Virtual Boy™
Programmers Manual

Compiled by:
David Tucker
david.tucker@goliathindustries.com
http://www.goliathindustries.com/vb/

version: 0.521 - January 2005

Special thanks to:
Bob VanderClay, Ben Hanor, Alberto Covarrubias, Amos Bieler, Frostgiant, Parasyte, and DogP for contributing info to this document.

1 System Overview

1.1 Introduction

The virtual boy is a portable 3D-videogame device developed by Gunpei Yokoi (1941-1997) for Nintendo. Featuring a 3D display capable of 384x224 resolution with 4 shades of red for each eye, and powered by a 20MHz V810 RISC CPU. Originally released in 1995 for around $200 in the US, the VB quickly fell under intense criticism from the video game Industry for being over priced and unimpressive. Nintendo let the system flounder for only a year before pulling the plug on it, making it the only system released by Nintendo to date that was a major flop. When the dust had settled there where 11 US/Japan titles, 3 US titles, and additional 8 titles released only in Japan.

This is a guide to hacking the Nintendo Virtual Boy. Over the past several years, with the help of many other people, I have begun reverse engineering the VB system. This is a collection of what has been discovered so far. I hope that you will be able to glean a little useful information from this document.

1.2 Equipment needed

To hack the VB for yourself you will need some specific equipment, depending on how far you want to go. For starters, to understand the internals and code that follows you will need at least a rudimentary understanding on Assembly language, and the inner workings of a computer (memory, CPU, I/O, etc.). In order to run the demo code you will need a PC, running windows or Linux, and an Internet connection to grab the emulator and assembler or gccVB. And finally to 'hack' the real VB you will need some soldering equipment and patience, along with an EPROM programmer to actually test the code on the real thing. Finally in this document I assume that you understand the difference between Binary, Integer, and Hex number systems (Base-2, Base-10, and Base16), and how to convert between them, see appendix A for details.

1.3 Hardware Overview

The VB hardware consists of:

V810 RISC CPU clocked at 20MHz
- Intel order architecture (little endian)
- 5 Maskable Interrupts (Controller, Timer, Expansion Port, Com Port, Display Retrace)
- 64KB Program Scratch Memory (true value?)
- 96KB Display Memory
- Up to 16MB of Cartridge ROM
- Up to 8MB of Cartridge Ram (saved ram)
- Up to 16MB of cartridge expansion area

Two Reflection Technologies P4 LED Displays
- 384x224 resolution per display
- Four shades of red, at one time, from a pallet of 32 shades (64 or 128?)
- Refreshes at 50.2 Hz, period is 20ms, display redraw takes 5ms per screen
- Double buffer of video memory

Bi-directional Link Port
- Clock 50 KHz(20µs) fixed or 40-500KHz user pulsed
- Hardware Interrupt

16 bit Controller Port
- Hardware/software read
- Hardware Interrupt

16 bit timer
- 20ms/100ms clock resolution
- Hardware Interrupt

16Bit Stereo Sound Processor
- 41.7KHz with 13 bit precision
- 6 channel PCM wave generator

Hardware Sprite engine
- Display 2048 sprites simultaneously
- 32 Worlds
- Simple 'parallax' 3D support
- Affine transforms (scale, rotate, skew)

1.4 History

0.51 – Cleanup, added affine mode information, backported changes from web version of document.
0.52 – Reformated document, enhanced description of graphics modes.
0.5.21 – Converted to Open Office format, cleaned up formating of document.

2. Graphic Subsystem

2.1 Graphics Overview

The virtual boy uses two Reflection Technologies P4 LED Displays, arranged so that they oscillate opposite of each other. Each display consists of a vertical column of 224 red LEDs and a mirror that oscillates horizontally. As the mirror moves forward the LEDs are toggled on and off in accordance with the times set out in the Column_Table in order to draw vertical columns of pixels on the display. There are a total of 384 columns resulting in a final image of 384x224 pixels. Since the left and right displays are 180° out of phase with each other the screens are refreshed one after the other. Each display cycles at 50.2 Hz, so the total display period is 20ms. Each display refresh takes 5ms, so 10ms of the 20ms display cycle is given over to screen redraws. However since the VB has a total of 4 screen buffers it can generate a new image for each display while it is drawing the current image on the screens. Effectively this is a double buffer system (or a quadruple buffer).

Graphics on the VB are defined using Characters (Char), Background Maps (BGMap), Objects (Obj), and World's (World).

A char is a solitary character or 'sprite'. It is an 8x8-pixel tile that defines a 4-color image. This is the basic element that all images are created from. Both the BGMap and the OBJ elements are collections of Chars.

BGMaps are a linear (full) collection, of 64x64 chars, this is useful to display large scenes like background graphics.

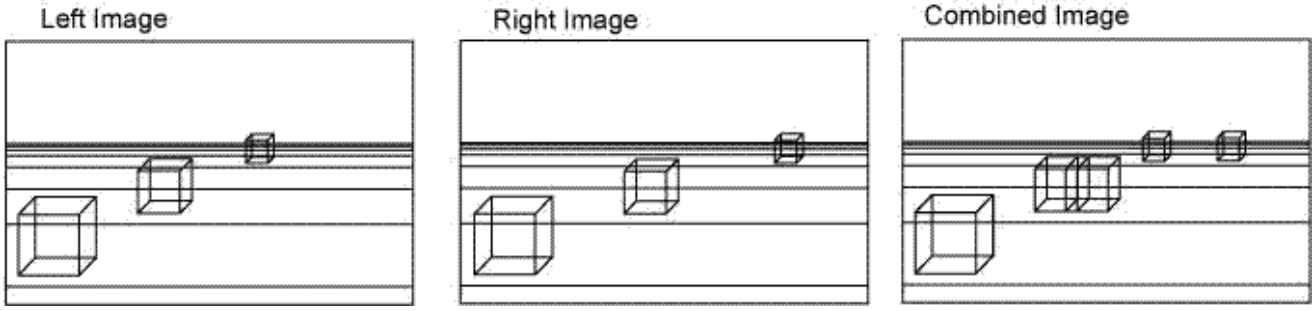
OBJs are a random (sparse) collection, there are a maximum of 4 OBJ collections with each collection containing several individual objects. There is room to define 1024 Chars total for all four OBJ's. Each OBJ defines a row and column offset followed by a pointer to the char to display. Each char can be aligned on odd boundaries and independently of other chars. OBJ's are ideal for smaller sprites that move a lot, tend to overlap, or are very sparse.

Worlds are collections of BGMap's and OBJ's. There are a maximum of 32 worlds, with each world containing one BGMap or One OBJ collection. Worlds are layered on top of each other so that closer worlds cover up worlds farther away, this helps create a 3D effect.

The data contained in the BGMap's or OBJ's along with the parallax info stored in each world are used to ultimately generate the final image to display on the two screens in the VB. These different images, displayed to each of the user's eyes, are what produce the stereoscopic 3D effect. There are two ways to generate the 3D information with the internal sprite engine, hardware paralax or pre-rendered bitmaps. It is also possible to generate a 3D effect through direct screen rendering as well.

Hardware paralax is the easiest (and thus, more common) method requiring only a single image for each Char, shared between the two displays. To achieve a stereoscopic effect, each char has a different horizontal position for each eye, using the parallax attribute. Positive values for parallax push the image further away from you and negative values bring it closer. This tends to generate a relatively weak 3D effect, but it uses fewer system resources.

Image 2.1- Paralax demonstration



The other method, pre-rendered bitmaps, involves creating a separate image for each display. This creates 3D objects, instead of 'cardboard cut-outs' that are simply on different planes. Of course, when using this method, you can use the parallax method as well.

Finally by drawing directly to the display you can completely bypass the sprite engine altogether. This gives you the freedom to generate any size image with as much 3D information as you want. However you are forced to do all of the hard work yourself and you may be restricted by the processing power of the VB. It is also possible to combine direct screen draws with the sprite engine to get the best of both worlds. In this way you can use the sprite engine to render the GUI and direct screen draws to render the game graphics.

Table 2.1 - 3D Graphics mode comparison

Style	Pros	Cons
Hardware Paralax	Only one OBJ required per 'character' Takes up less room in ROM Easier on art department	'Cardboard cut-out' effect isn't as realistic
Pre-Rendered Bitmap	Richer, more realistic graphics Less work than direct screen draws Any static 3D effect is possible	Takes more room in ROM Halves available Chars, OBJs etc Code is more complex
Direct screen drawing	Dynamic 3d Effects are possible No sprite limitations	Code is more complex Limited CPU power for rendering

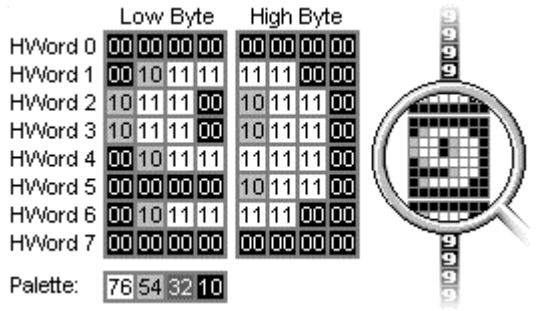
2.2 Characters

A Char is an 8x8 pixel sprite used by OBJ's, and BGMap's on the screen. Chars have the following characteristics: 2 bits per Pixel, 4-Colors actually an index into one of 8 4-color pallets. Each char is 8x8 pixels. Each line (8 pixels) is represented in 2 bytes (16 bits), with 8 lines per char. 2 bytes*8 = 16 bytes per char. Char RAM contains 2048 (0x800) Chars, arranged into four segments in the VB's address space. Char RAM is also mirrored into the range: 0x0007 8000 - 0x0007 FFFF, allowing linear access to all 2048 (0x800) Chars at once. To access char [n] in char ram: char[n] = n*16 + 0x00078000

Table 2.2 - Character Ram segmentation

Address Range	Chars
0x0000 6000 - 0x0000 7FFF	0 - 511 (0x000 - 0x1FF)
0x0000 E000 - 0x0000 FFFF	512 - 1023 (0x200 - 0x3FF)
0x0001 6000 - 0x0001 7FFF	1024 - 1535 (0x400 - 0x5FF)
0x0001 E000 - 0x0001 FFFF	1536 - 2047 (0x600 - 0x7FF)

Image 2.2 - Character layout



2.3 Background Map

BGMap's are the static images on the VB Screen. BGMap's are composed of chars from char ram, one BGMap is known as a segment. A segment is a 64x64-character image (512x512 pixels) that is 4096 characters in total. While it is possible to 'move' the whole BGMap on the display it is not possible to move the individual characters relative to each other. When displaying a BGMap on a world the H parameter must be a minimum of 8 pixels high, but can be increased in increments of 1. There are a maximum of 14 segments in the BGMap region, BGMap memory is 0x0002 0000 - 0x0003 C000 maximum. With the upper bound (0x0003 C000) being variable, it's shared with the parameter table, based on the number of active BGMap's.

1 segment: 16 bits * 4096 (0x1000) = 8192 bytes (0x2000 Bytes). Each entry (16 bits) is a index to a char in char ram (0-2047) or one Cell. Segments are laid out Left to right, top to bottom.

Table 2.3 - BGMap: Arrangement of cells within a segment,1 Segment = 4096 Cell's

Row							
0	0	1	2	...	61	62	63
1	64	65	66	...	125	126	127
				.			
				.			
				.			
62	4032	4033	4034	...	4045	4046	4047
63	4048	4049	4050	...	4093	4094	4095

There are a maximum of 14 segments in the **BGMap** region, **BGMap** memory is 0x0002 0000 - 0x0003 C000 maximum. With the upper bound (0x0003 C000) being variable, it's shared with the parameter table, based on the number of active **BGMap's**.

Table 2.4 - BGMap Cell format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BPLTS		**HFLP**	**VFLP**	0	**BCA** (0x000 - 0x7FF)										

BPLTS[0-3] - Pallet # for this character, using **BPLT#**, from VIP registers
HFLP - Horizontal Flip
VFLP - Vertical Flip
BCA - Character # to display from character ram

2.4 Object

OBJ's are the 'movable' objects on the screen, like the game character. OBJ memory is 0x0003 E000 - 0x0003 FFFF (0x02000 bytes) with each OBJ using 16x4bits for a total of 0x400 possible OBJ's. There are 4 offset registers in the VIP region (SPT0 - SPT3) that break up OBJ memory into workable chunks. Therefore you can use a maximum of 4 OBJ groups at a time. The display renderer looks at the current SPT pointer and counts backwards down to the next lower SPT pointer, starting at SPT3 for the first OBJ to be displayed. For example if we had SPT3=300, and SPT2 = 200 and SPT3 is the currently selected offset, then OBJ's 300 to 200 are displayed in that order.). So to grab OBJ3, we take the OBJ Base Address (0x0003 E000) add SPT3 (0x0003 E000 + 300 = 0x0003 E300), and index from this value, back to 0x0003 E200 grabbing objects as we go. OBJ's are special in that char's can overlap each other, can be positioned on odd boundaries, and they form a sparse matrix. This is great if you want to place a few chars randomly about the screen, i.e. making bubbles or stars.

Table 2.5 – OBJ format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JX (-0x007 - 0x17F)															
JLON	**JRON**	**JP** (-0x100 - 0x0FF)													
JY (-0x007 - 0x0DF)															
JPLTS		**HFLP**	**VFLP**	0	**JCA** (0x000 - 0x7FF)										

JX - Offset of char in X direction on Object Buffer, ranges in the negative, so a char can creep onto the screen smoothly.
JY - Offset of char in Y direction on Object Buffer, ranges in the negative, so a char can creep onto the screen smoothly.
JP - **Parallax**, True X coordinates are computed by **JX**-**JP =** True_X for the left screen, and **JX**+**JP** = True_X for the right screen.
JLON - Enable the **OBJ** for the left screen.
JRON - Enable the **OBJ** for the right screen.
JPLTS[0-3] - Pallet # for this character, using **JPLT#**, from VIP registers.
HFLP - Horizontal flip.
VFLP - Vertical flip.
JCA - Character number to display from Character Ram.

2.5 World

Worlds are a collection of OBJ's and BGMap's, that have been layered with transparences and transposed back into the screen resolution (384x224). There are a total of 32 worlds (numbered 31 to 0), but not all 32 worlds need to be used at once. Worlds are displayed back to front, starting at 31 as the farthest back, and moving forward to 0. If a world is not 'on' (LON, RON == 1) for the given screen your rendering, the world is skipped. If however the world is marked as and END world, that world and the rest are skipped. Worlds also support a few extra special effects, like sprite scaling, and rotation. As the VB renders the left and right screens it looks at the LON and RON bits respectively, to se if the world is to be displayed. Also the parallax is factored in at this time, by adding the parallax value to the GX offset for the right screen and subtracting for the left. There are 32 worlds at 16x16bits (32Bytes) for a total of 0x400 bytes. World Ram is at 0x0003 D800 - 0x0003 DBFF

Table 2.6 - World entry format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LON	**RON**	**BGM**		**SCX**		**SCY**		**OVR**	**END**	0	0	**BGMAP_BASE** (0xD)			
GX (-0xFFFF - 0x17F)															
GP (-0x100 - 0x0FF)															
GY (-0xFFFF - 0x0DF)															
MX (-0xFFFF - 0xFFF)															
MP (-0x100 - 0x0FF)															
MY (-0xFFFF - 0xFFF)															
W (0x000 - 0xFFF)															
H (0x000 - 0xFFF)															
PARAM_BASE (0x000 - 0xEBF)															
OVERPLANE_CHARACTER															
WRITING FORBIDDEN															
WRITING FORBIDDEN															
WRITING FORBIDDEN															
WRITING FORBIDDEN															
WRITING FORBIDDEN															

LON - World visible on the left screen.
RON - World visible on the right screen.
BGM - Type of world: 0-Normal **BGMap**, 1-H-bias **BGMap**, 2-Affine, 3-**OBJ**.
SCX - Number of BGMap's to combine in the X direction counted in powers of 2^n, the number of BGMaps would be 00=1 **BGMap**, 01=2, 10=4, 11=8 **BGMap's**. There can only be a maximum of 8 **BGMap's** combined in total, for both the X and Y direction.
SCY - Number of BGMap's to combine in the Y direction counted in powers of 2^n, the number of BGMaps would be 00=1 **BGMap**, 01=2, 10=4, 11=8 **BGMap's**. There can only be a maximum of 8 **BGMap's** combined in total, for both the X and Y direction.

SCX/SCY	00	01	10	11
00	1x1	1x2	1x4	1x8
01	2x1	2x2	2x4	invalid
10	4x1	4x2	invalid	invalid
11	8x1	invalid	invalid	invalid

OVR - Turns off the display wrapping, if you retrieve a pixle from (515,32) on a single bgmap it would be retrieved from (3,32) if over was not enabled. But with over enabled, nothing would be returned. (does this only apply to affine mode?)
END - No more worlds to process, used to save time, if the screen is sparse.
BGMAP_BASE - The number of the first **BGMap** to display, see **SCX**, **SCY** for total # of **BGMap's**. Always count left to right, top to bottom, for next **BGMap** to display.
GX - Screen X start position.
GY - Screen Y start position.
GP - Parallax offset for screen X position, true X coordinates are computed by **GX**-**GP** = True_X for the left screen, and **GX**+**GP** = True_X for the right screen.
MX - Buffer X start position.
MY - Buffer Y start position.

MP - **Parallax** offset for Buffer X position, actually shifts the start address to be cut out, in the X direction, to make a "Window" effect. Each eye sees a touch more on the edges than the other, make a square with your fingers and look through it with each eye in turn to see this better.
W - Width to cut out from the buffer and past on the screen.
H - Height to cut out from the buffer and past on the screen. Must be a minimum of 8 pixels high, but can be increased in increments of 1 pixel. (verify this)
PARAM_BASE - Parameter Table Base, used in H-Bias, and **Affine BGMap's**, for shifting/scaling. The last 4 bits of the PARAM_BASE must be zero. True_base = (**Param_Base** && 0xFFF0) * 2 + 0x0002 0000.
OVERPLANE_CHARACTER - Used in **Affine BGMap's** for rotation. (more info needed)

2.5.1 BGM: Normal Mode

Cut an image from the BGMap(s) starting at (MX +/-MP, MY) with a width and height of W ∧ H, and paste that image starting at (GX +/-GP, GY) onto the display image. The first BGMap is computed by taking the offset to BGMap memory (0x0002 0000) and adding BGMAP_BASE * 0x2000 (the size of one BGMap). To build the list of BGMap(s) to display, index through SCX and SCY grabbing the next BGMap in the list starting with the first BGMap as computed above.

2.5.2 BGM: Object Mode

Used for active characters. While displaying the worlds keep a counter of the next object group to display, counting from SPT3 to SPT0 as you display an OBJ group decrement the counter. OBJ groups ignore the MX, MY, MP, and GX, GY, GP values, and just display the whole 512x512 image starting at screen coordinates 0,0. Otherwise OBJ's are the same as normal BGMap's. You may only display a maximum of 4 OBJ's at a given time.

2.5.3 BGM: HBias Mode

This form, is used for 'wavy' effects each row on the screen can be shifted by a factor left or right, and this shifting is separate for the left and right displays. Param_Base points to the base offset of the H-Bias parameter table. A table of 2 HWORDS (2x16 bits) times the number of lines to be displayed. If the Image to display were 384x224 pixels, then the table would be 2 HWordsx224 in size. To display follow the procedures above, but when copying to the display buffer, add in the offset Hbias_L/R (-511 to 512) to the MX value, remember to use the appropriate value HBias_L, or HBias_R depending on the screen being rendered. So trueMXL = MX-MP+Hbias_L and trueMXR = MX+MP+Hbias_R. The true Param_Base is equal to (Param_Base*2) + 0x0002 0000.

Table 2.7 - H-bias Param table entry

31 16	15 0
HBias_L	**Hbias_R**

HBias_L - Horizontal offset for the left screen
HBias_R - Horizontal offset for the right screen

2.5.4 BGM: Affine Mode

This is used to display zooming and rotation effects. MX, MY, and MP are ignored in this mode, cut the BGMap from 0,0. GX, GY, GP, X, and Y are all used just like the Normal mode. The true Param_Base is equal to (Param_Base ∧∧ 0xFFF0) * 2 + 0x20000. Each line of the BGMap has an entry in the param_table. Each entry determines how that line is to be shifted, scaled, and rotated.

Table 2.8 - Affine Param table entry

31 0
H_SKW (12_BIT FP)
PRLX
V_SCL (12_BIT FP)
H_SCL (6_BIT FP)
V_SCW (6_BIT FP)
(Unknown)
(Unknown)
(Unknown)

H_SKW – Fixed point that defines the horizontal offset to start cutting out the image from the **BGMap**. This defines both the source X offset and the horizontal skew. Change it for each line to generate the horizontal skew. **True_h_skew** = (float)(**h_skw**/8.0)
V_SCL – Fixed point that defines the vertical offset to start cutting out the image from the **BGMap**. This defines both the source Y offset and the vertical scale. Change it for each line to generate the vertical scale. **True_v_scale** = (float)(**v_scl**/8.0)
Prlx – **Parallax** offset for screen X position, true X coordinates are computed by **GX**-**GP**-**Prlx** = True_X for the left screen, and **GX**+**GP**+**Prlx** = True_X for the right screen.
H_SCL – Fixed point scale factor for horizontal direction **true_h_scale** = (float)(**h_scl**/512.0)
V_SCW – Fixed point skew factor for vertical direction **true_v_skew** = (float)(**v_skw**/512.0)

The last three entries in the param table, along with the overplain character are unknown. (more info needed)

```
h_skw  = dest_y * y_skew
v_scl  = dest_y * y_scale

source_x = h_skw  + dest_x * h_scl
source_y = v_scl  + dest_x * v_skw
```

2.6 Colors

2.6.1 Pallet/Transparency

Each OBJ and BGMap cell is associated with a 'pallet'. There are 4 possible pallets for OBJ's (JPLT0-JPLT4) and 4 pallets for BGMap's (GPLT0-GPLT4). This allows for special pallet tricks, such as 'lightning'. Each pallet is an 8 Bit number making 4 2bit pallets, each 2 bit palled corresponds to one of the 3 brightness registers (BRTA-BRTC) with a value of 00b equaling pure black. Pallet entry 0 is always transparent.

2.6.2 Background Color

The BKCOL register tells the system what color to clear the background to. Values from 0-3 are valid, 0 being black, and 1-3 corresponding to BRTA-C.

2.6.3 Brightness

The registers BRTA, BRTB, and BRTC are the 3 brightness registers, each register holds a integer between 0-80 (0-63 or 0-127, or possibly a float_13?) that defines the hardware brightness level for that color entry. BRTA and BRTB are taken at face value but the true value of BRTC is tBRTC = BRTA+BRTB+BRTC.

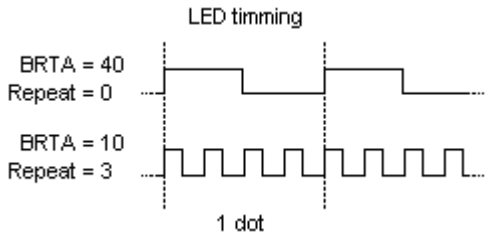
2.6.4 Repeat

Since the VB uses scanning mirrors to generate the display the dots do not have a fixed width, but vary in width based on there intensity. To help smooth out the dots and limit the gap between, the VB has a 'repeat' register (where) that allows a given dot to turn on multiple times within a given dot period. Repeat takes the literal brightness (BRTA-BRTC) of a given dot and repeats it the specified number of times, thus intensifying the dot by that number.

It is possible to change the repeat register once every display cycle, and also once every 4 columns of the display. By changing every 4 columns, you can give the illusion of having more colors active at a time.

For an example of using Repeat, imagine that you were setting up the BRTA register to an intensity of 40 (with Repeat set to the default 0). If you wanted to smooth out the appearance of the dot you could set the Repeat value to 3 (repeat 4 times), and the BRTA register to 10, to achieve a smother dot with an equivalent intensity.

Image 2.3 – Repeat timming



2.6.5 GClock

The low byte of the FRMCYC register controls the number of times to display the current screen before regenerating the display from data stored in the world tables. This is useful if you have a lot of computations to do in order to generate a screen and you can not get them all done in one display cycle. Normally this should be set to zero.

2.7 Direct Screen Draw

The VB has 4 buffers to store the display on, to perform direct screen draws we only need to write to these buffers when they are not being cleared or used for display refreshing. There are two ways to accomplish this. First we can disable screen refreshing altogether and manually control the buffers. This is what the game Water World does. Secondly we can wait for the next display refresh and draw immediately afterwards.

Since the VB uses vertical scan lines the screen memory is laid out in column-row ordering. Each column is 16 words tall with each word representing 16 pixels, using 2 bits per pixel. And there are a total of 384 columns in all.

Note*** clear bit 1 in tVIPREG.XPCTRL to disable screen refresh
Note*** bits 2&3 in tVIPREG.XPSTTS indicates the current screen buffer set being used
See red_dragon for more info

Screen Memory:

Left Frame Buffer 0 = 0x00000000
Left Frame Buffer 1 = 0x00008000
Right Frame Buffer 0 =0x00010000
Right Frame Buffer 1 = 0x00018000

2.8 Column Table

The column table helps correct for any distortions caused by an imbalance in the scanning mirrors used to make up the display. The table could be reset by the user to change the aspect ratio of the display, and to cause a certain region of the display to distort. This table must be filled in and the screen given time to stabilize (about 20 seconds) before turning on the display. Otherwise the user might suffer from eyestrain if the mirrors have not stabilized and an improper stereo image is displayed.

```
;-----------------------------------------------------------------
lb. ColTblData
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe
db. 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xfe, 0xe0, 0xbc
db. 0xa6, 0x96, 0x8a, 0x82, 0x7a, 0x74, 0x6e, 0x6a
db. 0x66, 0x62, 0x60, 0x5c, 0x5a, 0x58, 0x56, 0x54
db. 0x52, 0x50, 0x50, 0x4e, 0x4c, 0x4c, 0x4a, 0x4a
db. 0x48, 0x48, 0x46, 0x46, 0x46, 0x44, 0x44, 0x44
db. 0x42, 0x42, 0x42, 0x40, 0x40, 0x40, 0x40, 0x40
db. 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3c
db. 0x3c, 0x3c, 0x3c, 0x3c, 0x3c, 0x3c, 0x3c, 0x3c
db. 0x3c, 0x3c, 0x3c, 0x3c, 0x3c, 0x3c, 0x3c, 0x3c
;-----------------------------------------------------------------
lb. LoadColTblStart
        ;Write first half of column table
        movhi           0x0004, $0, $3            ;0x0003DC00 => $3
        movea           0xDC00, $3, $3            ;Start of column table
        movhi           0x0004, $0, $4            ;0x0003DCFF => $3
        movea           0xDCFF, $4, $4            ;End of column table
        .mov32          ColTblData, $6            ;Column Table data
lb. CTBL_Loop1
        ld.b            0x00[$6], $5              ;Data to write
        shl             0x18, $5                  ;zero high bits
        shr             0x18, $5
        st.h            $5, 0x0000[$3]            ;Column Table 1
        st.h            $5, 0x0200[$3]            ;Column Table 2
        add             0x02, $3
        add             0x01, $6
        cmp             $4,$3
        bge             0x0C
        .jump   CTBL_Loop1
                                          ;Write second half of column table
        movhi           0x0004, $0, $3            ;0x0003DDC00 => $3
        movea           0xDD00, $3, $3            ;Start of column table
        movhi           0x0004, $0, $4            ;0x0003DDCFF => $3
        movea           0xDDFF, $4, $4            ;End of column table
        .mov32          ColTblData, $6            ;Column Table data
        addi            0x7F,$6,$6
lb. CTBL_Loop2
        ld.b            0x00[$6], $5              ;Data to write
        shl             0x18, $5
        shr             0x18, $5
        st.h            $5, 0x0000[$3]            ;Column Table 1
        st.h            $5, 0x0200[$3]            ;Column Table 2
        add             0x02, $3
        add             -1, $6                    ;subtract 1
        cmp             $4,$3
        bge             0x0C
        .jump   CTBL_Loop2
```

3 Memory and I/O Registers

3.1 Condensed Memory Map

The virtual boy uses 128 Mbytes of the 32bit CPU's 4 GB addressable area. A26 ~ A24 are decoded and the 128Mbyte area is divided into 8 16-Mbyte areas. So internally all memory addresses are masked with 0x07FF FFFF. Only the significant bits of each memory area are decoded, so memory 'mirrors' itself within these regions.

Table 3.1 – Condensed Memory map

VIP Area: control registers, VRAM, DRAM	0x00000000- 0x0007FFFF
IMAGE	0x00080000- 0x00FFFFFF
Sound Area: control registers, data	0x01000000- 0x01FFFFFF
Hardware Control Area: wait state, controller, com port, timer	0x02000000- 0x020000xx
IMAGE	0x020000xx - 0x02FFFFFF
not used	0x03000000- 0x03FFFFFF
Game Pack Internal Expansion Area: unused	0x04000000- 0x04FFFFFF
NVC WRAM AREA: 64Kbytes	0x05000000- 0x0500FFFF
IMAGE	0x05010000- 0x05FFFFFF
Game Pak RAM area: 16Mbytes max	0x06000000- 0x06FFFFFF
Game Pak ROM area: 16Mbytes max	0x07000000- 0x07FFFFFF

3.2 Info at the end of the ROM

Mapped down from 0x07FF FFFF, remember the ROM replicates itself from 0x0700 0000 to 0x07FF xxxx, and we mask off the higher address lines, 0x07FF FFFF is the highest address possible.

ROM Info

0x07FF FDE0 - 0x07FF FDF3 Game Title
0x07FF FDF4 - 0x07FF FDF8 Reserved
0x07FF FDF9 - 0x07FF FDFA Manufacturer Code
0x07FF FDFB - 0x07FF FDFE Game ID Code
0x07FF FDFF ROM Version 1.x

Interrupt Vectors

0xFFFF FE00 - 0xFFFF FE0F INTKEY - Controller Interrupt
0xFFFF FE10 - 0xFFFF FE1F INTTIM - Timer Interrupt
0xFFFF FE20 - 0xFFFF FE2F INTCRO - Expansion Port Interrupt
0xFFFF FE30 - 0xFFFF FE3F INTCOM - Link Port Interrupt
0xFFFF FE40 - 0xFFFF FE4F INTVPU - Video Retrace Interrupt

0xFFFF FFF0 - 0xFFFF FFFF Reset Vector - This is how the ROM boots

3.3 Detailed Memory Map

0x0000 0000 - 0x0007 FFFF Display RAM, VIP 0x7FFFF bytes

0x0000 0000 - 0x0000 5FFF L FrameBuff0 0x6000 bytes
0x0000 6000 - 0x0000 7FFF CHR 0-511 0x2000 bytes
0x0000 8000 - 0x0000 DFFF L FrameBuff1 0x6000 bytes
0x0000 E000 - 0x0000 FFFF CHR 512-1023 0x2000 bytes
0x0001 0000 - 0x0001 5FFF R FrameBuff0 0x6000 bytes
0x0001 6000 - 0x0001 7FFF CHR 1024-1535 0x2000 bytes
0x0001 8000 - 0x0001 DFFF R FrameBuff1 0x6000 bytes
0x0001 E000 - 0x0001 FFFF CHR 1536-2047 0x2000 bytes

0x0002 0000 - 0x0003 BFFF BG Map 0x1C000 bytes ***(1)**
0x0003 C000 - 0x0003 D7FF ParamTable 0x017FF bytes
0x0003 D800 - 0x0003 DBFF World 0x00400 bytes
0x0003 DC00 - 0x0003 DDFF ColumbTbl1 0x00200 bytes
0x0003 DE00 - 0x0003 DFFF ColumbTbl2 0x00200 bytes
0x0003 E000 - 0x0003 FFFF Object 0x02000 bytes

0x0004 0000 - 0x0005 F7FF VIP Mirroring - How does this work?
0x0005 F800 - 0x0005 F870 VIP - Only accessible in HWords

*0x0005 F800: INTPND - Write the current interrupt here
*0x0005 F802: INTENB - Check if Interrupt is enabled
*0x0005 F804: INTCLR - Clear the bits in int pending
*0x0005 F820: DPSTTS - Display Status
*0x0005 F822: DPCTRL - Display Control

0x0005 F824: BRTA - Color for the given 4 bit column 0-80 (100?)
0x0005 F826: BRTB
0x0005 F828: BRTC - true_BRTC = **BRTA**+**BRTB**+**BRTC**

0x0005 F82A: REST - Counter, reset with 0x0000?
0x0005 F82E: FRMCYC - Repeat/G_CLK?
0x0005 F830: CTA - Column Table Address, R/ L (Read Only?)
*0x0005 F840: XPSTTS
*0x0005 F842: XPCTRL
0x0005 F844: VER - Static Number?

0x0005 F848: SPT0 - Pointers to the 4 OBJ group's in OBJ memory
0x0005 F84A: SPT1
0x0005 F84C: SPT2
0x0005 F84E: SPT3

0x0005 F860: GPLT0 - Set the BGMap current color pallet
0x0005 F862: GPLT1 - Selected by BPLTS from the BGMap Attrib Table
0x0005 F864: GPLT2 - There are 4 '2bit' pallets per GPLTx register
0x0005 F866: GPLT3
0x0005 F868: JPLT0 - Set the OBJ current color pallet
0x0005 F86A: JPLT1 - Selected by JPLTS from the OBJ Attribute table.
0x0005 F86C: JPLT2
0x0005 F86E: JPLT3

0x0005 F870: BKCOL - Background Color (0-3)

0x0007 8000 - 0x0007 FFFF CHR Data, mirrored from above (Linear) 0x8000 bytes
Serial access memory, 0x700 bytes (where?)
0x0008 0000 - 0x00FF FFFF Mirroring of RAM, from 0x0000 0000 - 0x0007 FFFF

0x0100 0000 - 0x0100 05FF Sound Memory
- Each data ram region has 32 6-bit registers addressed on even word boundaries, data mask with 0x3F

0x0100 0000 - 0x0100 007F	Sound1 Data Ram	0x20 bytes
0x0100 0080 - 0x0100 00FF	Sound2 Data Ram	0x20 bytes
0x0100 0100 - 0x0100 017F	Sound3 Data Ram	0x20 bytes
0x0100 0180 - 0x0100 01FF	Sound4 Data Ram	0x20 bytes
0x0100 0200 - 0x0100 027F	Sweep Data Ram	0x20 bytes
0x0100 0280 - 0x0100 02FF	Modulation Data Ram	0x20 bytes

- data masked with 0xFF, all registers are 8 bit's

0x0100 0400 - 0x0100 05FF Sound Control Registers

- Standard wave
0x0100 0400: S1CTRL - Sound1 Control Reg.
0x0100 0404: S1LEN - Length Reg.
0x0100 0408: S1FL - Frequency low byte
0x0100 040C: S1FH - Frequency high byte
0x0100 0410: S1?L - Unknown
0x0100 0414: S1?H - Unknown
0x0100 0418: S1INST - Instrument

- Standard wave
0x0100 0440: S2CTRL - Sound2 Control Reg.
0x0100 0444: S2LEN - Length Reg.
0x0100 0448: S2FL - Frequency low byte
0x0100 044C: S2FH - Frequency high byte
0x0100 0450: S2?L - Unknown
0x0100 0454: S2?H - Unknown
0x0100 0458: S2INST - Instrument

- Standard wave
0x0100 0480: S3CTRL - Sound3 Control Reg.
0x0100 0484: S3LEN - Length Reg.
0x0100 0488: S3FL - Frequency low byte
0x0100 048C: S3FH - Frequency high byte
0x0100 0490: S3?L - Unknown
0x0100 0494: S3?H - Unknown
0x0100 0498: S3INST - Instrument

- Standard wave
0x0100 04C0: S4CTRL - Sound4 Control Reg.
0x0100 04C4: S4LEN - Length Reg.
0x0100 04C8: S4FL - Frequency low byte
0x0100 04CC: S4FH - Frequency high byte
0x0100 04D0: S4?L - Unknown
0x0100 04D4: S4?H - Unknown
0x0100 04D8: S4INST - Instrument

- Sweep/Modulation

0x0100 0500:	S5CTRL	- Sound5 Control Reg.
0x0100 0504:	S5LEN	- Length Reg.
0x0100 0508:	S5FL	- Frequency low byte
0x0100 050C:	S5FH	- Frequency high byte
0x0100 0510:	S5?L	- Unknown
0x0100 0514:	S5?H	- Unknown
0x0100 0518:	S5INST	- Instrument

- Noise

0x0100 0540:	S6CTRL	- Sound6 Control Reg.
0x0100 0544:	S6LEN	- Length Reg.
0x0100 0548:	S6FL	- Frequency low byte
0x0100 054C:	S6FH	- Frequency high byte
0x0100 0550:	S6?L	- Unknown
0x0100 0554:	S6?H	- Unknown
0x0100 0558:	S6INST	- Instrument

0x0100 0580: SMREG - Main control register

0x0200 0000 - 0x0200 002C HCREG (hardware control registers)

data masked with 0xFF, all registers are 8 bit's

R W	**0x0200 0000:**	**LPC**	**- Link Port Control Reg.**	
R W	10000000	IntDisable	- 1-clears and disables interrupts - 0-enables	
	01000000	RFU	- Unused, set to 1	
	00100000	RFU	- Unused, set to 1	
R W	00010000	ClockSelect	- 0-internal clock (20 MHz/ - 40-500KHz), 1-external clock	
	00001000	RFU	- Unused, set to 1	
W	00000100	ComStart	- 1-starts communications on - Falling edge of clock	
R	00000010	ComStatus	- 1-during communication, 0-on idle	
	00000001	RFU	- Unused, set to 1	

R W	**0x0200 0004:**	**LPC2**	**- Link Port Control Reg.**	
R W	10000000	IntDisable	- 1-clears and disables interrupts - 0-enables	
	01000000	RFU	- Unused, set to 1	
	00100000	RFU	- Unused, set to 1	
R W	00010000	IntLevel	- ???	
R W	00001000	ControlSig	- ???	
R	00000100	ControlSample	- ???	
R W	00000010	ControlWrite	- ???	
R	00000001	ControlRead	- ???	

- In/Out data for the timer, keypad and link port -

R W	**0x0200 0008:**	**LPT**	**- Link Port Transmit data**
R	**0x0200 000C:**	**LPR**	**- LinkPort Receive data**
R	**0x0200 0010:**	**KLB**	**- Keypad LowByte**
R	**0x0200 0014:**	**KHB**	**- Keypad HighByte**
R W	**0x0200 0018:**	**TLB**	**- Timer LowByte**
R W	**0x0200 001C:**	**THB**	**- Timer HighByte**

R W	**0x0200 0020:**	**TCR**	**- Timer Control Reg.**	
	10000000	RFU	- Unused, set to 1	
	01000000	RFU	- Unused, set to 1	
	00100000	RFU	- Unused, set to 1	
R W	00010000	Tclock	- resolution of the clock 1-20µs - 0-100ms (default)	
R W	00001000	TINT	- 1-enable interrupt, 0-disable	
W	00000100	TClear	- 1-clear status flag	
R	00000010	TStat	- 1-counted to zero, 0-disabled	
R W	00000001	TEnable	- 1-restart count, 0-disable	

R W	**0x0200 0024:**	**WCR**	**- Wait States Control Register**	
	10000000	RFU	- Unused, set to 1	
	01000000	RFU	- Unused, set to 1	
	00100000	RFU	- Unused, set to 1	
	00010000	RFU	- Unused, set to 1	
	00001000	RFU	- Unused, set to 1	
	00000100	RFU	- Unused, set to 1	
R W	00000010	WEXP	- 1-1 wait, 0-2 wait (default)	
R W	00000001	WROM	- 1-1 wait, 0-2 wait (default)	

R W	0x0200 0028:	KCR	- Keypad Control Reg.
R W	10000000	IntDisable	- 1-clears and disables interrupts - 0-enables (default)
	01000000	RFU	- Unused, set to 1
R W	00100000	DataLatch	- 1-software data latch - 0-hardware read
R W	00010000	Dclock	- software data clock 1,0,1… - Software reads, 0-hardware read
	00001000	RFU	- Unused, set to 1
W	00000100	KeyStart	- 1-start hardware read - 0-idle (default)
R	00000010	KeyStatus	- 1-during communications, 0-idle
R W	00000001	Suspend	- 1-suspend read, 0-enable (default)

0x0400 0000	**0x04FF FFFF**	**Expansion area**	**0x00FF FFFF bytes max**
0x0500 0000	**0x0500 FFFF**	**Program RAM**	**0xFFFF bytes (mask with 0xFFFF)**
0x0600 0000	**0x0600 1FFF**	**Cartridge RAM**	**0x00FF FFFF bytes max**
0x0700 0000	**0x07FF FFFF**	**Cartridge ROM**	**0x00FF FFFF bytes max *(2)**

***(1)** The boundary between the BG Map and the Param Table is variable

***(2)** All ROM's must be powers of 2 in size (256k, 512, 1024, 2048 etc.) The ROM is placed at 0x700 0000 - up but due to addressing rollover you can always read the ROM backwards from 0x07FF FFFF down, this is how the reset vector is read.

4 Hardware Interfaces

4.1 Controller

To read the Keypad, write 0x84 to the Keypad Control Reg (0x0200 0028) to start the read cycle. Read the Keypad Control Reg (0x0200 0028) until the status bit (0x02) is zero. Than read the Keypad HighByte (0x0200 0014) and Keypad LowByte (0x0200 0010). Mask both with 0xFF to clear any sign extensions, and put them together. The 16 bits correspond to the 16 buttons on the controller. It is usually good practice to make sure the button was released before continuing on in your program.

Table 4.1 - Button Data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rdd	rdl	sel	str	ldu	ldd	ldl	ldr	rdr	rdu	lbb	rbb	b	a	1	bat

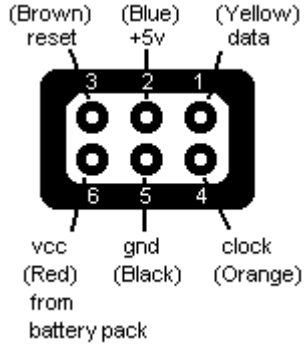
rdx – Right DPad, where x is Up, Down, Left, Right
ldx – Left DPad, where x is Up, Down, Left, Right
sel – Select
str – Start
lbb, **rbb** – Left/Right Button on back of controller
bat – Battery low, may flicker so test multiple times.

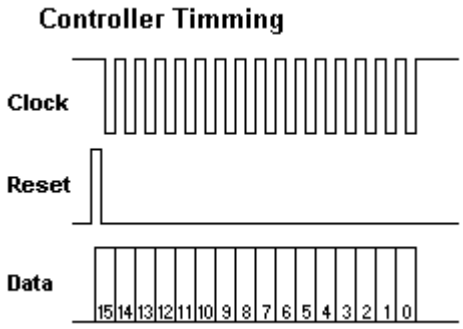
```
;;HCREG offset defines for readability
df. HCREGR_KLB            0x00000010
df. HCREGR_KHB            0x00000014
df. HCREGR_KCR            0x00000028

; Read Keypad, and store the 16 bit result in Register $3
movhi            0x0200, $0, $4           ; Pointer to Hardware Control Reg, 0x02000000 => $4
movea            0x0080, $0, $2           ;0x00000080 => $2
st.b             $2, HCREGR_KCR[$4]       ;Tell Controller to reset
add              0x04, $2                 ;0x00000084 => $2
st.b             $2, HCREGR_KCR[$4]       ;Tell Controller to read
ld.b             HCREGR_KCR[$4], $2       ;Read status
andi             0x0002, $2, $2
bne              -8                       ;Read Status until Zero
ld.b             HCREGR_KHB[$4], $2       ;Read Upper Byte
shl              0x18, $2                 ;remove sign bit
shr              0x10, $2                 ;and shift left 8 bits
ld.b             HCREGR_KLB[$4], $3       ;Read Lower Byte
shl              0x18, $3                 ;remove sign bit
shr              0x18, $3
or               $2, $3                   ;put them together in $3
```

To read data from the controller by hand drive Reset (data latch) high, then the first bit from the controller will be ready to read before the first tick of the clock. From there drive the clock high and read a bit, repeat an additional 15 times, and end with the clock high. Everything is latched on the rising edge, on the data latch line or the clock. Also the bits are inverted a High value means no button was pushed. Bit1 should always be low (a logical 1) and Bit0 is the battery status bit, it should be ignored. The controller is powered from pin 2 even though pin6 is the power from the batteries. It is not necessary to switch the controller on to get it to work.

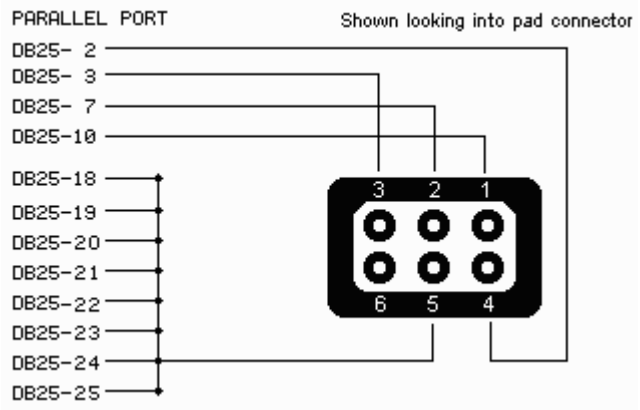
Image 4.1 - Controller Connector - looking into plug on end of cable.





To make an adapter to hook the controller to the parallel port of a PC wire up the controller to a male DB25 pin parallel port adapter using the following diagram. Once the adapter is maid you can use a program like SNESKey (http://www.csc.tntech.edu/~jbyork/) to read the controller, or you can read it by hand using the above info.

Image 4.2 – Link Port to parallel port adapter



4.2 Link Port

The link port is a 'pseudo' com port, there is a 'clock' line that all transmissions are synchronized on. This clock by default is driven at 50KHz (20 µs period), and data is latched on the rising edge. Data is transmitted/received 8 bits at a time starting with the MSB. At the default clock rate it takes 160 µs to complete a transmission. There is a generic control line that can be used to control the flow of communication. Also it appears there is a Synch In/Out pair that helps the VB to synch the send and receive data to the clock. There is a clock pull with a 20 ms period driven out of the Synch Out line at power up of the VB. The full pin out of the link port is not known at this time, here is the pin out as it stands now. (pull up the synch and ctl lines?) (fill in later)

Image 4.3 - Link Port – looking into VB port

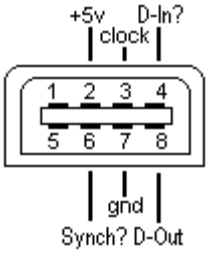
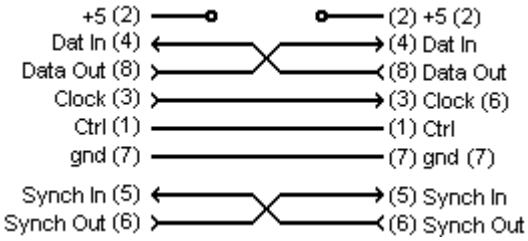


Image 4.4 - Link Cable - untested



4.3 Cartridge

Image 4.4 – Cartrige overview

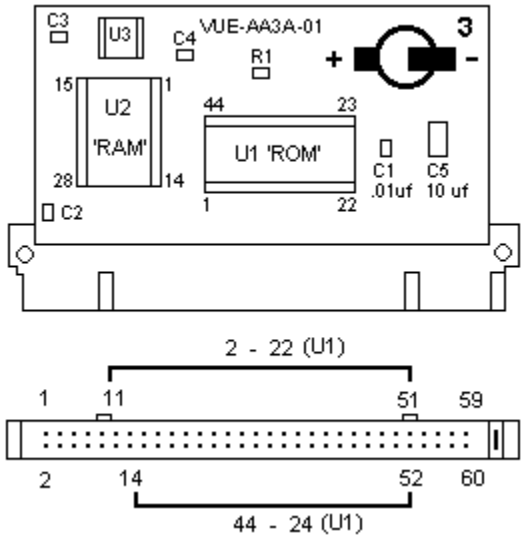


Table 4.2 – Cartrige Pinout

Cart Edge(top)	ROM	Ram	Cart Edge(bottom)	ROM	Ram
1 (gnd)			2 (gnd)		
3		27 (WE\)	4 (/ES)		
5 (/NC)			6		20 (CS1\)
7		26 (CS2)	8 (+5v)		
9 (/INTCRO)			10	(A22)	
11	2 (A18)		12	(A21)	
13	3 (A17)		14	44 (A20)	
15	4 (A7)	3 (A7)	16	43 (A19)	
17	5 (A6)	4 (A6)	18	42 (A8)	25 (A8)
19	6 (A5)	5 (A5)	20	41 (A9)	24 (A9)
21	7 (A4)	6 (A4)	22	40 (A10)	21 (A10)
23	8 (A3)	7 (A3)	24	39 (A11)	23 (A11)
25	9 (A2)	8 (A2)	26	38 (A12)	2 (A12)
27	10 (A1)	9 (A1)	28	37 (A13)	
29	11 (A0)	10 (A0)	30	36 (A14)	
31	12 (/CE)		32	35 (A15)	
33 (gnd)	13 (GND)		34	34 (A16)	
35	14 (/OE)	22 (OE\)	36 (+5v)	33 (/BYTE)	
37	15 (D0)	11 (D0)	38	31 (D15)	
39	16 (D8)		40	30 (D7)	19 (D7)
41	17 (D1)	12 (D1)	42	29 (D14)	
43	18 (D9)		44	28 (D6)	18 (D6)
45	19 (D2)	13 (D2)	46	27 (D13)	
47	20 (D10)		48	26 (D5)	17 (D5)
49	21 (D3)	15 (D3)	50	25 (D12)	
51	22 (D11)		52	24 (D4)	16 (D4)
53 (+5v)			54 (+5v)	23 (Vdd)	
55 (Rsound In)			56 (Lsound In)		
57 (Rsound Out)			58 (Lsound Out)		
59 (gnd)			60 (gnd)		

ROM
Toshiba TC53x200 or equivalent mask ROM. Can be replaced with a 27Cx00 EPROM, or 29Wx00 flash ROM, where x is 2, 4, 8, or 16.
1 - /NC
44 - /NC Used for the 32mbit ROM, this normally would be A20.
23 - +5v
33 - /BYTE (Always held high the chip is permanently in word mode)
13,32 – GND

RAM
Cypress CY6264 or equivalent SRAM.

Edge Connector
Pin 4 – (/ES), Expansion area select, driven low when accessing memory from 0x0400 0000-0x04FF FFFF
Pin 5 – Unknown, possibly reset
Pin 9 – (/INTCRO), Expansion port interrupt, drive low to generate an interrupt.
Pin 55,56 – L/Rsound in, analog sound input to right speaker
Pin 57,58 –L/Rsound out, analog sound form onboard sound processor. Hook pin 55 to 57 and pin 56 to 58 for normal sound operation.

Extra Pins for Installing a Flash ROM (more info needed)
9 RDY/BY\ - Output for flash ROM
10 RESET\ - Reset the flash ROM
12 WE\ - Write Enable

5 CPU

5.1 Overview

The Virtual Boy is based on NEC's V810 CPU core with the following added features. A custom interrupt controller, a bus wait state generator, a built in timer, link port controller, and a game pad controller. In addition the CPU core may have extra floating point opcodes as well.

The V810 CPU is based on a 32 bit RISC (Reduced Instruction Set Computer) architecture using a combination of 16-bit and 32-bit instructions to reduce the compilation size. It has 32 general-purpose registers (r0-r31), a Program Counter (**PC**), and 10 system registers. All registers are 32-bits wide, and all general-purpose registers can be used in any register operation as either data or and address register. Register 0 (r0) is the 'zero' register, its contents are always zero.

Table 5.1 - General Purpose Register Summary

Register	Name	Description
r0	Zero register	Always holds zero
r1 - r25	-	General purpose
r26	String destination bit offset	Used in BitString instruction's
r27	String source bit offset	
r28	String length register	
r29	String destination address register	
r30	String source address register	
r31	Link pointer	Stores the return address of a JAL instruction

Table 5.2 - System Register Summary

Register	Name	Application	Operation
s0	EIPC	Save status registers for exception/interrupt	Saves the PC during an exception or interrupt
s1	EIPSW	Save status registers for exception/interrupt	Saves the PSW during an exception or interrupt
s2	FEPC	Save status registers for NMI/duplex exception	Save the PC during an NMI or duplex exception
s3	FEPSW	Save status registers for NMI/duplex exception	Save the PSW during an NMI or duplex exception
s4	ECR	Exception cause register	Upper 16-bits (FECC) holds the exception code for a NMI/duplex exception, lower 16-bits (EICC) holds the code for exception/ interrupt
s5	PSW	Program status word	Flags indicating status of the CPU
s6	PIR	Processor ID register	Identifies the CPU type number, set to 0x0810x (x=unknown)
s7	TKCW	Task control word	Controls floating-point operations
s8 - s23	Reserved		
s24	CHCW	Cache control word	Controls the on-chip instruction cache
s25	ADTRE	Address trap register	When the address in this register matches the PC value execution jumps to a predefined address.
s26 - s31	Reserved		

The PSW (Program Status Word) is a set of flags that indicates the status of the CPU and the result of certain instruction executions. In particular the flags CY, OV, S, and Z are used extensively by the conditional branch instructions.

Table 5.3 - PSW summary

31 20	19 16	15	14	13	12	11 10	9	8	7	6	5	4	3	2	1	0
RFU	IntLevel	NP	EP	AE	ID	RFU	FRO	FIV	FZD	FOV	FUD	FPR	CY	OV	S	Z

RFU - Reserved for Future Use

Table 5.4 - PSW details

Bit	Name	Description
31-20	RFU	Unused fixed to 0
19-16	IntLevel	Maskable interrupt level (0-15)
15	NP	NMI pending, non maskable interrupt is being handled
14	EP	Exception pending, exception, trap or interrupt is being handled
13	AE	Address Trap Enable
12	ID	Interrupt disabled, 1-disable, 0-enable
11,10	RFU	Unused fixed to 0
9	FRO	Floating Reserved Operand
8	FIV	Floating Invalid
7	FZD	Floating Zero Divide
6	FOV	Floating Overflow
5	FUD	Floating Underflow
4	FPR	Floating Precision
3	CY	Carry
2	OV	Overflow
1	S	Sign, result is negative
0	Z	Zero, result is zero

The V810 uses a little endian addressing, that is to say the least significant byte comes first in a multi byte sequence. The standard data types include a **Byte** (8-bits), **HWord** (16-bits), and a **Word** (32-bits) in both signed and unsigned form. Words must be aligned on a word boundary, with the lower 2 bits masked to zero. And **HWords** must be aligned on a **HWord** boundary, with the least significant bit masked to 0. The V810 also supports **BitString** and 32-bit **floating-point** data types. The **floating-point** data type conforms to the 32-bit IEEE single format.

Table 5.5 - IEEE 32-bit floating-point format

31	30 23	22 0
S	exp (8)	mantissa (23)

BitStrings are a variable length string of bits ranging from 0 to 2^32-1 bits long. To define a **BitString** you must define 3 parameters:
- Address of the start of the string in memory aligned to a word boundary (last 2 bits are 0)
- Bit offset into data (0 to 31)
- Length of the string in bits (0 to 2^32 - 1)

When using **BitString** instructions load the appropriate data as defined above into the general purpose registers r26-r30 to define the source and destination strings, before calling a **BitString** opcode.

The V810 supports a full 32 bit addressing space (4-gigabytes). The handling of I/O is flexible, supporting both 32-bit memory mapped I/O and a full 32-bit port mapped I/O, however the VB only utilizes the memory mapped I/O. The external data buss supports both a 32-bit data mode and a 16-bit mode, but the VB only utilizes the 16-bit mode.

5.2 Instruction Summary

5.2.1 V810 Opcode Formats

Form 1 (I) - Register to Register computation, 16 bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						**reg2**					**reg1**				

Form 2 (II) - Immediate to Register computation, 16 bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						**reg2**					**Imm5**				

Form 3 (III) - Conditional Branch, 16 bit instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						**disp9**									0

Form 4 (IV) - Medium Jump, 32 bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						**disp26**									0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
disp26 (continued)															

Form 5 (V) - 3 operand instruction, 32 bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						**reg2**					**reg1**				

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
imm16															

Form 6 (VIa/b) - load/store instruction, 32 bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						**reg2**					**reg1**				

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
disp16															

Form 7 (VII) - extended instruction, 32 bits (floating point/bitstring)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						**reg2**					**reg1**				

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
sub-opcode						**RFU**									

RFU - Reserved for Further Use

reg1,2 - 5 bit int referencing general purpose registers 0 to 31.
imm5 - 5 bit immediate data, sign extended to 32 bits.
Disp9 - 9 bit displacement, sign extended to 32 bits.
imm16 - 16 bit Immediate data, sign extended to 32 bits.
disp16 - 16 bit displacement, sign extended to 32 bits.
disp26 - 26 bit displacement, sign extended to 32 bits.
regID - 5 bit int referencing system registers 0 to 31.
vector - 5-bit address of, trap vector 0-31.

5.2.2 Opcode Summary

opcode	form	instruction		summary	
- Register to Register computation					
0x00	I	mov	reg1, reg2	Move	reg2 ← reg1
0x01	I	add	reg1, reg2	Add	reg2 ← reg2 + reg1
0x02	I	sub	reg1, reg2	Subtract	reg2 ← reg2 - reg1
0x03	I	cmp	reg1, reg2	Comparison	reg2 - reg1
0x04	I	shl	reg1, reg2	Logical shift left	
0x05	I	shr	reg1, reg2	Logical shift right	
0x06	I	jmp	[reg1]	Unconditional Branch	PC ← reg1
0x07	I	sar	reg1, reg2	Arithmetic shift right	
0x08	I	mul	reg1, reg2	Signed multiplication	r30,reg2 ← reg2 * reg1
0x09	I	div	reg1, reg2	Signed division	reg2 ← reg2 / reg1
0x0A	I	mulu	reg1, reg2	Unsigned multiplication	r30,reg2 ← reg2 * reg1
0x0B	I	divu	reg1, reg2	Unsigned division	reg2 ← reg2 / reg1
0x0C	I	or	reg1, reg2	Logical OR	reg2 ← reg2 OR reg1
0x0D	I	and	reg1, reg2	Logical AND	reg2 ← reg2 AND reg1
0x0E	I	xor	reg1, reg2	Logical XOR	reg2 ← reg2 XOR reg1
0x0F	I	not	reg1, reg2	Logical NOT	reg2 ← NOT reg1
- Immediate to Register computation					
0x10	II	mov	imm5, reg2	Move	reg2 ← sign32(imm5)
0x11	II	add	imm5, reg2	Add	reg2 ← reg2 + sign32(imm5)
0x12	II	setf	imm5, reg2	Test flag condition	
0x13	II	cmp	imm5, reg2	Comparison	reg2 - sign32(imm5)
0x14	II	shl	imm5, reg2	Logical shift left	
0x15	II	shr	imm5, reg2	Logical shift right	
0x16	II	EI		- v830?	
0x17	II	sar	imm5, reg2	Arithmetic shift right	
0x18	II	trap	vector	Software trap	- imm5 is the vector.
0x19	II	reti		Return from Interrupt	- no reg2 or imm5 data
0x1A	II	halt		Processor stop	- no reg2 or imm5 data
0x1B	UDEF			- unknown	
0x1C	II	ldsr	reg2, regID	Load system register	regID ← reg2
0x1D	II	stsr	regID, reg2	Store system register	reg2 ← regID
0x1E	II	DI		- v830?	
0x1F	II	-		- Bit String Instructions, imm5 is subopcode - see subopcode table below	
- Conditional Branch, form 3 uses 7-bit opcode - All follow the form **PC** ← **PC** + sign32(dsp9) depending on the flags					
0x40	III	bv	disp9	if Overflow	[**OV** = 1]
0x41	III	bl	disp9	if Lower (less than - unsigned) also BC - if Carry	[**CY** = 1]

0x42	III	be	disp9	if Equal also BZ - if Zero	[**Z** = 1]
0x43	III	bnh	disp9	if Not higher (less than or equal - unsigned)	[(**CY** or **Z**) = 1]
0x44	III	bn	disp9	if Negative	[**S** = 1]
0x45	III	br	disp9	Always (unconditional branch)	
0x46	III	blt	disp9	if Less than - signed	[(**SX** xor **OV**) = 1]
0x47	III	ble	disp9	if Less than or equal - signed	[((**SX** xor **OV**) or **Z**) = 1]
0x48	III	bnv	disp9	if Not overflow	[**OV** = 0]
0x49	III	bnl	disp9	if Not lower (greater than or equal - unsigned) also BNC - if Not carry	[**CY** = 0]
0x4A	III	bne	disp9	if Not Equal also BNZ - if Not zero	[**Z** = 0]
0x4B	III	bh	disp9	if Higher (greater than - unsigned)	[(**CY** or **Z**) = 0]
0x4C	III	bp	disp9	if Positive	[**S** = 0]
0x4D	III	nop	disp9	No Operation (do nothing for 1 cycle)	
0x4E	III	bge	disp9	if Greater than or equal - signed	[(**S** xor **OV**) = 0]
0x4F	III	bgt	disp9	if Greater than - signed	[((**S** xor **OV**) or **Z**) = 0]

- Misc. 32 bit instructions

0x28	V	movea	imm16, reg1, reg2	Addition no flags	reg2 ← reg1 + sign32(imm16)
0x29	V	addi	imm16, reg1, reg2	Addition	reg2 ← reg1 + sign32(imm16)
0x2A	IV	jr	disp26	Jump relative	PC ← sign32(disp26) & 0xFFFF FFFE
0x2B	IV	jal	disp26	Jump and link	r31 ← PC+4, jr disp26
0x2C	V	ori	imm16, reg1, reg2	Logical OR	reg2 ← reg1 OR sign32(imm16)
0x2D	V	andi	imm16, reg1, reg2	Logical AND	reg2 ←reg1 AND sign32(imm16)
0x2E	V	xori	imm16, reg1, reg2	Logical XOR	reg2 ←reg1 XOR sign32(imm16)
0x2F	V	movhi	imm16, reg1, reg2	Add High	reg2 ←reg1 + shl16(imm16)
0x30	VIa	ld.b	disp16[reg1], reg2	Load Byte	reg2 ← [sign32(disp16) + reg1]
0x31	VIa	ld.h	disp16[reg1], reg2	Load HWord	reg2 ← [sign32(disp16) + reg1]
0x32	UDEF			- unknown	
0x33	VIa	ld.w	disp16[reg1], reg2	Load Word	reg2 ← [sign32(disp16) + reg1]
0x34	VIb	st.b	reg2, disp16[reg1]	Store Byte	[sign32(disp16) + reg1] ← reg2
0x35	VIb	st.h	reg2, disp16[reg1]	Store HWord	[sign32(disp16) + reg1] ← reg2
0x36	UDEF			- unknown	
0x37	VIb	st.w	reg2, disp16[reg1]	Store Word	[sign32(disp16) + reg1] ← reg2
0x38	VIa	in.b	disp16[reg1], reg2	Inport Byte	reg2 ← [sign32(disp16) + reg1]
0x39	VIa	in.h	disp16[reg1], reg2	Inport HWord	reg2 ← [sign32(disp16) + reg1]
0x3A	VIa	caxi	disp16[reg1], reg2		
0x3B	VIa	in.w	disp16[reg1], reg2	Inport Word	reg2 ← [sign32(disp16) + reg1]
0x3C	VIb	out.b	reg2, disp16[reg1]	Outport Byte	[sign32(disp16) + reg1] ← reg2
0x3D	VIb	out.h	reg2, disp16[reg1]	Outport HWord	[sign32(disp16) + reg1] ← reg2
0x3E	VII	-		- Floating Point Instructions - see subopcode table below	
0x3F	VIb	out.w	reg2, disp16[reg1]	Outport Word	[sign32(disp16) + reg1] ← reg2

All instructions greater than 0x3F are undefined. Except for the Branch instructions witch use a 7-bit opcode instead of a 6-bit opcode.
Unless otherwise noted, all mathematical operations are signed.

5.2.3 - Bit String Subopcode Summary

Subopcode	Instruction	Summary
0x00	sch0bsu	- search up, for 0's
0x01	sch0bsd	- search down, for 0's
0x02	sch1bsu	- search up, for 1's
0x03	sch1bsd	- search down, for 1's
0x04-0x07	UDEF	- unknown
0x08	orbsu	- logical OR 2 bit strings together
0x09	andbsu	- logical AND 2 bit strings together
0x0A	xorbsu	- logical XOR 2 bit strings together
0x0B	movbsu	- copy the first string over the second
0x0C	ornbsu	- logical OR 2 bit strings together, NOTing the first
0x0D	andnbsu	- logical AND 2 bit strings together, NOTing the first
0x0E	xornbsu	- logical XOR 2 bit strings together, NOTing the first
0x0F	notbsu	- logical NOT the first bit string storing in the second
0x10-0x1F	UDEF	- unknown

5.2.4 - Floating Point Subopcode Summary

Subopcode	Instruction		Summary	Example
0x00	cmpf.s	reg1, reg2	- compare FP	reg2 - reg1
0x01	UDEF		- unknown	
0x02	cvt.ws	reg1, reg2	- convert int to float	reg2 ← float(reg1)
0x03	cvt.sw	reg1, reg2	- convert float to int	reg2 ← int(reg1)
0x04	addf.s	reg1, reg2	- add 2 floats	reg2 ← reg2 + reg1
0x05	subf.s	reg1, reg2	- subtract 2 floats	reg2 ← reg2 - reg1

0x06	mulf.s	reg1, reg2	- multiply 2 floats	reg2 ← reg2 * reg1
0x07	divf.s	reg1, reg2	- divide 2 floats	reg2 ← reg2 / reg1
0x08	xb		- swap low bytes	reg1 ← (reg1&FFFF0000)|((reg1<<8)&FF00)|((reg1>>8)&FF)
0x09	xh		- swap half word	reg1 ← ((reg1<<16)&FFFF0000)|((reg1>>16)&FF)
0x0A	rev		- reverse the word	reg1 ← (mirror of reg1)
0x0B	trnc.sw	reg1, reg2	- convert float to unsigned int	reg2 ← uint(reg1)
0x0C	mpyhw		- unknown	
0x0D-0x3F	UDEF		- unknown	

5.3 Instruction Details

(Fill in later)

5.4 Interrupts/Exceptions

(need better description of NMI, Maskable Interrupt and Exception handling)

Interrupts are events that interrupt the execution of a program from an external source. Interrupts are divided into maskable interrupts and non-maskable interrupts (NMI) i.e. reset. Exceptions are events that interrupt the execution of a program that are generated by the program execution. For example dividing a number by zero would generate a 'Zero Division' exception. Otherwise interrupts and exceptions are almost identical. But interrupts take precedence over exceptions. The v810 handles interrupts and exceptions through an interrupt table. When a given interrupt/exception is generated the current PC and PSW registers are saved in the EIPC/EIPSW registers. And when a NMI or Duplexed exception is generated the PC and PSW are stored in the FEPC/FEPSW registers. Next the exception cause register (ECR) is filled in with the interrupt/exception number, the PSW Int Level is set to 1+current interrupt level, the PSW EP and ID bits are set to 1 and the PC is updated to point to the interrupt handler vector.

In order for a maskable interrupt to occur the NP bit of the PSW must be 0, the EP bit of the PSW must be zero, the ID bit of the PSW must be zero. And the interrupt being fired must have an id greater than or equal to the Interrupt Level stored in the PSW.

Table 5.6 - Interrupt/Exception Table

Interrupt/Exception name	Classification	Code	Handler Addr	Restore PC
Reset	Interrupt	0xFFF0	0xFFFFFFF0	undefined
NMI	Interrupt	0xFFD0	0xFFFFFFD0	next PC ***(2)**
Duplexed Exception	Exception	***(1)**	0xFFFFFFD0	current PC
Address trap	Exception	0xFFC0	0xFFFFFFC0	current PC
Trap instruction (0x1n)	Exception	0xFFBn	0xFFFFFFB0	next PC
Trap instruction (0x0n)	Exception	0xFFAn	0xFFFFFFA0	next PC
Invalid OpCode	Exception	0xFF90	0xFFFFFF90	current PC
Divide by Zero	Exception	0xFF80	0xFFFFFF80	current PC
FIV (float invalid op)	Exception	0xFF70	0xFFFFFF60	current PC
FDZ (float zero divide)	Exception	0xFF68	0xFFFFFF60	current PC
FOV (float overflow)	Exception	0xFF64	0xFFFFFF60	current PC
FUD (float underflow)***(3)**	Exception	0xFF62	0xFFFFFF60	current PC
FPR (float degradation)***(3)**	Exception	0xFF61	0xFFFFFF60	current PC
FRO (float reserved op)	Exception	0xFF60	0xFFFFFF60	current PC
INT level n (n = 0 to 15)	Interrupt	0xFEn0	0xFFFFFEn0	next PC ***(2)**

***(1)** Exception code of first exception is stored in the lower 16-bits of ECR and the second is stored in the upper 16-bits.
***(2)** If an instruction is aborted by an interrupt (DIV/DIVU, floating-point instruction, BitString instruction) the restore PC = current PC.
***(3)** The floating-point underflow and floating-point precision degradation exceptions do not occur in the v810.

The v810 is not set up to handle more than one interrupt at a time, it can handle up to 2 exceptions. In order to support multiple interrupts at a time your interrupt code must: (verify)

- disable all further interrupts by setting the ID bit of the PSW to 1
- save the EIPC and EIPSW registers
- clear the EP bit from the PSW
- finally re-enable interrupts by setting the ID bit to 0 in the PSW

(returning from an interrupt/exception)

5.5 Reset

Power on reset causes the system registers to initialize to the following. After initialization, program execution jumps to the reset vector at 0xFFFFFFF0 and begins execution.

Table 5.7 – Power on Reset

Register	Description	State
PC	Program Counter	0xFFFFFFF0
EIPC	Status saving register for interrupt	Undefined
EIPSW		
FEPC	Status saving register for NMI	Undefined
FEPSW		
FECC	Interrupt cause register	0x0000
EICC		0xFFF0
PSW	Program status word	0x00008000
r0	General-purpose register	Fixed to 0x00000000
r1 to r31		Undefined