       This bit does not actually store any data.

Bit 1: When set, the channel - upon reaching the end of the sound
       data - will swap to new values that have been written to
       the location and length registers since it last started.

Bit 2: When set, this channel’s swap flag will contribute to the
       Audio system’s CPU Interrupt source.
```

Internally, there are actually two sets of location, length, period and volume registers for each channel. The CPU always writes to the set of location and length registers opposite to that which the audio hardware is using. When a channel's restart bit is written to, the audio system switches the internal register sets over - this also occurs when a channel loops (IE: reaches the last sample byte) so long as bit 1 in its control register is set. Note: If a channel is locked, the switchover is delayed and will not occur until that channel is unlocked.

The volume and frequency registers act in a similar way, but have independent switchover bits which become set following a write to the volume or frequency registers (in the case of the frequency register, it is the write to its MSB that raises the frequency switchover flag). Again, locking a channel prevents the switchover occurring until it is unlocked.

To cleanly start a channel playing a simple looping sample, the procedure is:

- Lock the channel.
- Write loc/len/freq/vol registers for sound required, clear swap-on-loop flag and set restart bit.
- Unlock the channel.

To cleanly start a sound, have it play just once and "stop", the procedure is:

- Lock the channel
- Write loc/len/freq/vol registers for sound required, clear loop flag, set restart&swap-on-loop bits.
- Unlock the channel.
- Wait until the channel's swap flag has become set*
- Lock the channel
- Write to loc/len registers with a silent sample, clear swap-on-loop and restart bits.
- Unlock the channel.

* Note: Because the "restart channel" bit was written with a 1, the swap flag will go high as soon as the next audio build period occurs (as long as the channel is unlocked). Once that has happened you can write new values to the location and length registers. The checking of the loop flag can be replaced with a simple time delay of about 100 microseconds if preferred.

When updating multiple channels it may be desirable to lock all the applicable channels in one go, update all the registers sequencially as appropriate and finally unlock the channels involved. This ensures the channels will all start at exactly the same time.

Reading the channel swap flags:

As stated, when a channel reaches the end of the sound data (or is restarted with bit 0 in its control register) it sets a "swap" flag. These swap flags can be read from bit 6 of the port 1 ("port_hw_flags"). To select a particular channel's swap status bit to appear at this location, write 0 to 7 to port 8 ("port_selector") before reading "port_hw_flags". To clear each channel's swap flag, write (anything) to byte 0Eh of its register group (EG: a write to FF140Eh will clear channel 0's swap flag). The swap flags can contribute to a CPU interrupt - see the Interrupt section for details.

Audio Frequency Constant:

Each channel's "Frequency Constant" register determines the speed at which a sample clip is played: It is a 16-bit fractional proportion of the maximum sampling rate of 48828Hz. It's calculated as follows:

$$\text{Frequency constant} = \left(\frac{\text{sample rate}}{48828} * 65536\right) - 1$$

How the audio system works:

The audio system essentially has two parts, a section that handles the registers and determines which bytes are required from VRAM_B and a section that is responsible for fetching the sample bytes, buffering, mixing and outputting the sound.

The “native” sample rate of the audio output is 48828 Hz so the audio system needs to read 8 bytes (one for each channel) from VRAM_B every 512 25MHz system clock cycles. Because VRAM_B is shared with sprite data, the audio buffers are actually refilled with *4* bytes per channel at the beginning of certain scanlines (IE: those where the audio system has flagged that it needs more data). Therefore on some lines, the sprite system has to wait 64 system clock cycles before it can start its operations.

Only during the short periods when the audio system is refilling its buffers do changes made to the audio registers actually take effect. Therefore when updating the audio registers, consideration must be given to the fact that the audio hardware may latch new values from the registers at some point during the user’s audio register update code. The sound system forces the CPU to wait if it attempts to write to the registers during this time, but part of a multibyte register (such as a location value) may have already been written so a value from a partly written register could be latched by the audio hardware. The resolve this issue, channels can be “locked” - in this mode, the registers being updated by the CPU are not seen by the audio system until the channel is unlocked.

Mixing:

The outputs of the 8 channels are mixed and sent to the left and right audio signals. Channels 0-3 are mixed and sent to the left channel, channels 4-7 are mixed and sent to the right channel. Due to the nature of the mixing, the volume of the individual channels on each side is halved, so if using only 4 channels, optimal quality can be achieved by copying the four groups of control data to two channels each: 0&1, 2&3, 4&6 and 6&7.

Notes:

Unlocked channels which have zero in their length register will set their swap flags continually until the frequency accumulator has overflowed - this is because the new length value is also zero. Therefore ignore any swap flags from uninitialized channels.

When restarting multiple channels, it is only necessary to read the swap flag from one of the channels involved.

It is possible to play sounds that are longer that the available memory by using two buffers and setting the hardware to alternate between them, writing fresh data to buffer that is not playing at a given time. The procedure would be:

1. Lock the channel
2. Write loc/len/freq/vol registers for 1st buffer, clear loop flag, set restart&swap-on-loop bits.
3. Unlock the channel.
4. Wait until the channel’s swap flag has become set
5. Lock the channel
6. Write to loc/len registers for 2nd buffer, set swap-on-loop bit, clear restart bit.
7. Unlock the channel.
8. Wait until channel has looped, write new data to buffer 1
9. Wait until channel has looped, write new data to buffer 2
10. Loop to step 8

The waits can be replaced with audio interrupts etc.

PIC Communications

The EZ80P configures the Spartan3 FPGA by using a serial EEPROM and PIC microcontroller. Once the FPGA is configured, the PIC-EEPROM system can be controlled by the FPGA - this allows data to be stored on and recalled from the EEPROM (such as new FPGA configs) etc.

System to PIC communication is via simple bit-banging, with the clock on FPGA pin 81, outgoing data on FPGA pin 83 and incoming data on FPGA pin 65. Bytes are sent MSB first and bits are latched by the PIC on the rising edge of the clock. Maximum bitrate is ~ 100KHz, minimum bitrate is ~ 7 Hz. AMOEBA contains a parallel-to-serial shift register and status flags to help automate communications. The relevant ports are:

Port: 0000h - Read/Write - Label: "port_pic_data"

Bytes written here are sent serially to the PIC-EEPROM system. (Bytes sent from the PIC-EEPROM system appear here when the port is read)

Port 0001h - Read - Label: "port_hw_flags"

Bit 0 - Becomes set when a byte has been received from the PIC-EEPROM system
Bit 4 - Set whilst data is being transmitted (from port 0) to the PIC-EEPROM system.

Port 0001h - Write - Label: "port_pic_ctrl"

Bit 0 - When set, the comms data output is forced high (for databurst clocking - see below)

Action is taken by the PIC when it receives specific command bytes from the FPGA, a list follows:

CONFIG:
$88, $a1, $3f, $62 = Reconfigure FPGA from current config address base (and reset cpu)
$88, $b8, $16, $slot = Change config slot
$88, $37, $d8, $06 = make current FPGA config base address permanent (in PIC's EEPROM)*

DATABURST:
$88, $d4, $low, $middle, $high address bytes = Set databurst start address
$88, $e2, $low, $middle, $high count bytes = Set databurst length
$88, $c0 = initiate databurst to FPGA using current databurst base address and length

CONTROL:
$88, $25, $fa, $99 = Enable EEPROM programming (set programming permission to 1)
$88, $1f = Disable EEPROM programming (set programming permission to 0)

WRITE TO EEPROM:
$88, $f5, $low, $middle, $high address bytes = Erase 64KB block of EEPROM ($middle and $low = 00)*
$88, $98, $low, $middle, $high address bytes, 64 data bytes = Program bytes into EEPROM #*
$88, $8b, $xx = write $xx to EEPROM's Status Register (IE: Set protection bits: Irrelevant on SST25VF)*

GET INFO:
$88, $5c = Request PIC sends EEPROM capacity (in 128K slots)
$88, $76 = Request PIC sends the power-on boot slot
$88, $71 = Request PIC sends the slot it last configured from (MSB of the config base address used/2)
$88, $4e = Request PIC sends the firmware version
$88, $06 = Request EEPROM sends its Status Register (read the protection bits, bit 7 always reported as zero)
$88, $69 = Request PIC sends the "programming permission" status (0= programming disallowed, 1 = allowed)
$88, $79 = Request PIC sends the EEPROM type byte ($00 = original 25Xabc, $01 = SST25VFabc)

; * - Programming mode must be enabled first
; # - The 64KB EEPROM block in which the bytes are to be located must be erased prior to programming new data

PIC-EEPROM Communications protocol:

Databurst: This allows a string of bytes to be transmitted to the FPGA at high speed.

To initiate a data burst, first set up the read length and EEPROM location with appropriate commands (see DATABURST command list above) then..

1. Clear receive buffer
2. Send “initiate databurst” command code sequence
3. Wait for "OK" command acknowledge byte from PIC
4. Set com_data_out high
5. Wait for data byte in receive buffer - reads/stores it
6. Sets com_data_out low
7. Wait for data byte in receive buffer - reads/stores it
8. Loop to step 4 until all bytes received.

All other commands:

1. Clears receive buffer
2. Send command code sequence (incl any arguments)
3. Wait for byte in receive buffer and read it
4. If MSB of received byte is set - there was an error:
   0x8c = "Bad command"
   0x8b = "Write timed out"
   0x8f = "Writes are disabled"
   Else OK / data in [6:0]

Time out provisions should be made in the receiving software should be included in case of errors etc.