CowBite Virtual Hardware Specifications

Unofficial documentation for the CowBite GBA emulator
by Tom Happ

Sun Aug 4 17:21:40 2002

The most recent version is kept at
http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm
and http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.zip.

Available
With Frames and **Without Frames**

Table of Contents

1. Introduction

This document began as a scratchpad of notes that I added onto "Agent Q's GBA Spec" as I came across new information and ideas while building the CowBite emulator. Agent Q is no longer able to maintain that

document. Because there is so much additional information here, I thought it would be a shame to keep it to myself, and thus I'm releasing it again as a CowBite technical reference document. Though it has the name of the CowBite emulator on it, the information should still be more or less (and unofficially) valid on actual hardware.

I've tried to organize this document in a way that will be easy to navigate to the section you are looking for. I also try to update whenever I find that something is missing or erroneous. However, I can't catch everything without your feedback. If you find an error, have information not listed in this doc (and there is lots of *that*), or think that something could be improved upon, please contact me.

For the most part, this document is a reference and not a tutorial or instruction manual. I have done my best to provide samples and hints wherever I can, but if you are looking for a step-by-step guide, you will probably find it easier to begin with a tutorial like Dovoto's Pern Project, or one of the other tutorials on http://www.gbadev.org/ or in the links section of this page.

All of this information has been obtained legally by piecing together information from the ARM docs, gbadev.org, the gbadev mailing list, public domain demos, and other information such as the debug info from various emulators. My sincerest thanks to all of those who have mailed me with info and corrections!

Send me mail (SorcererXIII@yahoo.com) with your comments and especially with info and corrections.

Tom Happ

Revision 3.5 - (Colorful Version) Begun May 18th 2002 (Though updates are continuous)
Revision 3.0 - (HTML Version) Begun December 19th 2001
Revision 2.1 - I decided to rename this 3.0, as it turned out to be a major overhaul
Revision 2.0 - Begun July 23, 2001 (by me)
Revision 1.0 - (by Agent Q) 08/02/2001

I keep the most recent doc update at
http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm

The Basics

From the programmer's perspective, the system is composed of the following:

CPU - A 16.78 Mhz ARM7tdmi
Memory - 8 to 11 distinct areas of memory (depending on the Game Pak).
IO - Special hardware functions available to the programmer, primarily pertaining to graphics, sound, DMA, timers, serial communication, key input, and interrupts.

Programs run on the GBA are usually contained in a "Game Pak". A "Game Pak" consists mainly of ROM and possibly Cart RAM (in the form of SRAM, Flash ROM, or EEPROM, used mainly for save game info). The ROM is where compiled code and data is stored. Unlike home computers, workstations, or servers, there are no disks or other drives, so everything that might otherwise have been stored as separate resource files must be compiled into the program ROM itself. Luckily there are tools to aid in this process.

The primary means a program accesses specialized hardware for graphics, sound, and other IO is through the memory-mapped IO. Memory mapped IO is a means of communicating with hardware by writing to/reading from specific memory addresses that are "mapped" to internal hardware functions. For example, you might write to address 0x4000000 with the value "0x0100", which tells the hardware "enable background 0 and graphics mode 0". A secondary means is through the BIOS, which is embedded in the internal GBA system ROM. Using software interrupts it is possible to access pre-programmed (and hopefully optimized) routines lying in the the system ROM. These routines then access the hardware through the memory-mapped IO.

Other regions of memory that are directly mapped to the hardware are Palette RAM (which is a table consisting of all the available colors), VRAM (which performs a similar function to the video RAM on a PC - and thensome), and OAM (which contains the attributes for hardware accelerated sprites).

Programming for the GBA

For those wishing to learn how to write a program for the GBA, this is not really the right place. This document is more like a reference that you can keep open in the background and quickly thumb through as questions arise. Consider the following to be a generalization of how games are developed for the GBA.

C, C++, and ARM/Thumb assembly are the most common languages used in GBA development, mainly because they are fast and relatively low level (i.e. there is a large degree of correspondance between the structure of the language and underlying instruction set of the architecture). Members of the GBA development community have put together some development kits that greatly simplify the task of configuring a C/C++ compiler for GBA development. Two that I know of are Devkit Advance and HAM.

Most GBA programs are structured around the timing of the CPU and graphics hardware. The LCD has a refresh rate of about 59.73 hz, with each refresh consisting of a vertical draw period (when the GBA is drawing the screen) followed by a vertical blank period (when nothing is being drawn). The vertical draw and vertical blank periods are further subdivided into horizontal draw and blank periods. Programs typically use the VBlank and possibly the HBlank periods to update VRAM or graphics hardware registers in order to avoid unwanted visual artifacts, leaving the VDraw and HDraw periods to perform any software processing that will not effect the display. Common methods of syncing to VBlank include polling REG_DISPSTAT or REG_VCOUNT, calling the VBlankIntrWait BIOS function, or setting up an interrupt.

I suggest the following resources as being invaluable to homebrew (and even commercial) developers:

http://www.gbadev.org/ - This is one of the main hubs for GBA development on the web, containing numerous docs, tutorials, demos (many with source code), tools, and news.

http://www.devrs.com/gba - This is Jeff Frohwein's GBA site. Jeff Frohwein is something of a GBA dev guru; you will likely hear his name mentioned often throghout the community. His site contains many good links, documents, and tools, many of which he wrote himself.

http://groups.yahoo.com/group/gbadev/ -A forum on yahoo with a large archive of back posts.

PERN project tutorials - These great tutorials by Dovoto explain everything in an easy-to-understand, step-by-step fashion.

The Audio Advance - Documents and tutorials on the formerly enigmatic GBA sound system. Courtesy of Uze.

Unofficial GBA Software Development Kit (AKA Devkit Advance) - The most popular free devkit available.

HAM - If you want to get started *right now*, Emanuel's HAM devkit is especially easy to install, and can have you up and running in minutes.

2. CPU

This section is intended to be an overview only, detailing those aspects of the CPU which are important to understand when developing for the GBA in particular. A more thorough description of the ARM7tdmi CPU can

be found in the technical reference manuals on ARM's website.

The CPU is a 16.78 MHz ARM7tdmi RISC processor. It is a 32-bit processor but can be switched to "Thumb" state, which allows it to handle a special subset of 16-bit instructions that map to 32-bit counterparts. Instructions follow a three-stage pipeline: fetch, decode, execute. As a result, the program counter always points two instructions ahead of the one currently being executed.

CPU Registers

16 registers are visible to the user at any given time, though there are 20 banked registers which get swapped in whenever the CPU changes to various priveleged modes. The registers visible in user mode are as follows:

r0-r12: General purpose registers, for use in every day operations
r13 (SR): Stack pointer Register. Used primarily for maintaining the address of the stack. This default value (initialized by the BIOS) differs depending on the current processor mode, as follows:

```
User/System:  0x03007F00
IRQ:          0x03007FA0
Supervisor:   0x03007FE0
```

As far as I know the other modes do not have default stack pointers.

r14 (LR): Link Register. Used primarily to store the address following a "bl" (branch and link) instruction (as used in function calls)
r15 (PC): The Program Counter. Because the ARM7tdmi uses a 3-stage pipeline, this register always contains an address which is 2 instructions ahead of the one currrently being executed. In 32-bit ARM state, it is 8 bytes ahead, while in 16-bit Thumb state it is 4 bytes ahead.
CPSR: The Current Program Status Register. This contains the status bits relevant to the CPU -

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
N  Z  C  V   R  R  R  R   R  R  R  R   R  R  R  R   R  R  R  R   R  R  R R  I F T M  M M M M

0-4  (M) = Mode bits. These indicate the current processor mode:
           10000 - User mode
           10001 - FIQ mode
           10010 - IRQ mode
           10011 - Supervisor mode
           10111 - Abort mode
           11011 - Undefined mode
           11111 - System mode

5    (T) = Thumb state indicator. If set, the CPU is in Thumb state.
           Otherwise it operates in normal ARM state. Software should
           never attempt to modify this bit itself.
6    (F) = FIQ interrupt disable. Set this to disable FIQ interrupts.
7    (I) = IRQ interrupt disable. Set this to disable IRQ interrupts. On
           the GBA this is set by default whenever IRQ mode is entered.
           Why or how this is the case, I do not know.
8-27 (R) = Reserved
28   (V) = Overflow condition code
29   (C) = Carry/Borrow/Extend condition code
30   (Z) = Zero/Equal condition code
31   (N) = Negative/Less than condition code
```

Processor Modes

The ARM7tdmi has six modes: user, system, IRQ, FIQ, SVC, Undef, and Abt. The default is user mode. Certain events will trigger a mode switch. Some modes cause an alternate set of registers to be swapped in, effectively replacing the current set of registers until the mode is exited.

User: This is the default mode.

System: This is intended to be a priveleged user mode for the operating system. As far as I can tell it is otherwise the same as User mode. I am not sure if the GBA ever enters System mode during BIOS calls.

IRQ: This mode is entered when an Interrupt Request is triggered. Any interrupt handler on the GBA will be called in IRQ mode.

> Banked registers: The ARM7tdmi has several sets of banked registers that get swapped in place of normal user mode registers when a priveleged mode is entered, to be swapped back out again once the mode is exited. In IRQ mode, r13_irq and r14_irq will be swapped in to replace r13 and r14. The current CPSR contents gets saved in the SPSR_irq register.

FIQ: This mode is entered when a Fast Interrupt Request is triggered. Since all of the hardware interrupts on the GBA generate IRQs, this mode goes unused by default, though it would be possible to switch to this mode manually using the "msr" instruction.

> Banked registers: r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, and SPSR_fiq.

SVC: Supervisor mode. Entered when a SWI (software interrupt) call is executed. The GBA enters this state when calling the BIOS via SWI instructions.

> Banked registers: r13_svc, r14_svc, SPSR_svc.

ABT: Abort mode. Entered after data or instruction prefetch abort.

> Banked registers: r13_abt, r14_abt, SPSR_abt.

UND: Undefined mode. Entered when an undefined instruction is executed.

> Banked registers: r13_und, r14_und, SPSR_und.

CPU State

The ARM7tdmi has two possible states, either of which may be entered without fear of losing register contents or current processor mode.

To enter Thumb State: In this state the CPU executes 16-bit, halfword-aligned instructions. There are two ways it can be entered:

- A BX rn, where rn contains the address of the thumb instructions to be executed, +1. Bit 0 must be 1 or the switch won't be made and the CPU will try to interperet the binary Thumb code as 32-bit ARM instructions.
- Returning from an interrupt that was entered while in Thumb mode.
- Executing any arithmetic instruction with the PC as the target and the 'S' bit of the instruction set, with bit 0 of the new PC being 1.

To Enter ARM State: This is the default state. It executes 32-bit, word-aligned instructions. When in Thumb state, the CPU can be switched back to ARM state by:

- A BX rn, where rn contains the address of the ARM instructions to be executed. Bit 0 must be 0.

- Entering an interrupt.

For more complete information on the ARM7tdmi, be sure to check out ARM's technical reference manuals.

3. Memory

The following are the general areas of memory as seen by the CPU, and what they are used for.

System ROM:

```
Start: 0x00000000
End:  0x0003FFF
Size: 16kb
Port Size: 32 bit
Wait State: 0
```

0x0 - 0x00003FFF contain the BIOS, which is executable but not readable. Any attempt to read in the area from 0x0 to 0x1FFFFFFF will result in failure; what you will see on a read is the current prefetched instruction (the instruction after the instruction used to view the memory area), thus giving the appearance that this area of memory consists of a repeating byte pattern.

External Work RAM:

```
Start: 0x02000000
End:   0x0203FFFF
Size:  256kb
Port Size: 16 bit
Mirrors:  Every 0x40000 bytes from 0x02000000 to 0x02FFFFFF
```

This space is available for your game's data and code. If a multiboot cable is present on startup, the BIOS automatically detects it and downloads binary code from the cable and places it in this area, and execution begins with the instruction at address 0x02000000 (the default is 0x08000000). Though this is the largest area of RAM available on the GBA, memory transfers to and from EWRAM are 16 bits wide and thus consume more cycles than necessary for 32 bit accesses. Thus it is advised that 32 bit ARM code be placed in IWRAM rather than EWRAM.

Internal Work RAM:

```
Start: 0x03000000
End:   0x03007FFF
Size:  32kb
Port Size: 32 bit
Mirrors:  Every 0x8000 bytes from 0x03000000 to 0x03FFFFFF
```

This space is also available for use. It is the fastest of all the GBA's RAM, being internally embedded in the ARM7 CPU chip package and having a 32 bit bus. As the bus for ROM and EWRAM is only 16 bits wide, the greatest efficiency will be gained by placing 32 bit ARM code in IWRAM while leaving thumb code for EWRAM or ROM memory.

IO Ram:

```
Start: 0x04000000
End:   0x040003FF (0x04010000)
```

```
Size:  1Kb
Port Size:  Dual ported 32 bit
Mirrors:  The word at 0x04000800 (only!) is mirrored every 0x10000 bytes
          from 0x04000000 - 0x04FFFFFF.
```

This area contains a mirror of the ASIC (Application Specific Integrated Circuit) registers on the GBA. This area of memory is used to control the graphics, sound, DMA, and other features. See memory-mapped IO registers for details on the function of each register.

Palette RAM:

```
Start: 0x05000000
End:   0x050003FF
Size:  1kb
Port Size:  16 bit
Mirrors: Every 0x400 bytes from 0x05000000 to 0x5FFFFFF
```

This area specifies the 16-bit color values for the paletted modes. There are two areas of the palette: one for backgrounds (0x05000000) and another for sprites (0x05000200). Each of these is either indexed as a single, 256-color palette, or as 16 individual 16-color palettes, depending on the settings of a particular sprite or background.

VRAM:

```
Start: 0x06000000
End:   0x06017FFF
Size:  96kb
Port Size: 16 bit
Mirrors: Bytes 0x06010000 - 0x06017FFF is mirrored from 0x06018000 - 0x0601FFFF.
         The entire region from 0x06000000 - 0x06020000 is in turn mirrored every
         0x20000 bytes from 0x06000000 - 0x06FFFFFF.
```

The video RAM is used to store the frame buffer in bitmapped modes, and the tile data and tile maps for tile-based "text" and rotate/scale modes.

OAM:

```
Start: 0x07000000
End:   0x070003FF
Size:  1kb
Port Size: 32 bit
Mirrors: Every 0x400 bytes from 0x07000000 to 0x07FFFFFF
```

This is the Object Attribute Memory, and is used to control the GBA's sprites.

The following areas of memory are technically cart-dependent, but can generally be expected to behave as described.

GAME PAK ROM:

```
Start: 0x08000000
Size:  The size of the cartridge (0 - 32 megabytes)
Port Size: 16 bit
Wait State: 0
```

The ROM in the game cartridge appears in this area. If a cartridge is present on startup, the instruction found at location 0x08000000 is loaded into the program counter and execution begins from there. Note that the transfers to and from ROM are all 16 bits wide.

GAME PAK ROM IMAGE 1:

```
Start: 0x0A000000
Size:  The size of the cartridge (0 - 32 megabytes)
Port Size:  16 bit
Wait State: 1
```

This is a mirror of the ROM above. Used to allow multiple speed ROMs in a single game pak.

GAME PAK ROM IMAGE 2:

```
Start: 0x0C000000
Size:  The size of the cartridge (0 - 32 megabytes)
Port Size: 16 bit
Wait State: 2
```

This is a mirror of the ROM above. Used to allow multiple speed ROMs in a single game pak.

CART RAM:

```
Start: 0x0E000000 (also seem to appear at 0x0F000000)
Size:  0 - 64 kb
Port Size: 8 bit
```

This is either SRAM or Flash ROM. Used primarily for saving game data. SRAM can be up to 64kb but is usually 32 kb. It has a battery backup so has the longest life (in terms of how many times it can be written to) of all backup methods. Flash ROM is usually 64 kb. Its lifespan is determined by the number of rewrites that can be done per sector (a 10,000 rewrite minimum is cited by some manufacturers).

EEPROM:
This is another kind of cart memory, but operates differently from SRAM or Flash ROM. Unfortunately, I don't know the details of how it can be accessed by the programmer (mail me if you have more information on it). It uses a serial connection to transmit data. The maximum size is 128 mb, but it can be any size, and is usually 4 kb or 64 kb. Like Flash ROM it has a limited life; some manufacturers cite a minimum of 100,000 rewrites per sector.

There may be other regions of memory known as DEBUG ROM 1 and DEBUG ROM 2, though I really don't know whether these are a part of commercial carts or if they are mapped to some part of the internal ROM, or if they're even available on a standard GBA.

Note that EWRAM, IWRAM, VRAM, OAM, Palette RAM are all initialized to zero by the BIOS (i.e. you can expect them to be zeroed at startup).

4. Graphics Hardware Overview

The GBA has a TFT color LCD that is 240 x 160 pixels in size and has a refresh rate of exactly 280,896 cpu cycles per frame, or around 59.73 hz. Most GBA programs will need to structure themselves around this refresh rate. Each refresh consists of a 160 scanline vertical draw (VDraw) period followed by a 68 scanline blank (VBlank) period. Furthermore, each of these scanlines consists of a 1004 cycle draw period (HDraw) followed by a 228 cycle blank period (HBlank). During the HDraw and VDraw periods the graphics hardware processes background and obj (sprite) data and draws it on the screen, while the HBlank and VBlank periods are left open so that program code can modify background and obj data without risk of creating graphical artifacts.

Video Modes

Exactly what the GBA draws on screen depends largely on the current video mode (also sometimes referred to as the *screen mode* or *graphics mode*). The GBA has 6 such modes, some of which are bitmap-based and some of which are tile-based. The video mode is set by the bottom three bits of the hardware register known as REG_DISPCNT. Background data is handled differently depending on what mode is enabled. Backgrounds can either be text backgrounds (tile based), rotate-scale backgrounds (tile based backgrounds that can be transformed), or bitmap backgrounds. The starting address and size sprite graphics memory is also dependent on video mode. It starts at 0x6010000 for tile modes and 0x6014000 for bitmapped modes.

Enabling objs and one or more backgrounds in REG_DISPCNT will cause the GBA to draw the specified backgrounds and objs in order of priority.

Mode 0:
In this mode, four text background layers can be shown. In this mode backgrounds 0 - 3 all count as "text" backgrounds, and cannot be scaled or rotated. Check out the section on text backgrounds for details on this.

Mode 1:
This mode is similar in most respects to Mode 0, the main difference being that only 3 backgrounds are accessible -- 0, 1, and 2. Bgs 0 and 1 are text backgrounds, while bg 2 is a rotation/scaling background.

Mode 2:
Like modes 0 and 1, this uses tiled backgrounds. It uses backgrounds 2 and 3, both of which are rotate/scale backgrounds.

Mode 3:
Standard 16-bit bitmapped (non-paletted) 240x160 mode. The map starts at 0x06000000 and is 0x12C00 bytes long. See the Color Format table above for the format of these bytes.

This allows the full color range to be displayed at once. Unfortunately, the frame buffer in this mode is too large for page flipping to be possible. One option to get around this would be to copy a frame buffer from work RAM into VRAM during the retrace, or (so I have heard) to use DMA3 with the start mode bits set to 11.

Mode 4:
8-Bit paletted bitmapped mode at 240x160. The bitmap starts at either 0x06000000 or 0x0600A000, depending on bit 4 of REG_DISPCNT. Swapping the map and drawing in the one that isn't displayed allows for page flipping techniques to be used. The palette is at 0x5000000, and contains 256 16-bit color entries.

Mode 5:
This is another 16-bit bitmapped mode, but at a smaller resolution of 160x128. The display starts at the upper left hand corner of the screen, but can be shifted using the rotation and scaling registers for BG2. The advantage of using this mode is presumably that there are two frame buffers available, and this can be used to perform page flipping effects which cannot be done in mode 3 due to the smaller memory requirements of mode 5. Bit 4 of REG_DISPCNT sets the start of the frame buffer to 0x06000000 when bit 4 is zero, and 0x600A000 when bit 4 is one.

Color Format:

All colors (both paletted and bitmapped) are represented as a 16 bit value, using 5 bits for red, green, and blue, and ignoring bit 15. In the case of paletted memory, pixels in an image are represented as 8 bit or 4 bit indices into the palette RAM starting at 0x5000000 for backgrounds and 0x5000200 for sprites. Each palette is a table consisiting of 256 16-bit color entries. In the case of the bitmapped backgrounds in modes 3 and 4, pixels are represented as the 16-bit color values themselves.

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X B B B  B B G G  G G G R  R R R R

0-4 (R) = Red
5-9 (G) = Green
A-F (B) = Blue
```

5. Backgrounds

Depending on the current video mode, three different types of backgrounds are available. They are

Text Backgrounds:

These are tile-based backgrounds that descend from the usage of tiles to display characters in text modes of a PC or workstation. They are made up of 8x8 tiles, the bitmaps of which are stored at the tile data address. The address of this data is set using registers REG_BG0CNT - REG_BG3CNT. The HOFS and VOFS registers can be used to scroll around a larger area of up to 512x512 pixels (or 64 x 64 tiles).

In text backgrounds, the data for each pixel is stored as an 8 or 4 bit palette index. In 8-bit mode, the palette is at 0x05000000 stores a 15-bit color value for each of the 256 palette entries. In 4-bit mode, the the map index contains a 4-bit value indicating which of 16 16-color palettes to use for each tile. Each of these palettes is 32 bytes long and can be found at 0x05000000, 0x05000020, etc.

Scale/Rotate Backgrounds:

These backgrounds are also tile-based, and operate similarly to Text Backgrounds. However, these backgrounds may also be scaled or rotated. Additionally they may only use an 8-bit palette, and can vary in size from 128 to 1024 pixels across. The palette is at 0x5000000, and contains 256 16-bit color entries

Bitmapped Backgrounds:

These backgrounds vary depending on the video mode, but in all cases they rely on a single buffer upon which the image is drawn, either using an 8-bit palette or 16-bit color entries themsevles. Bitmap backgrounds are treated as BG2 for purposes of rotation, scaling, and blending. In the bitmap modes the frame buffer data extends into the obj tile data region, limiting it to the range from 0x6014000 - 0x6018000 (sprite indices 512 - 1024).

Background Map Entry Format

Text Background Map Format:
The tile map, which stores the layout of the tiles on screen, begins at the tile map address found for a particular background, detrmined by REG_BG0CNT - REG_BG3CNT. It has a selectable size up to 512x512. The tile map contains a 16-bit entry for each tile, with has the following format:

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
L L L L  V H T T  T T T T  T T T T

0-9 (T) = The tile number
A   (H) = If this bit is set, the tile is flipped horizontally left to right.
B   (V) = If this bit is set, the tile is flipped vertically upside down.
C-F (L) = Palette number
```

For 256 x 256 and 256 x 512 backgrounds, the formula for calculating a map index is roughly,

mapEntry = tileMapAddress[(tileY * 32) + tileX]

For text mode sizes 512 x 256 and 512 x 512 backgrounds, however, the map is 64 tiles across, and these are stored in blocks of 32 * 32 tiles. This means that to calculate the map entry that would appear 33 tiles or more across the background, the following equation should be used:

mapEntry = tileMap[(tileY * 32) + (tileX - 32) + 32*32]

For entries 33 tiles or more down (in mode 11), use

mapEntry = tileMap[((tileY-32) * 32) + tileX + 2*32*32];

And for entries 33 tiles or more down and 33 tiles or more across,

mapEntry = tileMap[((tileY-32) * 32) + (tileX-32) + 3*32*32];

Rotational Background Map Format :
This is the same idea as the text background map format, but you only have 8 bits for each entry. The format for the tile map entries is

```
7 6 5 4 3 2 1 0
T T T T T T T T

0-7 (T) = The tile number
```

Rotational backgrounds do not divide tile maps into blocks.

For specific details on the format of background data and map entries, check out the section on REG_BG0CNT - REG_BG3CNT (addresses 0x04000008 - 0x0400000E).

In all modes up to 128 sprites can be displayed as well as the 4 background layers. These use the second palette which is located at 0x05000200. See the OAM section for details on how to display sprites.

Both background tiles and sprites use palette entry 0 as the transparent color. Pixels in this color will not be drawn, and allow other background layers and sprites to show through.

6. OAM (sprites)

The GBA supports 128 simultaneous sprites. These can be up to 64x64 pixels in size. The OAM, which starts at 0x07000000, has one entry for each of the 128 sprites. Intermixed with this data are the rotation/scaling attributes, of which there are 32 sets of 4 16 bit values.

Each OAM entry is 8 bytes long and has the following format:

Bytes 1 and 2 (Attribute 0)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
S S A M  T T D R  J J J J  J J J J
```

0-7 (J) = Y co-ordinate of the sprite (pixels). Note that for regular sprites,
this is the y coordinate of the upper left corner. For rotate/scale
sprites, this is the y coordinate of the sprite's center. center .
Note on Coordinates: The values actually wrap around: to achieve a -1
y coordinate, use y = 255.

8 (R) = Rotation/Scaling on/off
9 (D) = 0 - sprite is single sized;
1 - sprite is virtually double sized; allowing sheared sprite pixels to
overflow sprite the size (specified by bits 14 - 15 of
OAM attribute 1). A 16x16 sized sprite is treated internaly as a
32x32 sprite. This specification comes in evidence when rotating a
sprite at 45°, since the H/V size of the sprite becomes
SQRT(16² + 16²) = SQRT(512) =~ 22.62 pixels. This will cause the
sprite to appear clipped if this bit is set to 0.
(Thanks to Kay for the description)

A-B (T) = 00 - normal
01 - semi-transparent
10 - obj window
11 - illegal code

Note that semi-transparent sprites appear as transparent even if
REG_BLDMOD has the sprites bit turned off. Also note that
sprites cannot be blended against one another. For more details, see
REG_BLDMOD.

C (M) = enables mosaic for this sprite.
D (A) = 256 color if on, 16 color if off
E-F (S) = See below

Bytes 3 and 4 (Attribute 1)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
S S V H  X X X I  I I I I  I I I I      (standard sprites)
S S F F  F F F I  I I I I  I I I I  (rotation/scaling on)
```

0-8 (I) = X coordinate of the sprite (pixels). For regular sprites,
this is the x coordinate of the upper left corner. For rotate/scale
sprites, this is the x coordinate of the sprite's center.
Note on coordinates: The values actually wrap around. To achieve a
-1 x, use x = 511.

C (H) = The flip horizinal bit
D (V) = The flip vertical bit

9-D (F) = For rotation scaling sprites, the index into the rotation data to
be used for that sprite. This index can be from 0 - 31. The
rotation/scaling data is located in OAM attribute 3 (bytes 7 and 8).
However, instead of the rotation and scaling data going with the
corresponding sprite, it is separated accross four sequential sprites.
This index can be thought of as referencing into an array of
four-sprite blocks, 32 bytes each.

E-F (S) = Size of the sprite. The top two bits of the size value are found in
attribute 0 and the bottom two bits
are in attribute 1. This forms a
4-bit value which sets the size of the sprite in the following way:

0000: 8 x 8 1000: 8 x 16
0001: 16 x 16 1001: 8 x 32
0010: 32 x 32 1010: 16 x 32

```
0011: 64 x 64        1011: 32 x 64
0100: 16 x 8         1100: Not used
0101: 32 x 8         1101: Not used
0110: 32 x 16        1110: Not used
0111: 64 x 32        1111: Not used
```

Bytes 5 and 6 (Attribute 2)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
L L L L  P P T T  T T T T  T T T T

0-9 (T) = Tile number. This value indexes selects the bitmap of the tile to be
          displayed by indexing into the tile data area. Each index refernces
          32 bytes, so the memory address of a tile is roughly 0x6010000 + T*32.
          (see Sprite Tile Data for details)

A-B (P) = Priority. This controls the priority of the sprite. Note thate sprites take
          precedence over backgrounds of the same priority.  See the description of
          priority under REG_BG0 - REG_BG3 for a more detailed explanation.

C-F (L) = Palette number. If you use 16 color palettes, this tells you which pallette
          number to use.
```

Bytes 7 and 8 (Attribute 3)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
S I I I  I I I I  F F F F  F F F F

0-7 (F) = Fraction.
8-E (I) = Integer
F = (S) = Sign bit
```

These bytes control sprite rotation and scaling. Instead of the rotation and scaling data going with the corresponding sprite, it is separated accross four sequential sprites. This is indexed by bits 9 - 13 in attribute 1 (bytes 3 and 4). Note that these are all relative to the center of the sprite (background rotation/scaling is relative to the upper left). Starting with sprite 0 and repeating every 4 sprites, they appear in the following order:

Sprite 0, Attribute 3 - PA (DX):
Scales the sprite in the x direction by an amount equal to 1/(register value). Thus, a value of 1.0 results in the original image size, while a value of 2 is half as large, and a value of .5 is twice as large.

Sprite 1, Attribute 3 - PB (DMX):
Shears the x coordinates of the sprite over y. A value of 0 will result in no shearing, a value of 1.00 will make the image appear to be sheared left going down the screen, and a value of -1 will make the image appear sheared right going down the screen.

Sprite 2, Attribute 3 - PC (DY):
Shears the y coordinates of the sprite over x. A value of 0 will result in no shearing, a value of 1.00 will make the image appear to be sheared upwards to the right, and a value of -1 will make the image appear sheared downwards and to the right.

Sprite 3, Attribute 3 - PD (DMY):
Scales the image in the y direction by an amount equal to 1/(register value). Thus, a value of 1.0

results in the original image size, while a value of 2 is half as large, and a value of .5 is twice as large.

To Make a Sprite Rotate and Scale: (Taken from PERN Tutorial Day 3)

The basic form of the equations for rotating and scaling is as follows:

```
pa = x_scale * cos(angle);
pb = y_scale * sin(angle);
pc = x_scale * -sin(angle);
pd = y_scale * cos(angle);
```

Sprite Tile Data

The tile data area contains the actual bitmap for each tile. The sprites do not share tile data with the BG layers as on the Gameboy Color. The sprite tile data starts at 0x06010000. All tiles are 8x8 pixels large. Sprites use the second palette which begins at 0x05000200. For 256 color sprites, there are 64 bytes per tile, one byte per pixel. This is an 8-bit value which is an index into the 256 color palette. For 16-color sprites, attribute 2 (bytes 5 and 6) of the OAM data contains a 4 bit index into 16 16 color palettes, and sprites have 32 bytes per tile, with 4 bits per pixel. Note that the tile index references 32 bytes at a time, so in the case of 256 color sprite tiles, you will want to set your tile number to reference ever other index (i.e. 0, 2, 4, 6, etc.). Another thing to note is that in the bitmapped modes (3-5) the memory required to hold background data is larger than 0x10000 bytes, forcing the GBA to cut away from available sprite tile data. Thus in these modes you may only reference sprites tiles of indices 512 and up.

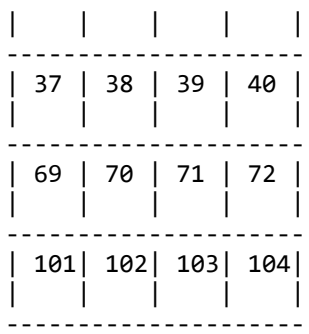
When the sprite is larger than 8x8 pixels, multiple tiles are glued together to make the sprite's width horizontally, and then vertically. How this is done depends on whether character data is stored in 2d or 1d mode (determined by bit 6 of DISPCNT).

1D Mapping:
In 1D mode, tiles are stored sequentially. If you were to set up a 32x32 16-color sprite, and set the tile number to 5, the sprite would be displayed as follows:

```
--------------------
| 5  | 6  | 7  | 8  |
|    |    |    |    |
--------------------
| 9  | 10 | 11 | 12 |
|    |    |    |    |
--------------------
| 13 | 14 | 15 | 16 |
|    |    |    |    |
--------------------
| 17 | 18 | 19 | 20 |
|    |    |    |    |
--------------------
```

2D Mapping:
Tiles on each row of the sprite are stored 32 slots in. Using the same 32x32 sprite above, with a tile number of 5, the sprite would be displayed as:

```
--------------------
| 5  | 6  | 7  | 8  |
```



7. Windowing

Windowing is a method of dividing the screen into subsections known as (surprise) windows. The windows serve as boundary areas to determine where various layers of the GBA will be shown and where they will be clipped. There are two primary windows, win0 and win1, which can be enabled in REG_DISPCNT. There is also the "obj" window, which can be thought of as another window which is defined by the visible regions of the objs on screen. Finally there is the "outside" or "out" window - the area of the screen not already occupied by any other winodw. The position and size of WIN0 and WIN1 are determined by REG_WIN0H, REG_WIN1H, REG_WIN0V, and REG_WIN1V (I/O offsets 0x40, 0x42, 0x44, 0x46). Exactly which characters and backgrounds appear within or without win0, win1, and the obj window is determined by REG_WININ and REG_WINOUT (0x48 and 0x4A).

Here are some things to keep in mind when using windows :

- WIN0 and WIN1 are drawn from the left and top boundary up to but not including the right and bottom boundaries.
- Everything in WIN0 appears "above" WIN1 (i.e. it has higher priority), and everything in windows 0 & 1 appears above the WINOUT and obj windows.
- If a bg or the obj's are turned off in dispcnt, they're off in all windows regardless of the settings in win_in and win_out.
- If only one window is on, WINOUT affects everything outside of it (if both windows are on, WINOUT affects everything outside both of them. IE, it affects (!WIN0)&&(!WIN1)).
- If a window is on, but the effective display bits are all clear, the backdrop is displayed.
- If the window left coordinate is greater than the window right coordinate, the window will be drawn outside of this region (i.e. to the left and to the right) rather than in the area inbetween.
- Likewise, if the window top coordinate is greater than the window bottom coordinate, the window will be drawn to the top and the bottom.
- A completely inverted window is drawn in the area outside of the "+" shaped region defined by its boundaries.

Windows can be used in console games for a variety of different effects. Though the window registers define a square region, differently shaped windows can be achieved by using hdma or hblank interrupts to change the parameters each scanline. Lantern lighting (when the hero has a lantern or flashlight that illuminates a certain region of a cave) and x-ray vision (use of the window to cut away layers that are in front) are two common effects created with windows. More are certainly possible.

Thanks again to gbcft for most of these notes and for his extensive testing on the nature of windowing.

8. Hardware Interrupts

Figuring out hardware interrupts was kind of painful. Everything below is what I have gleaned from reading ARM's docs, the list, the advice of other emulator and demo authors, and from various other emulator's debug info. I hope it is of some use to you. Let me know if you find any errors or typos.

Key points:
- All hardware interrupt vectors lie in the BIOS. You cannot handle interrupts directly, you must go through the BIOS. Thus, the instructions for exception handling in the ARM docs do not apply directly since we cannot handle the exceptions directly.

- Interrupts are enabled by setting the flags in the REG_IE and hardware registers like REG_DISPSTAT, REG_KEYCNT, and REG_DMAXCNT. The flag must be set in both REG_IE and the corresponding hardware register for it to work. When the interrupt signal is sent, the appropriate flag is set in REG_IF. The program code unsets this flag (by writing a 1 to that bit) in order to keep track of what interrupts have been handled.

- When an interrupt occurs, the CPU does the following:

1. Switches state to IRQ mode, bank-swaps the current stack register and link register (thus preserving their old values), saves the CPSR in SPSR_irq, and sets bit 7 (interrupt disable) in the CPSR.
2. Saves the address of the next instruction in LR_irq compensating for Thumb/ARM depending on the mode you are in.
3. Switches to ARM state, executes code in BIOS at a hardware interrupt vector (which you, the programmer, never see)

- The BIOS code picks up at the hardware interrupt vector and does the following:

4. Pushes registers 0 - 3, 12, LR_irq (which cointains the address following the instruction when the interrupt occrued) onto the stack
5. Places the address for the next instruction (in the BIOS, not in your code) in LR
6. Loads the address found at 0x03007FFC
7. Branches to that address.

- The program code at that address is executed.

8. It is the responsiblity of the code at that address to return once finished, using BX LR_irq

- The BIOS finishes up where your code leaves off:

9. It restores registers 0 - 3, 12, LR_irq
10. Branches to the intruction found in LR, using a SUBS PC, LR_irq, #4

- Upon receiving the SUBS PC, LR_irq, #4 instruction, the CPU

11. Copies the SPSR_irq back into the CPSR, restoring the status bits to their state when the interrupt occurred, and bank swaps back in the stack register and link register. The CPU will thus be placed in the correct state (ARM or Thumb) it was in when the exception occurred.

So, the basic model for setting up interrupts is:

1. Place the address for your interrupt code at 0x03007FFC.

2. Turn on the interrupts you wish to use:

- REG_DISPSTAT, REG_TMXCNT, REG_KEYCNT, or REG_DMAXCNT tell the hardware which interrupts to send
- 0x04000200 (REG_IE) masks which interrupts will actually be serviced (?)
- 0x04000208 (REG_IME) Turns all interrupts on or off.

3. When the interrupt is reached, the code at the address at 0x3007FFC gets loaded into the CPU. To prevent unwanted errors/behavior, the first thing this code should do is disable interrupts.

4. To determine what interrupt this is, check the flags in 0x04000202 (REG_IF). Unset the flag by writing a 1 to that bit.

5. Once finished with the service routine, reenable interrupts and execute a BX LR (*Not* a SUBS PC, LR #4, which is what the BIOS does). The BIOS will then take over and return your program to where execution left off.

Types of Hardware Interrupts

Enable these interrupts using REG_DISPSTAT, REG_TMXCNT, REG_KEYCNT, or REG_DMAXCNT, then setting the correct flags in REG_IE and REG_IME.

V-Blank: Occurs when the vcount reaches 160, or 0xA0. (Enable in REG_DISPSTAT)

H-Blank: Occurs at the end of every raster line, from 0 - 228. H-blank interrupts DO occur during v-blank (unlike hdma, which does not), so write your code accordingly. Thanks to gbcft for verifying this. (Enable in REG_DISPSTAT)

Serial: I am unsure about this; I presume it has to do with the link cable.

V-Count: Occurs when the vcount reaches the number specified in REG_DISPSTAT.

Timer: These occur whenever one of the timer registers is set to cause an interrupt whenever it overflows. Enable in REG_TMXCNT.

DMA: These occur after a DMA transfer, according to the flags in the DMA_CNT registers and in REG_IE. Enable in REG_DMAXCNT.

Key: Occurs when the user presses or releases the buttons specified in REG_KEYCNT.

Cassette: Occurs when the user yanks out or inserts the cartridge out while the GBA is still running. For a cartridge interrupt to work properly the ISR must reside in RAM. It is possible to switch cartridges and have the routine resume execution on a completely different ROM.

9. BIOS (Software Interrupts - partially implemented in CowBite)

The BIOS calls are basically SWI instructions; the value passed into the instruction tells the CPU which interrupt to execute.
The most significant portions of this section come from Martin Korth's very detailed description of the BIOS functions in his no$gba documentation. Thanks also to Forgotten's call list to his Visual Boy Advance FAQ.

Note that the BIOS SWI handler does not perform range checking, so calling SWI 43 - 255 (0x2B - 0xFF) will cause a lock up.

0x00: SoftReset

Resets the GBA and runs the code at address 0x2000000 or 0x8000000 depending on the contents of 0x3007ffa (0 means 0x8000000 and anything else means 0x2000000).

0x01: RegisterRamReset

Performs a selective reset of memory and I/O registers.
Input: r0 = reset flags

0x02: Halt

Halts CPU execution until an interrupt occurs.

0x03: Stop

Stops the CPU and LCD until the enabled interrupt (keypad, cartridge or serial) occurs.

0x04: IntrWait

Waits for the given interrupt to happen.
Input: r0 = initial flag clear, r1 = interrupt to wait

0x05: VBlankIntrWait

Waits for vblank to occur. Waits based on interrupt rather than polling in order to save battery power.
Equivalent of calling IntrWait with r0=1 and r1=1.

0x06: Div

Input: r0 = numerator, r1 = denominator
Output: r0 = numerator/denominator
r1 = numerator % denominator;
r3 = abs (numerator/denominator)

0x07: DivArm

Input: r0 = denominator, r1 = numerator
Output: r0 = numerator/denominator
r1 = numerator % denominator;
r3 = abs (numerator/denominator)
Note: For compatibility with ARM's library only. Slightly slower than SWI 6.

0x08: Sqrt

Input: r0 = unsigned 32-bit number
Output: r0 = sqrt(number) (unsigned 32-bit integer)

0x09: ArcTan

Input: r0 = Tangent(angle) (16-bit; 1 bit sign, 1 bit integral, 14 bit decimal)
Output: r0 = "-PI/2<THETA/<PI/2" in a range of 0xC000h-0x4000.
Note: There is a problem in accuracy with "THETA<-PI/4, PI/4<THETA"

0x0A: ArcTan2

Note: I'm unsure about this one, since there is conflicting info about its purpose, and I have not tested it.
Version 1:
Calculates the arctangent of the given point.
Input: r0 = X (signed 16-bit), r1 = Y (signed 16-bit)
Output: r0=arctan

Version 2:
Calculates the arctangent after correction processing.
Input: r0 = Tangent(angle) (16-bit; 1 bit sign, 1 bit integral, 14 bit decimal)
Output: r0 = 0000h-FFFFh for 0<=THETA<2PI

0x0B: CPUSet

Performs a memory transfer.
Input: r0 = source address, r1 = dest address
r2 = Length and mode:
bit 24 = Fixed source address (0 - copy, 1 = set with word at r0)
bit 26 = 32 or 16 bit transfer
bits 15 - 0 = number of transfers

0x0C: CPUFastSet

Also performs a memory transfer, in 32-byte blocks, using LDMIA/STMIA instructions
Performs a memory transfer.
Input: r0 = source address, r1 = dest address
r2 = Length and mode:
bit 24 = Fixed source address (0 - copy, 1 = set with word at r0)
bits 15 - 0 = number of transfers

0x0D: BiosChecksum

Calculates the checksum of the whole BIOS by adding every 32-bit word from the BIOS.
Output: r0 = BIOS checksum
Note: this seems to only be present in the release version of the BIOS

0x0E: BgAffineSet

Calculates the affine parameters for bgs (rotation and scaling).
Input: r0 = source, r1 = dest, r2 = number of calculations

Where the data at the source is an array of r2 structures of the format:

```
typdef struct tBGAffineSource {
     s32 x;     //Original data's center X coordinate (8bit fractional portion)
     s32 y;     //Original data's center Y coordinate (8bit fractional portion)
     s16 tX;    //Display's center X coordinate
     s16 tY;    //Display's center Y coordinate
     s16 sX;    //Scaling ratio in X direction (8bit fractional portion)
     s16 sY;    //Scaling ratio in Y direction (8bit fractional portion)
     u16 theta; //Angle of rotation (8bit fractional portion) Effective Range 0-FFFF
} BGAffineSource;
```

And the return data (placed at the dest address) will be of the format:

```
typdef struct tBGAffineDest {
     s16 pa;  //Difference in X coordinate along same line
     s16 pb;  //Difference in X coordinate along next line
     s16 pc;  //Difference in Y coordinate along same line
     s16 pd;  //Difference in Y coordinate along next line
     s32 x;   //Start X coordinate
     s32 y;   //Start Y coordinate
} BGAffineDest;
```

(from no$gba docs)

0x0F: ObjAffineSet

Calculates and sets the OBJ's affine parameters from the scaling ratio and angle of rotation.
The four affine parameters are set every offset bytes, starting from the dest pointer address.
If the offset value is 2, the parameters are stored contiguously. If the value is 8, they match the structure of OAM.

Input: r0 = source, r1 = dest, r2 = number of calculations, r3 = Offset in bytes for parameter addresses (2=contigiously, 8=OAM)
Where the data at the source address is of the format:

```
typedef struct tObjAffineSource {
     s16 sX;    //Scaling ratio in X direction (8bit fractional portion)
     s16 sY;    //Scaling ratio in Y direction (8bit fractional portion)
     u16 theta;     //Angle of rotation (8bit fractional portion) Effective Range 0-FFFF
} ObjAffineSource;
```

And the data at the destination address is of the format:

```
typedef struct tObjAffineDest {
     s16 pa;  //Difference in X coordinate along same line
     s16 pb;  //Difference in X coordinate along next line
     s16 pc;  //Difference in Y coordinate along same line
     s16 pd;  //Difference in Y coordinate along next line
} ObjAffineDest;
```

0x10: BitUnPack

Unpacks bit packed data.
Input: r0 = source, r1 = dest, r2 = pointer to unpack parameters

Where the unpack parameters are of the format:

```
typedef struct tBUP {
    u16 sourceLength;     //Length of Source Data in bytes (0-0xFFFF)
    u8 sourceWidth;       //Width of Source Units in bits (only 1,2,4,8 supported)
    u8 destWidth;         //Width of Destination Units in bits (only 1,2,4,8,16,32 supported)
    u32 destOffset;       //31-bit Data Offset (Bit 0-30), and Zero Data Flag (Bit 31)
} BUP;
```

The Data Offset is always added to all non-zero source units.
If the Zero Data Flag was set, it is also added to zero units.

Data is written in 32-bit units, Destination can be Wram or Vram. The size of unpacked data must be a multiple of 4 bytes.
The width of source units (plus the offset) should not exceed the destination width.

0x11: LZ77UnCompWRAM

Uncompresses LZSS data 8 bits at a time
Input: r0 = source address, r1 = dest address

0x12: LZ77UnCompVRAM

Uncompresses LZSS data 16 bits at a time
Input: r0 = source address, r1 = dest address

Note: The LZ77 decompressors actually decompress LZSS, not LZ77, which is slightly different. You will have to look on the web to find the algorithm as it is beyond the scope of this document. The following assumes a general famliarity with LZSS.

On the GBA, the ring buffer or "window" is of size 4096, the minumim compressed length is 3 and the maximum compressed length is 18. Looking into a compressed buffer you will find the size of the uncompressed memory in bytes 2, 3, and 4 (I'm not sure what the first byte does, but it seems to always be set to "01"), followed by the coded data. This is divided up into sections consisting of an 8 bit key followed by a corresponding eight items of varying size. The upper bits in the key correspond to the items with lower addresses and vice versa. For each bit set in the key, the corresponding item will be 16 bits; the top bits four being the number of bytes to output, minus 3, and the bottom sixteen bits being the offset behind the current window position from which to output. For each bit

which is not set, the corresponding item is an uncompressed byte and gets sent to the output.

Thanks to Markus for providing me with some source that helped me figure out all of this.

0x13: HuffUnComp

Unpacks data compressed with Huffman and writes it 32-bits at a time.
Input: r0 = source address, r1 = dest address
Where the data at the source address is of the format:

Data Header (32bit)
Bit 0-3 Data size in bit units (normally 4 or 8)
Bit 4-7 Compressed type (must be 2 for Huffman)
Bit 8-31 24bit size of decompressed data in bytes

Tree Table
u8 tree table size/2-1

Each of the nodes below defined as:

```
7 6 5 3 3 2 1 0
L R 0 0 0 0 0 0
```

0 - 5 (O) = offset to next node -1 (2 byte units)
6 (R) = Right node end flag (if set, data is in next node)
7 (L) = Left node end flag

1 node Root node
2 nodes Left, and Right node
4 nodes LeftLeft, LeftRight, RightLeft, and RightRight node
- Compressed data

0x14: RLUnCompWRAM

Uncompresses RLE data 8 bits at a time
Input: r0 = source address, r1 = dest address

0x15: RLUnCompVRAM

Uncompresses RLE data 16 bits at a time (slower)
Input: r0 = source address, r1 = dest address

The data at the source address for both RLU functions is of the format:
Data header (32-bit)
Bit 0-3 Reserved
Bit 4-7 Compressed type (must be 3 for run-length)
Bit 8-31 Size of decompressed data

Repeat below. Each Flag Byte followed by one or more Data Bytes. Flag data (8bit)
Bit 0-6 Expanded Data Length (uncompressed N-1, compressed N-3)
Bit 7 Flag (0=uncompressed, 1=compressed)

Data Byte(s) - N uncompressed bytes, or 1 byte repeated N times

0x16: Diff8bitUnFilterWRAM

Unpacks data filtered with 8-bit difference and writes it 8-bits at a time.
Input: r0 = source, r1 = dest

0x17: Diff8bitUnFilterVRAM

Unpacks data filtered with 8-bit difference and writes it 16-bits at a time.
Input: r0 = source, r1 = dest

0x18: Diff16bitUnFilter

Unpacks data filtered with 16-bit difference and writes it 16-bits at a time.
Input: r0 = source, r1 = dest

In each UnFilter function the format of the dest is:
Data Header (32-bit)
Bit 0-3 Data size (must be 1 for Diff8bit, 2 for Diff16bit)
Bit 4-7 Type (must be 8 for DiffFiltered)
Bit 8-31 24bit size after decompression

Data Units (each 8bit or 16bit depending on used SWI function)
Data0 //original data
Data1-Data0 //difference data
Data2-Data1
Data3-Data2

0x19: SoundBiasChange

Sets the sound bias from 0 to 0x200 or from 0x200 to 0 depending on the value of R0.
Input: r0 = 0 to set it to 0, other values to set it to 0x200

0x1A: SoundDriverInit

Initializes the built in sound driver. Do not initialize more than once.
Input: r0 = SoundArea

Note: This has been abridged from the description in the no$gba documentation. I suggest checking there for a more thorough description.

The SoundArea Structure has the following format:

```
typedef struct tSoundArea {
     u32 ident;           //Flag the system checks to see if the work area has been
                          //initialized/currently being accessed.
     vu8 DmaCount;        //User access prohibited
     u8  reverb;          //Variable for applying reverb effects to direct sound
     u16 d1;              //User access prohibited
     void (*func)();      //User access prohibited
     int intp;            //User access prohibited
     void* NoUse;         //User access prohibited
     SoundChannel vchn[MAX];     //Array of Direct Sound channel structures.
     s8 pcmbuf[PCM_BF*2]; //??
} SoundArea;

typedef struct tSoundChannel {
     u8 sf;     //Channel status.  When 0 sound it is stopped.  To start sound, set other
                //parameters and then write 0x80h.  Logical OR 0x40 for a key-off, or
                //write zero for a pause. The use of other bits is prohibited.
     u8 r1;     //User access prohibited
     u8 rv;     //Sound volume output to right side
     u8 lv;     //Sound volume output to left side
     u8 at;     //The attack value of the envelope.
     u8 de;     //The decay value of the envelope.
     u8 su;     //The sustain value of the envelope.
     u8 re;     //The release value of the envelope. Key-off to enter this state.
     u8 r2[4];  //User access prohibited
```

```
        u32 fr;    //The frequency of the produced sound.
        WaveData*  wp;     //Pointer to the sound's waveform data.
        u32 r3[6];  //  User access prohibited
        u8 r4[4];   //User access prohibited
    } SoundChannel;

    typedef struct tWaveData {
        u16 type;   //Indicates the data type. This is currently not used.
        u16 stat;   //0 - non-looped 0x4000 - forward loop
        u32 freq;   //Frequency.
        u32 loop;   //Loop pointer (start of loop)
        u32 size;   //Number of samples (end position)
        s8 data[];  //The actual waveform data. Takes (number of samples+1)bytes of 8bit
                    //signed linear uncompressed data. The last byte is zero for a non-looped
                    //waveform, and the same value as the loop pointer data for a looped waveform.
    } WaveData;
```

0x1B: SoundDriverMode

Sets the operation of the built in sound driver.
Input: r0 = operation mode

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
X  X  X  X   R  R  R  R   R  R  R  R   F  F  F  F   V  V  V  V   C  C  C C  S R R R  R R R R

0-6   (R) = Direct Sound Reverb value (0-127, default=0) (ignored if Bit7=0)
7     (S) = Direct Sound Reverb set (0=ignore, 1=apply reverb value)
8-11  (C) = Direct Sound Simultaneously-produced (1-12 channels, default 8)
12-15 (V) = Direct Sound Master volume (1-15, default 15)
16-19 (F) = Direct Sound Playback Frequency (1-12 = 5734,7884,10512,13379,
                                             15768,18157,21024,26758,31536,
                                             36314,40137,42048, def 4=13379 Hz)
20-23 (D) = Final number of D/A converter bits (8-11 = 9-6bits, def. 9=8bits)
```

0x1C: SoundDriverMain

Main function of the built in sound driver that is called by applications after SoundDriverVSync to render the sound. Does not have to be called at a precise 60hz interval, but must be called once during every 60hz period.

0x1D: SoundDriverVSync

Call this function at the very start of vblank (or, presumably, any other 60hz increment) to sync the BIOS sound driver.

0x1E: SoundChannelClear
0x1F: MIDIKey2Freq
Ox20: MusicPlayerOpen
0x21: MusicPlayerStart
0x22: MusicPlayerStop
0x23: MusicPlayerContinue
0x24: MusicPlayerFadeOut
0x25: MultiBoot
0x26: HardReset (Undocumented)
0x27: CustomHalt (Undocumented)
0x28: SoundDriverVSyncOff

Call this function in the event that the SoundDriverVSync function misses the 60hz interval (for example, when loading new data at the beginning of a stage).. It will keep the sound DMA from overflowing and playing noise.

0x29: SoundDriverVSyncOn

Call this after calling SoundDriverVSyncOff, to restore sound driver operation.

0x2A: GetJumpList (for sound?) (Undocumented)

?: FIQMasterEnable

10. Memory-Mapped Hardware Registers

The following section describes the function of each of the memory-mapped addresses in IO RAM. The register naming scheme is based on a variant of the popular gba.h by Eloist (specifically, that used by Uze in the examples on his Audio Advance site).

The notation for each entry is as follows:

```
                          R  <- 'R' means "Read Only", 'W' means "Write Only"
F E D C  B A 9 8  7 6 5 4  3 2 1 0   <- These are the bits
W V U S  L K J I  F D B A  C M M M   <- These letters are used in the key
                                    Entries marked with an 'X' usually
                                    serve no function, are unwriteable,
                                    and remain at 0.

0-2 (M)   = The video mode. See video modes list above. <- The key for each bit field
```

Addresses: 0x04000000 - 0x4000054 - Graphics Hardware Registers

Address: 0x4000000 - REG_DISPCNT (The display control register)

```
                          R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
W V U S  L K J I  F D B A  C M M M

0-2 (M) = The video mode. See video modes list above.
3   (C) = Game Boy Color mode. Read only - should stay at 0.
4   (A) = This bit controls the starting address of the bitmap in bitmapped modes
          and is used for page flipping. See the description of the specific
          video mode for details.
5   (B) = Force processing during hblank. Setting this causes the display
          controller to process data earlier and longer, beginning from the end of
          the previous scanline up to the end of the current one. This added
          processing time can help prevent flickering when there are too many
          sprites on a scanline.
6   (D) = Sets whether sprites stored in VRAM use 1 dimension or 2.
          1 - 1d: tiles are are stored sequentially
          0 - 2d: each row of tiles is stored 32 x 64 bytes in from the start of the
          previous row.

7   (F) = Force the display to go blank when set. This can be used to save power
          when the display isn't needed, or to blank the screen when it is being
          built up (such as in mode 3, which has only one framebuffer). On the SNES,
          transfers rates to VRAM were improved during a forced blank; it is logical
          to assume that this would also hold true on the GBA.

8   (I) = If set, enable display of BG0.
```

```
9   (J) = If set, enable display of BG1.
A   (K) = If set, enable display of BG2.
B   (L) = If set, enable display of BG3.
C   (S) = If set, enable display of OAM (sprites).
D   (U) = Enable Window 0
E   (V) = Enable Window 1
F   (W) = Enable Sprite Windows
```

Address: 0x4000004 - REG_DISPSTAT

```
                            R R R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
T T T T  T T T T  X X Y H  V Z G W

0   (W) = V Refresh status. This will be 0 during VDraw, and 1 during VBlank.
          VDraw lasts for 160 scanlines; VBlank follows after that and lasts 68
          scanlines. Checking this is one alternative to checking REG_VCOUNT.

1   (G) = H Refresh status. This will be 0 during HDraw, and 1 during HBlank HDraw
          lasts for approximately 1004 cycles; HBlank follows, and lasts
          approximately 228 cycles, though the time and length of HBlank may in
          fact vary based on the number of sprites and on rotation/scaling/blending
          effects being performed on the current line.

2   (Z) = VCount Triggered Status. Gets set to 1 when a Y trigger interrupt occurs.
3   (V) = Enables LCD's VBlank IRQ. This interrupt goes off at the start of VBlank.
4   (H) = Enables LCD's HBlank IRQ. This interrupt goes off at the start of HBlank.
5   (Y) = Enable VCount trigger IRQ. Goes off when VCount line trigger is reached.
8-F (T) = Vcount line trigger. Set this to the VCount value you wish to trigger an
          interrupt.
```

Address: 0x4000006 - LCY / REG_VCOUNT (Read Only)

This location stores the current y location of the LCD hardware. It is incremented as the lines are drawn. The 160 lines of display are followed by 68 lines of Vblank period, before the whole thing starts again for the next frame. Waiting for this register to reach 160 is one way to synchronize a program to 60Hz.

Addresses: 0x400008 - 0x400001E - Background Registers

Address: 0x4000008 - REG_BG0CNT
Address: 0x400000A - REG_BG1CNT
Address: 0x400000C - REG_BG2CNT
Address: 0x400000E - REG_BG3CNT

These addresses set up the four background layers. The format is:

```
    ?
F E D C  B A 9 8  7 6 5 4  3 2 1 0
Z Z V M  M M M M  A C X X  S S P P

0-1 (P) = Priority - 00 highest, 11 lowest
          Priorities are ordered as follows:
```

```
          "Front"
          1. Sprite with priority 0
          2. BG with     priority 0

          3. Sprite with priority 1
          4. BG with     priority 1

          5. Sprite with priority 2
          6. BG with     priority 2

          7. Sprite with priority 3
          8. BG with     priority 3

          9. Backdrop
          "Back"

          When multiple backgrounds have the same priority, the order
          from front to back is:  BG0, BG1, BG2, BG3.  Sprites of the same
          priority are ordered similarly, with the first sprite in OAM
          appearing in front.

2-3 (S) = Starting address of character tile data
          Address = 0x6000000 + S * 0x4000
6   (C) = Mosiac effect - 1 on, 0 off

7   (A) = Color palette type -
          1 - standard 256 color pallete
          0 - each tile uses one of 16 different 16 color palettes (no effect on
              rotates/scale backgrounds, which are always 256 color)

8-C (M) = Starting address of character tile map
          Address = 0x6000000 + M * 0x800
D   (V) = Screen Over. Used to determine whether rotational backgrounds get tiled
          repeatedly at the edges or are displayed as a single "tile" with the area
          outside transparent. This is forced to 0 (read only) for backgrounds
          0 and 1 (only).

E-F (Z) = Size of tile map
          For "text" backgrounds:
          00 : 256x256 (32x32 tiles)
          01 : 512x256 (64x32 tiles)
          10 : 256x512 (32x64 tiles)
          11 : 512x512 (64x64 tiles)

          For rotational backgrounds:
          00 : 128x128 (16x16 tiles)
          01 : 256x256 (32x32 tiles)
          10 : 512x512 (64x64 tiles)
          11 : 1024x1024 (128x128 tiles)
```

Address: 0x4000010 - REG_BG0HOFS Horizontal scroll co-ordinate for BG0 (Write Only)
Address: 0x4000012 - REG_BG0VOFS Vertical scroll co-ordinate for BG0 (Write Only)
Address: 0x4000014 - REG_BG1HOFS Horizontal scroll co-ordinate for BG1 (Write Only)
Address: 0x4000016 - REG_BG1VOFS Vertical scroll co-ordinate for BG1 (Write Only)
Address: 0x4000018 - REG_BG2HOFS Horizontal scroll co-ordinate for BG2 (Write Only)
Address: 0x400001A - REG_BG2VOFS Vertical scroll co-ordinate for BG2 (Write Only)
Address: 0x400001C - REG_BG3HOFS Horizontal scroll co-ordinate for BG3 (Write Only)
Address: 0x400001E - REG_BG3VOFS Vertical scroll co-ordinate for BG3 (Write Only)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X S S  S S S S  S S S S
```

```
0-9 (S) = Scroll value (pixels)
```

These registers are only effective for text backgrounds; they set the pixel that is displayed in the top left hand corner of the GBA's display. In other words, a value of -5, -5 puts the upper left hand corner of your background at x=5,y=5. All four BG planes wrap when they reach their right or bottom edges.

Addresses: 0x4000020 - 0x4000026 / 0x4000030 - 0x4000036 - Background Rotation/Scaling Registers (Write Only)

These registers affect the translation, rotation, and scaling of tile-based rotate/scale backgrounds as well as the bitmapped backgrounds (which should be treated as BG2 for this purpose). The function of these registers is very hard to describe in words but easy to see the effects of on screen. I highly recommend checking out Stephen Stair's RSDemo - it lets you see the contents of the regs as you modify them as well as the effect they have on the background. Should also be somewhat useful for figuring out sprite rotation and scaling.

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
S I I I  I I I I  F F F F  F F F F

0-7 (F) = Fraction
8-E (I) = Integer
F   (S) = Sign bit
```

These registers apply only to Rotate/Scale backgrounds. Individual descriptions follow:

Address: 0x4000020 - REG_BG2PA (BG2 Read Source Pixel X Increment)(Write Only)
Address: 0x4000030 - REG_BG3PA (BG3 Read Source Pixel X Increment) (Write Only)

The effect of these registers is to scale the background (relative to the upper left corner) in the x direction by an amount equal to 1/(register value).

Address: 0x4000022 - REG_BG2PB (BG2 Write Destination Pixel X Increment) (Write Only)
Address: 0x4000032 - REG_BG3PB (BG3 Write Destination Pixel X Increment) (Write Only)

The effect of these registers is to shear the x coordinates of the background over y, relative to the upper left corner. A value of 0 will result in no shearing, a value of 1.00 will make the background appear to be sheared left as you go down the screen, and a value of -1 will make the background appear sheared right as you go down the screen.

Address: 0x4000024 - REG_BG2PC (BG2 Read Source Pixel Y Increment) (Write Only)
Address: 0x4000034 - REG_BG3PC (BG3 Read Source Pixel Y Increment) (Write Only)

The effect of these registers is to shear the y coordinates of the background over x, relative to the upper left corner. A value of 0 will result in no shearing, a value of 1.00 will make the background appear to be sheared upwards to the right, and a value of -1 will make the background appear sheared downwards and to the right.

Address: 0x4000026 - REG_BG2PD (BG2 Write Destination Pixel Y Increment) (Write Only)
Address: 0x4000036 - REG_BG3PD (BG3 Write Destination Pixel Y Increment) (Write Only)

The effect of these registers is to scale the background in the y direction (relative to the upper left corner) by an

amount equal to 1/(register value).

Address: 0x4000028 - REG_BG2X (X Coordinate for BG2 Rotational Background)(Write Only)
Address: 0x4000038 - REG_BG3X (X Coordinate for BG3 Rotational Background)(Write Only)

Address: 0x400002C - REG_BG2Y (Y Coordinate for BG2 Rotational Background)(Write Only)
Address: 0x400003C - REG_BG3Y (Y Coordinate for BG3 Rotational Background)(Write Only)

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
X  X  X  X   S  I  I  I   I  I  I  I   I  I  I  I   I  I  I  I   I  I  I I  F F F F  F F F F

0-7  (F) - Fraction
8-26 (I) - Integer
27   (S) - Sign bit
```

These registers define the location of the pixel that appears at 0,0. They are very similar to the background scrolling registers, REG_HOFS and REG_VOFS, which become disabled when a rotate/ scale background is in use.

Addresses: 0x4000040 - 0x400004A - Windowing Registers

Address: 0x4000040 - REG_WIN0H (Window 0 X Coordinates) (Write Only)
Address: 0x4000042 - REG_WIN1H (Window 1 X Coordinates)(Write Only)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
L L L L  L L L L  R R R R  R R R R

0-7 (R) = X coordinate for the rightmost side of the window
8-F (L) = X coordinate for the leftmost side of the window
```

Address: 0x4000044 - REG_WIN0V (Window 0 Y Coordinates) (Write Only)
Address: 0x4000046 - REG_WIN1V (Window 1 Y Coordinates)(Write Only)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
T T T T  T T T T  B B B B  B B B B

0-7 (B) = Y coordinate for the bottom of the window
8-F (T) = Y coordinate for the top of the window
```

Address: 0x4000048 - REG_WININ (Inside Window Settings)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X T S  R Q P O  X X L K  J I H G

0 (G) = BG0 in win0
1 (H) = BG1 in win0
2 (I) = BG2 in win0
3 (J) = BG3 in win0
4 (K) = Sprites in win0
5 (L) = Blends in win0
```

```
8 (O) = BG0 in win1
9 (P) = BG1 in win1
A (Q) = BG2 in win1
B (R) = BG3 in win1
C (S) = Sprites in win1
D (T) = Blends in win1
```

Address: 0x400004A - REG_WINOUT (Outside Window and Sprite Window)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X T S  R Q P O  X X L K  J I H G

0 (G) = BG0 outside
1 (H) = BG1 outside
2 (I) = BG2 outside
3 (J) = BG3 outside
4 (K) = Sprites outside
5 (L) = Blends outside
8 (O) = BG0 in sprite win
9 (P) = BG1 in sprite win
A (Q) = BG2 in sprite win
B (R) = BG3 in sprite win
C (S) = Sprites in sprite win
D (T) = Blends in sprite win
```

Addresses: 0x400004C - 0x4000054 - Effects Registers

Address: 0x400004C - REG_MOSAIC (Write Only)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
V V V V  U U U U  J J J J  I I I I

0-3 (I) = BG X Size
4-7 (J) = BG Y Size
8-B (U) = Sprite X Size
C-F (V) = Sprite Y Size
```

Use this register to control the size of the mosaic on backgrounds/sprites that have mosaic enabled..

Address: 0x4000050 - REG_BLDMOD

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X T S  R Q P O  M M L K  J I H G

0   (G) = Blend BG0 (source)
1   (H) = Blend Bg1 (source)
2   (I) = Blend BG2 (source)
3   (J) = Blend BG3 (source)
4   (K) = Blend sprites (source)
5   (L) = Blend backdrop (source)
6-7 (M) = Blend Mode
```

```
          There are four different modes:
          00: All effects off
          01: alpha blend
          10: lighten (fade to white)
          11: darken (fade to black)

8   (O) = Blend BG0 (target)
9   (P) = Blend BG1 (target)
A   (Q) = Blend BG2 (target)
B   (R) = Blend BG3 (target)
C   (S) = Blend sprites (target)
D   (T) = Blend backdrop (target)
```

Use this register to determine the blending mode and which layer(s) you wish to perform blending on. In the case of alpha blends (Mode 01), specify the layers that are "on top" using the source flags (bits 0 - 5) and the layers that are on the bottom using the target flags (bits 8-13). The target layer must be below the source layer in terms of its priority, or the blend will not take effect. Other things to note about alpha blends:

- If there is more than one target layer, the blend will only occur for a target with lower priority in areas where it shows through targets of higher priority due to the transparent pixel being set
- Source layers will only blend with areas of a target layer that are visible beneath them. If another layer is blocking the way (even if it is another source layer), there will be no blend and the original source color will be drawn.
- As a result of these two conditions, it is never possible for any given pixel to be a blend of more than 2 layers. This eliminates the possiblity of using these registers to have 3 or more layers of translucent graphics showing through one another.
- A layer cannot blend with itself

- If an obj has semi-transparency enabled, it will blend normally (as if it were specified as a source layer)
- Unfortunately, it is not possible to alpha blend sprites against one another, no matter how your prioritize them. Alpha blended sprites that are "in front of" other sprites will blend with the other target layers while still occluding the sprites behind them (i.e. it will look like the portion of the non-blended sprite that is behind the blended one has disappeared), for a most unnatural effect.

Address: 0x4000052 - REG_COLEV (Write Only)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X B  B B B B  X X X A  A A A A

0-4 (A) = Coefficient A, the source pixel (layer above)
8-C (B) = Coefficient B, the target pixel (layer below)
```

Use this in conjunction with REG_BLDCNT to determine the amount of blending between layers. An unblended pixel of normal intensity is is considered to have a coefficient of 16. Coefficient A and Coefficient B determine the ratio of each of the sources that will get mixed into the final image. Thus, if A is 12 and B is 4, the resulting image will appear to be 12/16 the color of A and 4/16 the color of B. Note that A and B can add up to be greater than 16 (for an additive or brightening effect) or less than 16 (for a darkening effect).

Address: 0x4000054 - REG_COLEY (Write Only)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  X X X F  F F F F
```

```
0-4 (F) = The lighten/darken value
```

This is the amount by which to lighten or darken the source layers (as specified in REG_BLDCNT) .
The higher the value, the greater the fade. 16 is the peak fade value; values from 16 - 31 shade the layer with either pure black (for a darken) or pure white (for a lighten).

Addresses 0x040000060 - 0x0400000A6 (Sound Controls: Partially unimplemented in CowBite)

Note: I've obtained this info (most of it verbatim) from Uze's BeLogic unofficial GBA sound info site, which gives a much more thorough explanation as well as some sample source code and demos. Thanks to Uze for providing such a great resource on GBA sound.

Address: 0x04000060 - REG_SOUND1CNT_L (Sound 1 Sweep control) (Unimplemented in CowBite)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  X T T T  A S S S

0-2 (S) = Number of sweep shifts. These control the amount of change in
          frequency (either increase or decrease) at each change. The wave's new
          period is given by: T=T±T/(2^n), where n is the sweep shift's value.

3   (A) = Sweep increase or decrease. When decrementing, if the frequency value
          gets smaller than zero, the previous value is retained. When
          incrementing, if the frequency gets greater than the maximum frequency
          (131Khz or 2048 for the register value) the sound stops.
          0 - Addition (frequency increase)
          1 - Subtraction (frequency decrease)

4-6 (T) = Sweep Time. This is the delay between sweep shifts. After each delay,
          the frequency increments or decrements.
          000:  Disable sweep function
          001:  Ts=1 / 128khz (7.8 ms)
          010:  Ts=2 / 128khz (15.6 ms)
          011:  Ts=3 / 128 khz (23.4 ms)
          100:  Ts=4 / 128 khz (31.3 ms)
          101:  Ts=5 / 128 khz (39.1 ms)
          110:  Ts=6 / 128 khz (46.9 ms)
          111:  Ts=7 / 128 khz (54.7 ms)
```

Sound channel 1 produces a square wave with envelope and frequency sweep functions. This register controls the frequency sweep function. When the sweep function is not required, set the sweep time to zero and set the increase/decrease bit to 1.

Address: 0x04000062 - REG_SOUND1CNT_H (Sound 1 Length, wave duty and envelope control) (Unimplemented in CowBite)

```
                    W W  W W W W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
I I I I  M T T T  D D L L  L L L L
```

0-5 (L) = Sound length. This is a 6 bit value obtained from the following formula: Sound length= (64-register value)*(1/256) seconds. After the sound length has been changed, the sound channel must be resetted via bit F of REG_SOUND1CNT_X (when using timed mode).

6-7 (D) = Wave duty cycle. This controls the percentage of the ON state of the square wave.

00 - 12.5%
01 - 25%
10 - 50%
11 - 75%

8-A (T) = Envelope step time. This is the delay between successive envelope increase or decrease. It is given by the following formula: Time=register value*(1/64) seconds.

B (M) = Envelope mode. Controls if the envelope is to increase or decrease in volume over time.

0 - Envelope decreases
1 - Envelope increases

C-F (I) = Initial Envelope value. 1111 produces the maximum volume and 0000 mutes the sound. When sound 1 is playing, modifying the volume envelope bits has no effect until the sound is resetted.

Address: 0x04000064 - REG_SOUND1CNT_X (Sound 1 Frequency, reset and loop control) (Unimplemented in CowBite)

```
W           W W W  W W W W  W W W W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
R T X X  X F F F  F F F F  F F F F
```

0-A (F) = Sound frequency. The minimum frequency is 64Hz and the maximum is 131Khz. Can be calculated from the following formula: F(hz)=4194304/(32*(2048-register value)).

E (T) = Timed sound. When set to 0, sound 1 is played continuously regardless of the length data in REG_SOUND1CNT_H. When set to 1, sound is played for that specified length and after that, bit 0 of REG_SOUNDCNT_X is reset.

F (R) = Sound reset. When set, sound resets and restarts at the specified frequency. When sound 1 is playing, modifying the volume envelope bits has no effect until the sound is resetted. Frequency and sound reset must be perfomed in a single write since both are write only. Frequency can always be changed without resetting the sound channel.

Address: 0x04000068 - REG_SOUND2CNT_L (Sound 2 Length, wave duty and envelope control) (Unimplemented in CowBite)

```
                  W W  W W W W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
I I I I  M T T T  D D L L  L L L L
```

0-5 (L) = Sound length. This is a 6 bit value obtained from the following formula: Sound length= (64-register value)*(1/256) seconds. After the sound length has been changed, the sound channel must be resetted via bit F of REG_SOUND1CNT_X (when using timed mode).

```
6-7 (D) = Wave duty cycle. This controls the percentage of the ON state of
          the square wave.

          00 - 12.5%
          01 - 25%
          10 - 50%
          11 - 75%

8-A (T) = Envelope step time. This is the delay between successive envelope increase
          or decrease. It is given by the following formula: Time=register value*(1/64)
          seconds.

B   (M) = Envelope mode. Controls if the envelope is to increase or decrease in volume
          over time.

          0 - Envelope decreases
          1 - Envelope increases

C-F (I) = Initial Envelope value. 1111 produces the maximum volume and 0000 mutes
          the sound. When sound 2 is playing, modifying the volume envelope bits has
          no effect until the sound is resetted.
```

Address: 0x0400006C- REG_SOUND2CNT_H (Sound 2 Frequency, reset and loop control) (Unimplemented in CowBite)

```
W           W W W  W W W W  W W W W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
R T X X  X F F F  F F F F  F F F F

0-A (F) = Sound frequency. The minimum frequency is 64Hz and the maximum is
          131Khz. Can be calculated from the following formula:
          F(hz)=4194304/(32*(2048-register value)).

E   (T) = Timed sound. When set to 0, sound 2 is played continuously regardless of
          the length data in REG_SOUND2CNT_L. When set to 1, sound is played for
          that specified length and after that, bit 1 of REG_SOUNDCNT_X is reset.

F   (R) = Sound reset. When set, sound resets and restarts at the specified
          frequency. When sound 2 is playing, modifying the volume envelope bits
          has no effect until the sound is resetted. Frequency and sound reset must
          be perfomed in a single write since both are write only. Frequency can
          always be changed without resetting the sound channel.
```

Address: 0x04000070 - REG_SOUND3CNT_L (Sound 3 Enable and wave ram bank control) (Unimplemented in CowBite)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  N S M X  X X X X


5 (M) = Bank Mode (0 - 2 32 sample banks, 1 - 1 64 sample bank)
6 (S) = Bank Select. Controls which bank is active for playing/reloading. If set
        to 0, samples are played from bank 0 and writing to the Wave Ram will
        store the data in Bank 1, and vice-versa.
7 (N) = Sound Channel 3 output enable. When this is set and bit 15 from
        REG_SOUND3CNT_X is set, the sound starts to play.
```

Sound channel 3 is a circuit that can produce an arbitrary wave pattern. Samples are 4 bit, 8 samples per word, and are located in Wave Ram registers from 0x400090 to 0x40009F. The Wave Ram is banked, providing the

ability to play a 64 samples pattern or to select between two 32 samples patterns (Bit 5). Sound channel 3 always produces some audio artifacts (distortion) when sound is initialized. Fortunately, switching banks does not require re-initialisation during playback, thus allowing for dynamic reloading of the Wave Ram without generating any distortion.

Both banks of Wave Ram are filled with zero upon initialization of the Gameboy, Bank 0 being selected. So writing to bank 0 implies setting bit 6 to 1 before loading Wave Ram then set it back to 0 to play it.

Address: 0x04000072 - REG_SOUND3CNT_H (Sound 3 Sound length and output level control) (Unimplemented in CowBite)

```
                  W W W W  W W W W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
R R R X  X X X X  L L L L  L L L L


0-7 (L) = Sound length. The sound length is an 8 bit value obtained from the
          following formula:  Register=Note length (in seconds)*256
          Hence a 1 second maximum and a 3.9 millisecond minimum sound duration.
          After the sound length has be changed, the sound channel must be
          resetted via bit F of REG_SOUND3CNT_X.

D-F (R) = Output volume ratio:

          000 - Mute
          001 - 100%
          100 - 75%
          010 - 50%
          011 - 25%
```

Address: 0x04000074 - REG_SOUND3CNT_X (Sound 3 Frequency, reset and loop control) (Unimplemented in CowBite)

```
W           W W W  W W W W  W W W W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
R T X X  X F F F  F F F F  F F F F

0-A (F) = Sound frequency. The minimum frequency is 64Hz and the maximum is
          131Khz. Can be calculated from the following formula:
          F(hz)=4194304/(32*(2048-register value)).

E (T) = Timed sound. When set to 0, sound 3 is played continuously regardless of
        the length data in REG_SOUND3CNT_H. When set to 1, sound is played for
        that specified length and after that, bit 2 of REG_SOUNDCNT_X is reset.

F (R) = Sound reset. When set, sound resets and restarts at the specified
        frequency. Frequency and sound reset must be perfomed in a single write
        since both are write only. In continuous mode, frequency can be changed
        without resetting the sound channel.
```

Address: 0x04000078 - REG_SOUND4CNT_L (Sound 4 Length, output level and envelope control) (Unimplemented in CowBite)

```
                    W W  W W W W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
I I I I  M T T T  X X L L  L L L L
```

0-5 (L) = Sound length. This is a 6 bit value obtained from the following formula:
Sound length= (64-register value)*(1/256) seconds. After the sound length
has been changed, the sound channel must be resetted via bit F of
REG_SOUND4CNT_H (when using timed mode).
8-A (T) = Envelope step time. This is the delay between successive envelope
increase or decrease. It is given by the following formula:
Time=register value*(1/64) seconds.
B (M) = Envelope mode. Controls if the envelope is to increase or decrease in volume
over time.

0 - Envelope decreases
1 - Envelope increases

D-F (I) = Initial Envelope value. 1111 produces the maximum volume and 0000 mutes
the sound.

Address: 0x0400007C - REG_SOUND4CNT_H (Sound 4 Noise parameters, reset and loop control) (Unimplemented in CowBite)

```
W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
R T X X  X X X X  P P P P  S C C C
```

0-2 (C) = Clock divider frequency. This divides the CPU frequency. Its output is
then fed into the counter's pre-scaler (controlled by bits 4-7) which
further devides the frequency.

000: f*2 f=4.194304 Mhz/8
001: f
010: f/2
011: f/3
100: f/4
101: f/5
110: f/6
111: f/7

3 (S) = Counter stages: 0=15 stages, 1=7 stages. This controls the period of the
polynomial counter. It is given by (2^n)-1 where n is the number of
stages. So for n=7, the pseudo-noise period lasts 63 input clocks.
After that, the counter restarts the same count sequence.
4-7 (P) = Counter Pre-Stepper frequency:

0000: Q/2
0001: Q/2^2
0010: Q/2^3
0011: Q/2^4
....
1101: Q/2^14
1110: Not used
1111: Not used
Where Q is the clock divider's output frequency

E (T) = Timed sound. When set to 0, sound 4 is played continuously regardless of
the length data in REG_SOUND4CNT_L. When set to 1, sound is played for that
specified length and after that, bit 3 of REG_SOUNDCNT_X is reset.
F (R) = Sound reset. When bit F is set to 1, Envelope is set to initial value,
the LFSR count sequence is resetted and the sound restarts. In continuous
mode, all parameters can be changed but the sound needs to be resetted when
modifying the envelope initial volume or the clock divider for changes to
take effects.

Channel 4 produces pseudo-noise generated by a polynomial counter. It is based on a 7/15 stages linear-feedback shift register (LFSR). LFSR counts in a pseudo-random order where each state is generated once and only once during the whole count sequence. The sound is produced by the least significant bit's output stage.

Address: 0x04000080 - REG_SOUNDCNT_L (Sound 1-4 Output level and Stereo control) (Unimplemented in CowBite)

```
                          ?
F E D C  B A 9 8  7 6 5 4  3 2 1 0
R Q P O  N M L K  J I I I  H G G G

0-2 (G) = DMG Left Volume
3   (H) = Vin Left on/off (?) - According to BeLogic, Vin on/off allowed the
          original GameBoy paks to provide their own sound source. It is unkown
          whether they still work on a GBA.
4-6 (I) = DMG Right Volume
7   (J) = Vin Right on/off (?)
8   (K) = DMG Sound 1 to left output
9   (L) = DMG Sound 2 to left output
A   (M) = DMG Sound 3 to left output
B   (N) = DMG Sound 4 to left output
C   (O) = DMG Sound 1 to right output
D   (P) = DMG Sound 2 to right output
E   (Q) = DMG Sound 3 to right output
F   (R) = DMG Sound 4 to right output
```

This register controls only the DMG output amplifiers and have no effects on the individual sound channels' processing, or Direct Sound channels' volume.

Address: 0x04000082 - REG_SOUNDCNT_H (Direct Sound control and Sound 1-4 output ratio) (Implemented in CowBite)

```
W       W
F E D C  B A 9 8  7 6 5 4  3 2 1 0
Q P O N  M L K J  X X X X  I H G G

0-1 (G) = Output Sound Ratio for channels 1-4.

          00 - 25%
          01 - 50%
          10 - 100%
          11 - ??

2   (H) = Direct sound A output ratio (0 - 50%, 1 - 100%)
3   (I) = Direct sound B output ratio (0 - 50%, 1 - 100%)
8   (J) = Direct Sound A to right output
9   (K) = Direct sound A to left output
A   (L) = Direct sound A Sampling rate timer (timer 0 or 1). Use this to set which
          timer contorls the playback frequency.
B   (M) = Direct sound A FIFO reset
C   (N) = Direct sound B to right output
D   (O) = Direct sound B to left output
E   (P) = Direct sound B Sampling rate timer (timer 0 or 1). Use this to set
          which timer controls the playback frequency.
F   (Q) = Direct sound B FIFO reset
```

This register is used in controlling Direct Sound on the GBA. Output ratios control the volume, in percentage, that gets output to the speakers.

Address: 0x04000084 - REG_SOUNDCNT_X (Master sound enable and Sound 1-4 play status)(Partially Implemented in CowBite)

```
                        R R R R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  N X X X  J I H G

0 (G) = DMG Sound 1 Status (Read only). 0 - Stopped, 1 - Playing
1 (H) = DMG Sound 2 Status (Read only). 0 - Stopped, 1 - Playing
2 (I) = DMG Sound 3 Status (Read only). 0 - Stopped, 1 - Playing
3 (J) = DMG Sound 4 Status (Read only). 0 - Stopped, 1 - Playing
7 (N) = All Sound circuit enable
```

This register is used to monitor the play status of sounds and to turn on or off all sound circuits. Turning the sound circuits off saves battery power, allowing them to last up to 10% longer.

Address: 0x04000088 - REG_SOUNDBIAS (Sound bias and Amplitude resolution control) (Unimplemented in CowBite)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
R R X X  X X B B  B B B B  B B B X

1-9 (B) = PWM bias value, controlled by the BIOS.
E-F (R) = Amplitude resolutions
          00 - 9 bit at 32768 hz
          01 - 8 bit at 65536 hz
          10 - 7 bit at 131072 hz
          11 - 6 bit at 262144 hz
```

The BIAS setting is used to offset the sound output and bring it back into a signed range. When the BIOS starts up, it runs a timing loop where it slowly raises the BIAS voltage from 0 to 512. This setting should not be changed. At best, the sound will become distorted. At worst the amplifier inside the GBA could be damaged.

When accessing bits F-E, a read-modify-write is required. The default value for bits F-E is 00. Most if not all games use 01 for this setting.

Address: 0x04000090 - REG_WAVE_RAM0_L (Sound 3 samples 0-3)
Address: 0x04000092 - REG_WAVE_RAM0_H (Sound 3 samples 4-7)
Address: 0x04000094 - REG_WAVE_RAM1_L (Sound 3 samples 8-11)
Address: 0x04000096 - REG_WAVE_RAM1_H (Sound 3 samples 12-15)
Address: 0x04000098 - REG_WAVE_RAM2_L (Sound 3 samples 16-19)
Address: 0x0400009A - REG_WAVE_RAM2_H (Sound 3 samples 20-23)
Address: 0x0400009C - REG_WAVE_RAM3_L (Sound 3 samples 23-27)
Address: 0x0400009E - REG_WAVE_RAM3_H (Sound 3 samples 28-31)

These registers together contain four (4 bytes each) 4-bit wave RAM samples for Sound channel 3.

Address: 0x040000A0 - REG_FIFO_A_L (Direct Sound channel A samples 0-1)(Write Only)
Address: 0x040000A2 - REG_FIFO_A_H (Direct Sound channel A samples 2-3)(Write Only)

Address: 0x040000A4 - REG_FIFO_B_L (Direct Sound channel B samples 0-1)(Write Only)
Address: 0x040000A6 - REG_FIFO_B_H (Direct Sound channel B samples 2-3)(Write Only)

These are the locations of the Direct Sound 8-bit FIFO samples, from which Direct Sound pulls the music data to be played on the speakers. Note that there are only 8 bytes total for all your samples. You repeatedly fill these from a buffer of your own using DMA0 or DMA1, or by using timer interrupts.

To fill them using DMA, first set Timer 0 or Timer 1 to refresh at the appropriate sample rate (for example, 16khz). Next, set the DMA source address to a sound sample in memory, and the destination address to one of these FIFO registers. Use REG_SOUNTCNT_H to reset FIFO and tell Direct Sound to get its sampling rate from Timer 0 or Timer 1. Finally, set the DMA control register to start on FIFO empty (start mode 11) and to repeat, then enable the timers. All of this will cause the hardware to play sound samples in FIFO at the rate specified in your timer, and automatically refill them using DMA.

To fill these using interrupts, follow a similar process, but instead of using DMA, set the clock to interrupt on overflow. When using interrupts instead of DMA, BeLogic recommends setting the timer divider to 1024 and start the timer at 0xFFFF order to get a sampling rate of 16.384 khz. This apparently causes less distortion than if you simply set the start time of the clock to 0xFFFF - (2^24/16000).

Note that reading from these registers can yield unpredictable results. It might be interesting to see just *how* unpredictable...

Addresses: 0x40000B0, 0x40000BC, 0x40000C8, 0x40000D4 (DMA Source Registers) (Write Only)

Address: 0x40000B0 - REG_DMA0SAD (DMA0 Source Address)(Write Only)

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
X  X  X  X   X  A  A  A   A  A  A  A   A  A  A  A   A  A  A  A   A  A  A A  A A A A  A A A A

0-26 (A) = 27-bit source address
```

This is the source address for DMA channel 0 transfers. Note that it is 27-bit.

Address: 0x40000BC - REG_DMA1SAD (DMA1 Source Address)
Address: 0x40000C8 - REG_DMA2SAD (DMA2 Source Address)
Address: 0x40000D4 - REG_DMA3SAD (DMA3 Source Address)

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
X  X  X  X   A  A  A  A   A  A  A  A   A  A  A  A   A  A  A  A   A  A  A A  A A A A  A A A A

0-27 (A) = 28-bit source address
```

This is the source address for DMA channel 1, 2, or 3 transfers. Note that it is 28-bit.

Addresses: 0x40000B4, 0x40000C0, 0x40000CC, 0x40000D8 (DMA Destination Registers) (Write Only)

Address: 0x40000B4 - REG_DMA0DAD (DMA0 Destination Address)
Address: 0x40000C0 - REG_DMA1DAD (DMA1 Destination Address)
Address: 0x40000CC - REG_DMA2DAD (DMA2 Destination Address)

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
X  X  X  X   X  A  A  A   A  A  A  A   A  A  A  A   A  A  A  A   A  A  A A  A A A A  A A A A

0-27 (A) = 27-bit destination address
```

This is the dest address for DMA channel 0, 1, and 2 transfers. Note that it is 27-bit.

Address: 0x40000D8 - REG_DMA3DAD (DMA3 Destination Address)(Write Only)

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
X  X  X  X   A  A  A  A   A  A  A  A   A  A  A  A   A  A  A  A   A  A  A A  A A A A  A A A A

0-27 (A) = 28-bit destination address
```

This is the dest address for DMA channel 3 transfers. Note that it is 28-bit.

Addresses: 0x40000B8, 0x40000C4, 0x40000D0, 0x40000DC (DMA Count Registers)(Write Only)

Address: 0x40000B8 - REG_DMA0CNT_L (DMA0 Count Register)
Address: 0x40000C4 - REG_DMA1CNT_L (DMA1 Count Register)
Address: 0x40000D0 - REG_DMA2CNT_L (DMA2 Count Register)
Address: 0x40000DC - REG_DMA3CNT_L (DMA3 Count Register)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X L L  L L L L  L L L L  L L L L

0-D (L) = Number of words or halfwords to copy
```

(Note: In some places you will see the DMA control and DMA count registers depicted as a single 32-bit register called REG_DMAXCNT. I opted to treat them as two 16-bit registers for sake of clarity.)

Addresses: 0x40000BA, 0x40000C6, 0x40000D2, 0x40000DE (DMA Control Registers)

Address: 0x40000BA - REG_DMA0CNT_H (DMA0 Control Register)
Address: 0x40000C6 -REG_DMA1CNT_H (DMA1 Control Register)
Address: 0x40000D2 - REG_DMA2CNT_H (DMA2 Control Register)
Address: 0x40000DE - REG_DMA3CNT_H (DMA3 Control Register)

```
         ?
F E D C  B A 9 8  7 6 5 4  3 2 1 0
N I M M  U S R A  A B B X  X X X X


(Note: In some places you will see the DMA control and DMA count registers
depicted as a single 32-bit register called REG_DMAXCNT. I opted to treat
them as two 16-bit registers for sake of clarity.)

5-6 (B) = Type of increment applied to destination address. If enabled, the
          address will be incremented/decremented by 2 or 4 bytes, depending on
          the selected size. When the DMA is activated, the contents of these
          registers are copied to internal counters in the DMA hardware, which
          then increments/decrements these registers during transfer, preserving
          the contents of the IORAM registers.*

          00: Increment after each copy
```

```
          01: Decrement after each copy
          10: Leave unchanged
          11: Increment without after each copy, reset to initial value at the
              end of transfer (or at the end of the current repetition)

7-8 (A) = Type of increment applied to source address:
          00: Increment after each copy
          01: Decrement after each copy
          10: Leave unchanged
          11: Illegal

          Note: I am somewhat uncertain about option "11" for both of these.
          Can anyone confirm?

9   (R) = Repeat. When in start modes 1 or 2, this bit causes the transfer to
          repeat for each interval.
A   (S) = Size. If set, copy 32-bit quantities (words) If clear, copy 16-bit
          quantities (half words)
B   (U) = Unknown.  For DMA 0, 1, and 2, this bit is read only and set to 0. However,
          for DMA 3, it appears to be writeable. Thoughts, anyone?
C-D (M) = Start Mode.

          00: Transfer immediately
          01: Transfer on vblank (i.e. vdma)
          10: Transfer on hblank (i.e. hdma.  Note that, unlike h-interrupts,
                                  hdma does NOT occur during vblank.)
          11: The function of this varies based on the DMA channel.

              For DMA 1 or 2: Instructs the DMA to repeat on FIFO-empty requests.
              When this is set the size and count are ignored and a single 32 bit
              quantity is transferred on FIFO empty.
              For DMA 3: Apparently allows transfers to start at the beginning of a
              rendering line, copying data into a buffer as the line is being drawn
              on the screen. Useful for flicker-free transfers in mode 3, which has
              no backbuffer.

E   (I) = IRQ. Setting this bit causes the DMA to generate an interrupt when it is
          done with the data transfer.
F   (N) = Set this bit to enable DMA operation. Clear to end DMA operation.
```

This address controls a DMA transfer which allows large amounts of data to be transferred from one area of memory to another. It is theoretically twice as fast as transfering by the CPU, which uses at least one cycle for a read instruction and another for a write. DMA can also be used to clear memory to a constant value, if the source address is not incremented with each copy. Fist, set the DMASAD and DMADAD registers to point to the addresses you want. Writing to DMACNT_H address with a '1' in the N field and a '00' in the M field will start the transfer immediately.

DMA transfers may occur on an interrupt if the start mode bits are set for this. DMAs have a priority ranking with 3 at the lowest and 0 at the highest. For most cases, program code will be using DMA3 as it is lowest priority, allowing it to be interrupted by more important DMA (see below).

Specific DMAs have the following properties:

DMA0: This DMA is the highester priority, but cannot be used to access cartridge memory (addresses 0x8000000 and higher). It is suitable for time-critical operations such as transfering scale and rotate data to the background scaling registers. Since it takes precedence over other DMAs, it will not be postponed or interrupted (possibly causing undesirable results such as screen artifacts).

DMA1 and **DMA2**: These are the only DMA that can be used for sound FIFO. If start mode "11" is set, the DMA will be triggered on FIFO empty. I believe that FIFO A always sends its empty requests to DMA1 and that

FIFO B sends its empty requests only to DMA2, though I don't have any verification of this.

DMA3: This is is the lowest priority and thus often used as a "general purpose" DMA. Using this DMA for your basic memory transfers ensures that sound FIFO DMA and other time-critical DMA are not delayed, making audio or visual artifacts less likely.

* (Originally I had assumed a direct mapping between the source/destination registers and the current transfer address, and thus this section of the doc distinguished between transfers which wrote-back to the registers and those which did not. This appears to have been an incorrect assumption, and was brought to light as I delved further into sound emulation)

DMA Transfer Ratings

The following table lists the cycle timings for various DMA transfers. The format of each entry is :

```
16 bit DMA / 32 bit DMA
```

Units are in *cycles per item transfered*. Thus, a rating of 4/8 indicates that the transfer takes 4 cycles for every 16 bits transferred with 16 bit DMA, or 8 cycles for every 32 bits transfered with 32 bit DMA.

Source	Destination					
	EWRAM	IWRAM	IO	PAL RAM	VRAM	OAM
ROM 0 WS	4/8	2/3	2/3	2/4	2/4	2/3
ROM 1 WS	5/10	3/5	3/5	3/6	3/6	3/5
ROM 2 WS	6/12	4/7	4/7	4/8	4/8	4/7
EWRAM	6/12	4/7	4/7	4/8	4/8	4/7
IWRAM	4/7	2/2	2/2	2/3	2/3	2/2
I/O	4/7	2/2	2/2	2/3	2/3	2/2
PAL RAM	4/8	2/2	2/3	2/4	2/4	2/2
VRAM	4/8	2/3	2/3	2/4	2/4	2/2
OAM	4/7	2/2	2/2	2/3	2/3	2/2

Note that it is not possible to DMA transfer from or to SRAM (Cart RAM) or BIOS, and (obviously) it is not possible to transfer to ROM.

Thanks to Kay for supplying these transfer statistics!!

Addresses: 0x4000100 - 0x400010E (Timer registers)

Address: 0x4000100 - REG_TM0D (Timer 0 Data)
Address: 0x4000104 - REG_TM1D (Timer 1 Data)
Address: 0x4000108 - REG_TM2D (Timer 2 Data)
Address: 0x400010C - REG_TM3D (Timer 3 Data)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
D D D D  D D D D  D D D D  D D D D

0-F (D) = Current count of the timer.
```

Note that these registers are R/W. The default is to start counting from 0x0000, but if a value is written to this

register, the timer will henceforth use that as a starting value. Thus the rate at which timers overflow and generate interrupts (see REG_TMXCNT, below) can be customized.

Timer 0 and Timer 1 are used to control the rate of Direct Sound FIFO. When using DMA with start mode 11, they can automatically cause it to refill the FIFO. To set the rate of playback in hz, write the value 0xFFFF - (2^24/Plaback Freq in hz) to the register. This sets the start value such that the timer will overflow precisely when the next sound sample is needed, and cause the DMA to activate. When using interrupts, set the start value of these to 0, but use REG_TMXCNT to change the update frequency to 1024, thus causing an interrupt rate of 16.384khz.

Address: 0x4000102 - REG_TM0CNT (Timer 0 Control)
Address: 0x4000106 - REG_TM1CNT (Timer 1 Control)
Address: 0x400010A - REG_TM2CNT (Timer 2 Control)
Address: 0x400010E - REG_TM3CNT (Timer 3 Control)

```
                              *
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  E I X X  X C F F

0-1 (F) = Frequency at which the timer updates.

          00: Default frequency (full) - 16.78MHz (~17mlns ticks per second)
          01: Every 64 clock pulses - ~262187.5KHz
          10: Every 256 clock pulses - ~65546.875KHz
          11: Every 1024 clock pulses - ~16386.71875KHz

2   (C) = Cascade (* Unused on TM0) - When this bit is set, the frequency of this
          timer is ignored. Instead the timer increments when the timer below it
          overflows. For example, if timer 1 is set to cascade, it will increment
          whenever timer 0's value goes from 0xFFFF to 0x0000.
6   (I) = Generate an interrupt on overflow
7   (E) = Enable the timer.
```

Addresses 0x4000120 - 0x400012A - Serial Communication Registers) (Unimplemented in CowBite)

Using the link port and a link cable, the GBA can transmit serial data in one of five modes: Normal, Multilink, JOY BUS, General-Purpose, and UART. The function of the serial communication registers changes significantly depending on the mode. Thus each of these registers comes has several different discriptions associated with it, listed under the associated mode.

Note: The serial comm information originates primarily from Martin Korth's no$gba documentation, which is the most thorough explanation of GBA serial comm that I've read anywhere. My original info on Multiplayer mode originates from Andrew May's description of the GBA linker hardware, which ePAc for discovered and put into a format consistent with the rest of this spec

Address: 0x4000120 - REG_SCD0
Address: 0x4000122 - REG_SCD1
Address: 0x4000124 - REG_SCD2
Address: 0x4000126 - REG_SCD3

Normal Mode (0x4000120 and 0x4000122 only)

```
31 30 29 28  27 26 25 24  23 22 21 20  19 18 17 16  15 14 13 12  11 10 9 8  7 6 5 4  3 2 1 0
D  D  D  D   D  D  D  D   D  D  D  D   D  D  D  D   D  D  D  D   D  D  D D  D D D D  D D D D

0-31 (D) = The 32-bit data to sent/received over the link cable.
```

For 32-bit normal mode only. Contains 32-bit data. Outgoing data should be written to this register before starting the transfer. During transfer, transmitted bits are shifted-out (MSB first), and received bits are shifted-in simultaneously. Upon transfer completion, the register contains the received 32-bit value.

Multi-Player Mode

```
R R R R  R R R R  R R R R  R R R R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
D D D D  D D D D  D D D D  D D D D

0-F (D) = The data received.
```

- SCD0 contains the data sent by the master (also called slave 0)
- SCD1 contains the data sent by the first slave (slave1)
- SCD2 contains the data sent by the second slave (slave2)
- SCD3 contains the data sent by the last slave (slave3)

These registers are automatically reset ot 0xFFFF on transfer start. After transfer, they contain the incoming data from all linked GBAs (including data from the local machine).

Address: 0x4000128 - REG_SCCNT_L (Serial Communication channel control register)

Normal Mode
Multiplayer Mode
UART Mode

Normal Mode

```
                          R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X I M M  X X X X  S X X X  N L C B

0   (B) = Shift Clock (SC) (0 - External, 1 - Internal)
1   (C) = Internal Shift Clock (0 - 256khz, 1 - 2 MHz)
2   (L) = SI State (opponents SO) (0 - Low, 1 - High/None)
3   (N) = SO during inactivity (0 - Low, 1 - High)
7   (S) = Start/Busy bit (0 - inactive/ready, 1 - start/active)
C-D (M) = Comm Mode [8bit=00,32bit=01,Multiplayer=10,UART=11]
E   (I) = Interrupt on completion
```

The following is from Martin Korth's no$gba docs:

Recommended Communication Procedure for SLAVE unit (external clock)
- Initialize data which is to be sent to master (placed in REG_SCCNT_H for 8-bit mode or REG_SCD0 for 32-bit mode)
- Set Start bit (bit 7)
- Set SO to LOW to indicate that master may start now.
- Wait for IRQ (or for Start bit to become zero). (Check timeout here!)
- Set SO to HIGH to indicate that we are not ready.
- Process received data (received in REG_SCCNT_H for 8-bit mode or REG_SCD0 for 32-bit mode)

Recommended Communication Procedure for MASTER unit (internal clock)
- Initialize data which is to be sent to slave (placed in REG_SCCNT_H for 8-bit mode or REG_SCD0 for 32-bit mode)
- Wait for SI to become LOW (slave ready). (Check timeout here!)
- Set Start flag.
- Wait for IRQ (or for Start bit to become zero).
- Process received data (received in REG_SCCNT_H for 8-bit mode or REG_SCD0 for 32-bit mode)

Martin also notes that that when more than two GBAs are connected, data is rotated from the first to the last GBA rather than just being swapped between the first and second in the chain. For a more detailed description, see the help files included in no$gba.

Multi-Player Mode

```
                  * R R R  R R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X I M M  X X X X  S E D D  N L B B

0-1 (B) = Baud rate [00=9600,01=38400,10=57600,11=115200]
2   (L) = SI-Terminal (0 - Parent, 1- Child)
3   (N) = SD-Terminal (0 - Bad connection, 1 - All GBAs Ready)
4-5 (D) = ID of GBA [00=master,01=slave1,10=slave2,11=slave3]
6   (E) = Error (1 on error)
7   (S) = Start/Busy bit (1 triggers the start on the master, signifies busy on slaves)
          * Read only on slaves
C-D (M) = Comm Mode [8bit=00,32bit=01,Multiplayer=10,UART=11]
E   (I) = Interrupt on completion
```

To transfer data in this mode, you must coordinate the actions of all the GBAs which are linked together. First set the mode bits in REG_RCNT and in this register. Each GBA slave must place the data they wish transfered in REG_SCCNT_H. Then the Master/Slave 0 initiates the transfer by setting bit 7 of REG_SCCNT_L. This causes the hardware to transfer the data. It will magically appear in the destination registers of each slave, according to the following:

REG_SCCNT_H from GBA with id 00 goes into REG_SCD0 on each GBA
REG_SCCNT_H from GBA with id 01 goes into REG_SCD1 on each GBA
REG_SCCNT_H from GBA with id 10 goes into REG_SCD2 on each GBA
REG_SCCNT_H from GBA with id 11 goes into REG_SCD3 on each GBA

Thus each GBA in the chain has a duplicate of the data.

ePAc has commented that the master is the GBA in the set that has the purple connector connected to its ext port. So if you have a GBA that want to be a MBserver for a set of clients, then you need to put the cart in the one with the purple connector. It is unclear to me how the other GBAs know what ID they are; perhaps this is also set according to the link cable connector?

UART Mode

```
                  R R R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X I M M  U T A F  S E R S  P C B B

0-1 (B) = Baud rate [00=9600,01=38400,10=57600,11=115200]
2   (C) = CTS Flag (0 - Send always, 1- Send only when SC = LOW)
```

```
3   (P) = Parity Control (0 - Even, 1 - Odd)
4   (S) = Send Data Flag (0 - Not full, 1 - Full)
5   (R) = Receive Data Flag (0 - Not empty, 1 - Empty)
6   (E) = Error (1 on error)
7   (L) = Data Length (0 - 7 bits, 1 - 8 bits)
8   (F) = FIFO Enable Flag
9   (A) = Parity Enable Flag
10  (T) = Send Enable Flag
11  (U) = Receieve Enable Flag
C-D (M) = Comm Mode [8bit=00,32bit=01,Multiplayer=10,UART=11]
E   (I) = Interrupt (0 - Disable, 1 - IRQ when bits 4,5, or 6 become set)
```

To send data in UART mode:
The receiver outputs SD=LOW (which is input as SC=LOW at the remote side) when it is ready to receive data (that is, when Receive Enable is set, and the Receive shift register (or receive FIFO) isn't full.
When CTS flag is set to always/blindly, then the sender transmits data immediately when Send Enable is set, otherwise data is transmitted only when Send Enable is set and SC is LOW.

The error flag is set when a bad stop bit has been received (stop bit must be 0), when a parity error has occured (if enabled), or when new data has been completely received while the receive data register (or receive FIFO) is already full.

The error flag is automatically reset when reading from this register.

Init & Initback
The content of the FIFO is reset when FIFO is disabled in UART mode, thus, when entering UART mode initially set FIFO=disabled.
The Send/Receive enable bits must be reset before switching from UART mode into another SIO mode!

JOY BUS and General-Purpose

This register is not used in these modes.

Address: 0x400012A - REG_SCCNT_H (Serial Communication Source Register)

As with the other serial registers, this info comes from the no$gba documentation.

Normal

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  S S S S  S S S S

0-7 (S) = The 8-bit data to sent/received over the link cable.
```

For 8-bit normal mode only. Contains 8-bit data (only the lower 8bit are used). Outgoing data should be written to this register before starting the transfer. During transfer, transmitted bits are shifted-out (MSB first), and received bits are shifted-in simultaneously. Upon transfer completion, the register contains the received 8-bit value.

Multiplayer

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
S S S S  S S S S  S S S S  S S S S

0-F (S) = The 16-bit data to be sent over the link cable.
```

UART

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  S S S S  S S S S

0-7 (S) = The 8-bit data to sent/received over the link cable.
```

This is the send/receive shift register, or (when FIFO is used) the send/receive FIFO. The send/receive FIFO may store up to four 8-bit data units each. For example, while 1 unit is still being transferred from the send shift register, it is possible to deposit another 4 units in the send FIFO, which are then automatically moved to the send shift register one after each other.

Addresses 0x4000130 - 0x4000132 - Keypad Input and Control Registers

Address: 0x4000130 - REG_KEY (The input register)(Read Only)

```
              R R  R R R R  R R R R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X J I  D U L R  S E B A

0 (A) = A button
1 (B) = B button
2 (E) = Select button
3 (S) = Start button
4 (R) = D-pad Right
5 (L) = D-pad Left
6 (U) = D-pad Up
7 (D) = D-pad Down
8 (I) = Right shoulder button
9 (J) = Left shoulder button
```

This register stores the state of the GBA's buttons. Each of the inputs is active low. This means that a '0' bit indicates that the key is pressed, while a '1' bit indicates that the key is not pressed. In general a game which samples these (rather than using interrupts) should do so at least once every refresh (60hz), or more in the case of fast action fighting games (like Street Fighter).

Address: 0x4000132 - REG_P1CNT (Key Control Register - As yet unimplemented in CowBite)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
T K X X  X X J I  D U L R  S E B A


0 (A) = A button
1 (B) = B button
2 (E) = Select button
3 (S) = Start button
4 (R) = D-pad Right
5 (L) = D-pad Left
6 (U) = D-pad Up
7 (D) = D-pad Down
8 (I) = Right shoulder button
9 (J) = Left shoulder button
E (K) = generate interrupt on keypress
F (T) = interrupt "type"
```

```
0: "OR" operation -- interrupt will be generated if _any_ of specified
   keys (bits 0-9) is pressed
1: "AND" operation, interrupt will be generated if _all_ specified keys
   are pressed at the same time.
```

Use this register to set which keypresses generate interrupts. The appropriate bits must also be set in REG_IE and REG_IME.

Address: 0x4000134 - REG_RCNT

```
               *  * * * *  * * * *
F E D C  B A 9 8  7 6 5 4  3 2 1 0
M M X X  X X X I  Q P O N  M L K J

0 (J) = SC Data Bit (* GP Mode)
1 (K) = SD Data Bit (* GP Mode)
2 (L) = SI Data Bit (* GP Mode)
3 (M) = SO Data Bit (* GP Mode)

4 (N) = SC Direction (0 - Input, 1 - Output) (* GP Mode)
5 (O) = SD Direction (0 - Input, 1 - Output) (* GP Mode)
6 (P) = SI Direction (0 - Input, 1 - Output) (* GP Mode)
7 (Q) = SO Direction (0 - Input, 1 - Output) (* GP Mode)

8 (I) = Interrupt request enable. (* GP Mode) Interrupts can be requested when SI changes
        from HIGH to LOW, as General Purpose mode does not require a serial shift clock,
        this interrupt may be produced even when the GBA is in Stop (low power standby)
        state.

E-F (M) = Mode selection
          0 X - Normal, Multiplayer, UART modes
          1 0 - General-Purpose Mode
          1 1 - JOY BUS Mode

* General Purpose Mode Only
```

Thanks to Martin Korth for all the details about this register, found in his no$gba documentation.

The function of this register varies depending on the desired transfer mode. The mode bits should be initialized before REG_SCCNT_L. In all modes but General-Purpose mode, bits 0 - D go unused. In General-Purpose mode, this register serves as a 4bit bi-directional parallel port, directly controlling the individual SI,SO,SC, and SD pins. Each can be separately declared as input (with internal pull-up) or as output signal.

Address: 0x4000140 - REG_JOYCNT (JOY BUS Control Register)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
M M X X  X X X I  X X X X  X S C R

0 (R) = Device Reset Flag (Command 0xFF)
1 (C) = Receive Complete Flag (Command 0x14 or 0x15?)
2 (S) = Send Complete Flag (Command 0x14 or 0x15?)
8 (I) = IRQ when receive a Device Reset Command.
```

Thanks to Martin Korth for all the details about this register, found in his no$gba documentation.

Bits 0 - 1 of this register work similarly to the bits in REG_IF; write a "1" to them to reset them and "acknowledge" that the incoming data was processed.

Address: 0x4000150 - REG_JOYRE_L
Address: 0x4000152 - REG_JOYRE_H
Address: 0x4000154 - REG_JOYTR_L
Address: 0x4000156 - REG_JOYTR_H

These registers are used to send and recieve data in JOY BUS mode.

Address: 0x4000158 - REG_JOYSTAT (JOY BUS Receive Status Register)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
M M X X  X X X X  X X G G  S X R X

1   (R) = Receive Status Flag (0 - Remove GBA is/was recieving)
3   (S) = Send Status Flag (1 - Remot GBA is/was sending)
4-5 (G) = General use flag.  Not assigned for any specific purpose.
```

Thanks to Martin Korth for all the details about this register, found in his no$gba documentation.

Bit 1 of this register is automatically set when writing to REG_JOYTR, and bit 3 is automatically reset when reading from REG_JOYRE.

Possible commands received in JOY BUS mode (not possible to issue commands):

```
Command FFh - Device Reset

  Receive FFh (Command)
  Send    00h (GBA Type number LSB (or MSB?))
  Send    04h (GBA Type number MSB (or LSB?))
  Send    XXh (lower 8bits of SIOSTAT register)

Command 00h - Type/Status Data Request

  Receive 00h (Command)
  Send    00h (GBA Type number LSB (or MSB?))
  Send    04h (GBA Type number MSB (or LSB?))
  Send    XXh (lower 8bits of SIOSTAT register)
```

```
Command 15h - GBA Data Write (to GBA)

  Receive 15h (Command)
  Receive XXh (Lower 8bits of JOY_RECV_L)
  Receive XXh (Upper 8bits of JOY_RECV_L)
  Receive XXh (Lower 8bits of JOY_RECV_H)
  Receive XXh (Upper 8bits of JOY_RECV_H)
  Send    XXh (lower 8bits of SIOSTAT register)

Command 14h - GBA Data Read (from GBA)

  Receive 14h (Command)
  Send    XXh (Lower 8bits of JOY_TRANS_L)
  Send    XXh (Upper 8bits of JOY_TRANS_L)
  Send    XXh (Lower 8bits of JOY_TRANS_H)
  Send    XXh (Upper 8bits of JOY_TRANS_H)
  Send    XXh (lower 8bits of SIOSTAT register)
```

Addresses 0x4000200 - 0x4000208 - Interrupt Registers

Address: 0x4000200 - REG_IE (Interrupt Enable Register)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X T Y  G F E D  S L K J  I C H V

0 (V) = VBlank Interrupt
1 (H) = HBlank Interrupt
2 (C) = VCount Interrupt
3 (I) = Timer 0 Interrupt
```

```
4 (J) = Timer 1 Interrupt
5 (K) = Timer 2 Interrupt
6 (L) = Timer 3 Interrupt
7 (S) = Serial Communication Interrupt
8 (D) = DMA0 Interrupt
9 (E) = DMA1 Interrupt
A (F) = DMA2 Interrupt
B (G) = DMA3 Interrupt
C (Y) = Key Interrupt
D (T) = Cassette Interrupt
```

Use this register to mask out which interrupts are enabled or disabled.

Address: 0x4000202 - REG_IF (Interrupt Flags Regster)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X T Y  G F E D  S L K J  I C H V

0 (V) = VBlank Interrupt
1 (H) = HBlank Interrupt
2 (C) = VCount Interrupt
3 (I) = Timer 0 Interrupt
4 (J) = Timer 1 Interrupt
5 (K) = Timer 2 Interrupt
6 (L) = Timer 3 Interrupt
7 (S) = Serial Communication Interrupt
8 (D) = DMA0 Interrupt
9 (E) = DMA1 Interrupt
A (F) = DMA2 Interrupt
B (G) = DMA3 Interrupt
C (Y) = Key Interrupt
D (T) = Cassette Interrupt
```

This register will determine which interrupt is currently being serviced. When your interrupt service routine get scalled, check these flags to determine what called it. In order to keep yourself from servicing the wrong interrupt at a later time, you should reset the flags to 0 by writing a 1 to them.

Address: 0x4000204 - REG_WSCNT (Wait State Control)

```
R
F E D C  B A 9 8  7 6 5 4  3 2 1 0
G P X C  C N M M  L K K J  I I S S

0-1 (S) = SRAM wait state
          00:  4 cycles          01:  3 cycles
          10:  2 cycles          11:  8 cycles
2-3 (I) = Bank 0x08000000 initial wait state
          00: 4 cycles           01: 3 cycles
          10: 2 cycles           11: 8 cycles
4   (J) = Bank 0x08000000 subsequent wait state
          0: _2_ cycles  1: 1 cycle
5-6 (K) = Bank 0x0A000000 initial wait state
          00: 4 cycles           01: 3 cycles
          10: 2 cycles           11: 8 cycles
7   (L) = Bank 0x0A000000 subsequent wait state
          0: _4_ cycles  1: 1 cycle
8-9 (M) = Bank 0x0C000000 initial wait state
          00: 4 cycles           01: 3 cycles
          10: 2 cycles           11: 8 cycles
A   (N) = Bank 0x0C000000 subsequent wait state
          0: _8_ cycles          1: 1 cycle
```

```
B-C (C) = Cart clock.  Don't touch.
          00:  Terminal output clock fixed lo
          01:  4 Mhz
          10:  8 Mhz
          11:  16 Mhz
E   (P) = Prefetch.  When enabled, the GBA attempts to read opcodes from ROM
          when the CPU is not using the bus.  The 8-word-by-16-bit prefetch
          buffer makes subsequent ROM reads faster (0 waits) in code that
          accesses both ROM and RAM.
          0: Disable (and save battery power)
          1: Enable
F   (G) = Game Pak type
          0: AGB multiplexed bus
          1: DMG/CGB bus
```

Use this register to control wait state settings and the prefetch buffer for ROM and SRAM. Thanks to Damian Yerrick for contributing this info, and for pointing me to some relevant reading material.

Address: 0x4000208 - REG_IME (Interrupt Master Enable)

```
7 6 5 4  3 2 1 0
X X X X  X X X M

0 (M) = Master interrupt enable. When off, all interrupts are disabled.  This
        must be on for the interrupt bits in REG_IE to have any effect.
```

Address: 0x4000300 - REG_HALTCNT_L (First Boot/Debug Control - Unimplemented in CowBite)

```
7 6 5 4  3 2 1 0
X X X X  X X X F

0 (F) = First Boot Flag (0 = First, 1 = Further)
```

Thanks to Martin Korth for the details of this register. The following comes from his description in the no$gba documentation.

After initial reset, the GBA BIOS initializes the register to 0x1, and any further execution of the Reset vector (0x00000000) will pass control to the Debug vector (0x0000001C) when sensing the register to be still set to 0x1. Normally the debug handler rejects control unless it detects Debug flags in cartridge header, in that case it may redirect to a cut-down boot procedure (bypassing Nintendo logo and boot delays).

Address: 0x4000301 - REG_HALTCNT_H (Low Power Mode Control - Unimplemented in CowBite)

```
7 6 5 4  3 2 1 0
M X X X  X X X X

7 (M) = Power Down Mode (0 = Halt, 1 = Stop)
```

Thanks to Martin Korth for the details of this register. The following comes from his description in the no$gba documentation.

In Halt mode, the CPU is paused until an interrupt occurs, this should be used to reduce power-consumption during periods when the CPU is waiting for interrupt events. In Stop mode, most of the hardware including sound and video are paused, this very-low-power mode could be used much like a screensaver.

The current GBA BIOS addresses only the upper eight bits of this register (by writing 0x00 or 0x80 to address 0x04000301), however, as the register isn't officially documented, some or all of the bits might have different meanings in future GBA models. For best forwards compatibility, it'd generally be more recommended to use the BIOS Functions SWI 2 (Halt) or SWI 3 (Stop) rather than writing to this register directly.

Address: 0x4000800 - REG_IMC_L (Internal Memory Control - Unimplemented in CowBite)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  X X X X  X X M X  L K J I

0 (I) = Purpose unkown.  When set, locks the GBA or sets the top-left
        tile number to 0x82.
1 (J) = Unknown.
2 (K) = Unknown.
3 (L) = Unkown.
5 (M) = When unset this locks up the GBA or just disallows VRAM changes
        (but can be used simulteneously with bit 0).  It is set by default.
```

This register is undocumented even for official developers. What little we know about this it comes separately from Martin Korth's no$gba documentaion and gbcft's experimentations. gbcft comments that he did his testing a while back and is unsure about his assessment of bits 0 and 5. Anybody who knows about it can feel free to email me with more info.

This register (the whole 32-bit word, including REG_IMC_H) is mirrored at intervals of 0x10000 bytes in IORAM (it is the only IORAM
register that is mirrored in in this way).

Address: 0x4000802 - REG_IMC_H (Internal Memory Control - Unimplemented in CowBite)

```
F E D C  B A 9 8  7 6 5 4  3 2 1 0
X X X X  W W W W  X X X X  X X X X

8 - B (W)= WRAM Wait state control.
           1101 - 2 Waitstates (3/3/6 cycles 8/16/32bit accesses)
           1110 - 1 Waitstate (2/2/4 cycles for 8/16/32bit acceses)

           Default setting appears to be 1101.  A value of 1111 causes the
           system to lock up.
```

Thanks to Martin Korth for detailing the function of this registers in his no$gba documentation. Note that the benefits of this register are not guaranteed to work on all GBAs and may very well have undesirable results if tampered with! In other words, if you use this register at all, do so in your own private tests.

---

11. Miscellaneous/Weirdness

This is a new section devoted any interesting little tidbits that don't quite fit in anywhere else. I'm starting it off with . . .

Unknown Registers

These are registers that I don't know the function of. Gradually their numbers have dwindled down to almost nothing.While most aren't even readable, certain ones seem to be writeable. I have an idea of what some of them are based on the headers included with many open source demos, but as to their specific function, I have no idea. Let me know if you find out what purpose they serve, if any, and I will put your notes here (and of course give credit).

```
0x002:          REG_DISPCNT_H - Writing to bit 0 will cause (horizontally)
                adjacent pairs of pixels to have their green component
                swapped. What good is this for?  Anybody know?

0x04E:          REG_MOSAIC_H - Unreadable (gobbledygook)
0x056 - 0x05E:  Unreadable (gobbledygook)
0x066:          REG_SOUND1CNT_X (high 16 bits) Unreadable (0x0000)
0x06A:          REG_SOUND2CNT_L (high 16 bits) Unreadable (0x0000)
0x06E:          REG_SOUND2CNT_H (high 16 bits) Unreadable (0x0000)
0x076:          REG_SOUND3CNT_X (high 16 bits) Unreadable (0x0000)
0x07A:          REG_SOUND4CNT_L (high 16 bits) Unreadable (0x0000)
0x07E:          REG_SOUND4CNT_L (high 16 bits) Unreadable (0x0000)
0x086:          REG_SOUNDCNT_X (high 16 bits) Unreadable (0x0000)
0x08A:          REG_SOUNDBIAS (high 16 bits) Unreadable (0x0000)
0x08C:          Unreadable (gobbledygook)
0x08E:          Unreadable (gobbledygook)
0x0A8 - 0xAE:   Unreadable (gobbledygook)
0x0E0 - 0x0FE:  Unreadable (gobbledygook)
0x110 - 0x11E:  Unreadable (gobbledygook)
0x12C:          Unreadable (gobbledygook)
0x12E:          Unreadable (gobbledygook)
0x136:          REG_IR (high 16 bits) Unreadable (0x0000).
                Served as the Infrared register on prototypes.
0x138 - 13E:    Unreadable (gobbledygook)
0x142:          Unreadable (0x0000)
0x144-14E:      Unreadable (gobbledygook)
0x15A:          JOYSTAT_H - Unreadable (0x0000)
0x15C - 0x1FE:  Unreadable (gobbledygook)
0x206:          REG_WSCNT (high 16 bits) Unreadable (0x0000)
0x20A:          REG_IME (high 16 bits) Unreadable (0x0000)
0x20C - 0x2FE:  Unreadable (gobbledygook)
0x302:          REG_PAUSE (high 16 bits) Unreadable (0x0000)
0x304 - 0x3FE:  Unreadable (gobbledygook)
0x410:          Martin Korth has observed that the BIOS writes 0xFF (8 bit) to this address on startup.
```

I have noted that some of these registers always read as a certain repeating gobbledygook value no matter what is written to them (I take this to be of the same origin as the gobbledygook that comes from reading BIOS -- see Jeff Frohwein's GBA FAQ), but some always read 0. For the most part these seem to be the "high" 16 bits of registers we already know the function of (thanks gbcft for pointing out what probably should have been obvious to me). Others are registers that probably do something, but I just don't know what. They may read as something other than 0 if the conditions are right.

Possibilities:
The following are some suggested possiblities for what the unkown registers could be (thanks goes out to Kay for this), based on what Nintendo has included in previous hardware:

SPR_TIME_OVER bit: sprite blit time out register (1 bit). This bit is set when sprite processing overloads during HDraw
SPR_RANGE_OVER bit: displayed objects number (regardless of the size) becomes or is greater than maximum GFX ASIC is able to process
xxxx_VER_NUMBER reg: regsiter containing the xxxx IC version number (usually bit 0 to 3 if made by Nintendo)

- Martin Korth notes that there also appears to be an undocumented register that is used to mask out cartridge memory (except first 4KBytes of ROM) in Single Game Pak slave mode . ..

- Like all Nintendo hardware designs, the GBA may have one or more 8/16 bit in/out parallel data ports mapped at the bottom of I/O addresses.

12. Links

The following are links to other documents which should prove useful to GBA developers:

http://www.gbadev.org/ (A great resource for GBA developers)
Devrs.com GBA Site (Another great resource, maintained by Jeff Frohwein)
Jeff Frohwein's GBA Dev FAQ
The Pern Project (Tutorials for GBA development)
Mappy SDK (A document similar to this one but for Joat's Mappy emulator)
ARM's Technical Reference Manuals (Docs that you should have around if you want to understand the ARM processor)
The Audio Advance (A great site by Uze explaining the GBA's sound system)
Andrew May's site (Technical info on the GBA link port)

CowBite (The emulator I've been writing off and on. Make sure you check it out and give me feedback!)
no$gba (A good asm debugger/emulator for commercial developers with VERY good documentation on the GBA hardware. Free version of no$gba is compatible with multiboot games.)
Mappy (A great emulator for asm debugging and graphical feedback on sprites, backgrounds, etc.)
Visual Boy Advance (A very accurate emulator that is gdb compatible)

13. Thanks

The following are individuals who contributed info or corrections to this document. Thanks guys!

Agent Q (Wrote the original spec, version 1.0)
Uze (All of the sound register info comes directly from his Audio Advance site)
Martin Korth (Author no$gba of who has given me permission to consolidate additional info from his emulator's informative help documents with this one, most particularly serial registers, some BIOS functions, and undocumented registers.)
Forgotten (VBA Author. Many of the BIOS call descriptions come from his Visual Boy Advance FAQ.
gbcft (LOTS of info on interrupts, windowing, memory mirrors, the "Unkown Registers" section; helped me debug a lot of errors in the emulator, and offered many corrections, info, and suggestions).
Kay (Contributed memory port sizes and wait states, DMA cycle timings, info regarding the BIOS, and various advice, testing, and expertise regarding the GBA and older console systems)
Damian Yerrick (Contributed the WSCOUNT register)
Markus (Actually I asked him for help with LZSS. Also, his gfx2gba tool has proven *extremely* helpful in my non-CowBite projects.:)

ePac (Gave me links to serial info and did a nice writeup about it in the gbadev group)
Costis (A variety of new info/corrections)
Grauw (Info on forced blanking, hblank lenghths, and on the BIOS wait function.)
Max
Otaku
Ped (Pointed out errors in the memory ranges, DISPCNT bit 5, and a bad typo regarding rotates/scale backgrounds).
Yarpen (Almost all the information on the timer registers and the keyboard control register. Thanks!)

General Thanks

All those GBA demo authors. You rock!
Every other emu author
http://www.gbadev.org/
The gbadev list on yahoo
SimonB and all the others who run/moderate the above sites
Dovoto and the PERN Project
Jeff Frohwein and his Devrs.com site
Nocturn and his tutorials
Uze from BeLogic for all the great information on the GBA's sound!
Andrew May for his site on GBA serial data
Nintendo

If I've forgotten to list your name or an important resource in this section, let me know.